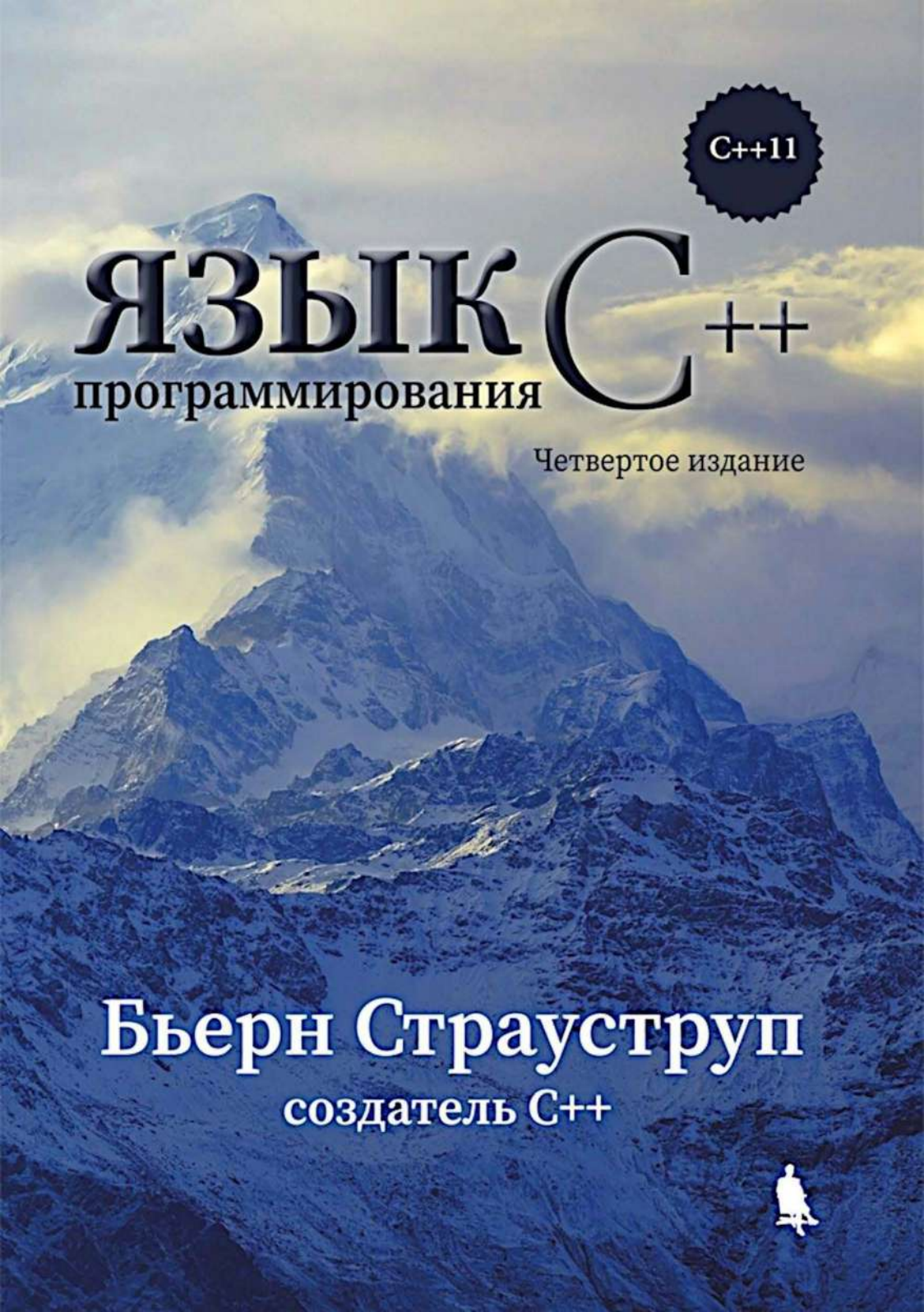
C++11
ЯЗЫК C++
программирования
Четвертое издание
Бьерн Страуструп
создатель C++

Б. Страуструп

Язык программирования C++ для профессионалов

Учебник

4-е издание стереотипное
электронное

Интернет-Университет
Информационных Технологий
www.intuit.ru

Ай Пи Ар Медиа

Москва
2025

УДК 004
ББК 32.97

Страуструп, Б.
Язык программирования C++ для профессионалов : учебник / Б. Страуструп. — 4-е изд., стер. электрон. — Москва : Национальный Открытый Университет «ИНТУИТ» : Ай Пи Ар Медиа, 2025. — 670 с. — Текст : электронный.

ISBN 978-5-4497-0922-6

В учебнике дается описание языка C++, его ключевых понятий и основных приемов программирования на нем. Это завершенное руководство, написанное создателем языка, которое содержит описание всех средств C++, в том числе управление исключительными ситуациями, шаблоны типа (параметризованные типы данных) и множественное наследование.

C++ является языком программирования общего назначения. Естественная для него область применения — системное программирование, понимаемое в широком смысле этого слова. Кроме того, C++ успешно используется во многих областях приложения, далеко выходящих за указанные рамки. Реализации C++ есть на всех машинах, начиная с самых скромных микрокомпьютеров — до самых больших супер-ЭВМ, и практически для всех операционных систем. Поэтому книга дает лишь описание собственно языка, не объясняя особенности конкретных реализаций, среды программирования или библиотек.

Учебное электронное издание

Технический редактор *А.Д. Матлахова*
Обложка *С.С. Сизиумова, Я.А. Кирсанов*

Содержание

Об издании

Курс, созданный по книге Б. Страуструпа "Язык программирования C ++", дает описание языка, его ключевых понятий и основных приемов программирования на нем. Это завершенное руководство, написанное создателем языка, которое содержит описание всех средств C++, в том числе управление исключительными ситуациями, шаблоны типа (параметризованные типы данных) и множественное наследование.

Книга делится на три части. Первые десять лекций являются учебником, служащим введением в язык, включая подмножество собственно C. В трех последующих лекциях обсуждаются вопросы проектирования и создания программного обеспечения с помощью C ++. Книга завершается полным справочным руководством по языку.

В книге вы найдете:

- законченный учебник и руководство по языку.
- полное освещение средств языка, нацеленных на абстрактные типы данных и объектно-ориентированное программирование.
- обсуждение программистских и технических вопросов, возникающих в процессе проектирования и создания больших программных систем.
- описание способов построения библиотек высокого класса.
- примеры реализации ключевых типов данных, определяемых пользователем, таких как графические объекты, ассоциативные массивы и потоки ввода- вывода.

Эта книга будет хорошим помощником опытному программисту, решившему использовать C++ для нетривиальных задач. Ее можно считать ключевой в любом собрании книг по C++.

Об авторе книги:

Бьерн Страуструп является разработчиком языка C++ и создателем первого транслятора. Он - сотрудник научно-исследовательского вычислительного центра AT&T Bell Laboratories в Мюррей Хилл (Нью-Джерси, США). Он получил звание магистра математики и вычислительной техники в университете г. Аарус (Дания), а докторское

звание по вычислительной технике в кэмбриджском университете (Англия). Он специализируется в области распределенных систем, операционных систем, моделирования и программирования. Вместе с М. А. Эллис он является автором полного руководства по языку C++ - "Руководство по C++ с примечаниями".

Предисловие

"А дорога идет все дальше и дальше" (Бильбо Бэгинз)

Как было обещано в первом издании книги, запросы пользователей определили развитие C++. Его направлял опыт широкого круга пользователей, работающих в разных областях программирования. За шесть лет, отделяющих нас от первого издания описания C++, число пользователей возросло в сотни раз. За эти годы были усвоены многие уроки, были предложены и подтвердили практикой свое право на существование различные приемы программирования. О некоторых из них и пойдет речь ниже.

Сделанные за эти шесть лет расширения языка прежде всего были направлены на повышение выразительности C++ как языка абстракции данных и объектно-ориентированного программирования вообще и как средства для создания высококачественных библиотек с пользовательскими типами данных в частности. Библиотекой высокого качества мы считаем библиотеку, позволяющую пользователю определять с помощью классов понятия, работа с которыми сочетает удобство, эффективность и надежность. Под надежностью понимается то, что класс предоставляет защищенный по типам интерфейс между пользователями библиотеки и ее разработчиками. Эффективность предполагает, что использование классов не влечет за собой больших накладных расходов по памяти или времени по сравнению с "ручными" программами на C.

Эта книга является полным описанием языка C++. Лекции с 1 по 10 представляют собой учебник, знакомящий с языком. В лекциях с 11 по 13 обсуждаются вопросы проектирования и развития программного обеспечения. Завершается книга справочным руководством по языку C ++. Естественно, что все расширения языка и способы их использования, которые появились после выхода в свет первого

издания, являются частью изложения. К ним относятся уточненные правила для разрешения перегрузки имени, средства управления памятью и средства контроля доступа, надежная по типам процедура связывания, статические и постоянные функции-члены, абстрактные классы, множественное наследование, шаблоны типов и обработка особых ситуаций.

C++ является языком программирования общего назначения. Естественная для него область применения - системное программирование, понимаемое в широком смысле этого слова. Кроме того, C++ успешно используется во многих областях приложения, далеко выходящих за указанные рамки. Реализации C++ теперь есть на всех машинах, начиная с самых скромных микрокомпьютеров - до самых больших супер-ЭВМ, и практически для всех операционных систем. Поэтому книга дает лишь описание собственно языка, не объясняя особенности конкретных реализаций, среды программирования или библиотек.

Читатель найдет в книге много примеров с классами, которые, несмотря на несомненную пользу, можно считать игрушечными. Такой стиль изложения позволяет лучше выделить основные понятия и полезные приемы, тогда как в настоящих, законченных программах они были бы скрыты массой деталей. Для большинства предложенных здесь классов, как то связанные списки, массивы, строки символов, матрицы, графические классы, ассоциативные массивы и т.д., - приводятся версии "со 100% гарантией" надежности и правильности, полученные на основе классов из самых разных коммерческих и некоммерческих программ. Многие из "промышленных" классов и библиотек получились как прямые или косвенные потомки игрушечных классов, приводимых здесь как примеры.

В этом издании книги по сравнению с первым больше внимания уделено задаче обучения. Вместе с тем, уровень изложения в равной мере учитывает и опытных программистов, ни в чем не умаляя их знаний и профессионализма. Обсуждение вопросов проектирования сопровождается более широкой подачей материала, выходящей за рамки описаний конструкций языка и способам их использования. В этом издании приводится больше технических деталей и повышена строгость изложения. В особенности это относится к справочному руководству,

которое вобрало в себя многолетний опыт работы в этом направлении. Предполагалось создать книгу с достаточно высоким уровнем изложения, которая бы служила программистам не только книгой для чтения. Итак, перед вами книга с описанием языка C++, его основных принципов и методов программирования. Надеемся, что она доставит вам радость.

Выражение признательности

Кроме лиц, перечисленных в соответствующем разделе предисловия к первому изданию книги, мне хотелось бы выразить свою благодарность Элу Эхо, Стиву Бароффу, Джиму Коплину, Тому Хансену, Петеру Джаглу, Брайану Кернигану, Эндрю Кенигу, Биллу Леггету, Лоррейн Мингаччи, Уоррену Монтгомери, Майку Моубри, Робу Мюррею, Джонатану Шапиро, Майку Вилоту и Петеру Вейнбергу за комментарии черновых вариантов второго издания книги. В развитии языка C++ за период от 1985 до 1991 гг. принимали участие многие специалисты. Я могу упомянуть лишь нескольких из них: Эндрю Кенига, Брайана Кернигана, Дага Макилроя и Джонатана Шапиро. Кроме того, выражаю признательность многим участникам создания справочного руководства C++, предложившим свои варианты, а также тем, с кем довелось нести тяжкую ношу в течение первого года работы комитета X3J16 по стандартизации языка C++.

Мюррей-Хилл, шт.Нью Джерси Бьерн Страуструп

Предварительные замечания

"О многом - молвил Морж,- пришла пора поговорить ". Л.Кэрролл

Данная лекция содержит краткий обзор книги и некоторые дополнительные замечания о языке C++. Замечания касаются истории создания C++, идей, которые оказали существенное влияние на разработку языка, и некоторых мыслей по поводу программирования на C++. Эта лекция не является введением; приведенные замечания не являются необходимыми для понимания последующих лекций. Некоторые из них предполагают знакомство читателя с C++.

Структура книги

Книга состоит из трех частей. Лекции с 1 по 10 являются учебником по языку. В лекциях с 11 по 13 обсуждаются вопросы проектирования и развития программного обеспечения с учетом возможностей C++. В конце книги приведено полное справочное руководство по языку. Исчерпывающее описание конструкций C++ содержится только там. Учебная часть книги содержит примеры, советы, предостережения и упражнения, для которых не нашлось места в руководстве.

Книга в основном посвящена вопросу, как с помощью языка C++ структурировать программу, а не вопросу, как записать на нем алгоритм. Следовательно, там, где можно было выбирать, предпочтение отдавалось не профессиональным, но сложным для понимания, а тривиальным алгоритмам. Так в одном из примеров используется пузырьковая сортировка, хотя алгоритм быстрой сортировки больше подходит для настоящей программы. Часто написать ту же программу, но с более эффективным алгоритмом, предлагается в виде упражнения.

Лекция 1 содержит краткий обзор основных концепций и конструкций C++. Она позволяет познакомиться с языком в общих чертах. Подробные объяснения конструкций языка и способов их применения содержатся в последующих главах. Обсуждаются в первую очередь средства, обеспечивающие абстракцию данных и объектно-ориентированное программирование. Основные средства процедурного программирования упоминаются кратко.

В лекциях 2, 3 и 4 описываются средства C++, которые не используются для определения новых типов: основные типы, выражения и структуры управления. Другими словами, эти главы содержат описание той части языка, которая по сути представляет C. Изложение в указанных лекциях идет в углубленном виде.

Лекции 5 - 8 посвящены средствам построения новых типов, которые не имеют аналогов в C. В лекции 5 вводится основное понятие - класс. В ней показано, как можно определять пользовательские типы (классы), инициализировать их, обращаться к ним, и, наконец, как уничтожать их. Лекция 6 посвящена понятию производных классов, которое позволяет строить из простых классов более сложные. Оно дает также

возможность эффективной и безопасной (в смысле типа) работы в тех ситуациях, когда типы объектов на стадии трансляции неизвестны. В лекции 7 объясняется, как можно определить унарные и бинарные операции для пользовательских типов, как задавать преобразования таких типов, и каким образом можно создавать, копировать и удалять объекты, представляющие пользовательские типы. Лекции 8 посвящена шаблонам типа, т.е. такому средству C++, которое позволяет определить семейство типов и функций.

В лекции 9 обсуждается обработка особых ситуаций, рассматриваются возможные реакции на ошибки и методы построения устойчивых к ошибкам систем. В лекции 10 определяются классы `ostream` и `istream`, предоставляемые стандартной библиотекой для потокового ввода-вывода.

Лекции 11 - 13 посвящены вопросам, связанным с применением C++ для проектирования и реализации больших программных систем. В лекции 11 в основном рассматриваются вопросы проектирования и управления программными проектами. В лекции 12 обсуждается взаимосвязь между языком C++ и проблемами проектирования. В лекции 13 показаны способы создания библиотек.

Завершается книга справочным руководством по C++.

Ссылки на различные части книги даются в виде $2.3.4, что означает раздел 3.4 лекции 2. Для обозначения справочного руководства применяется буква R, например, $.8.5.5.

Замечания по реализации

Существует несколько распространяемых независимых реализаций C++. Появилось большое число сервисных программ, библиотек и интегрированных систем программирования. Имеется масса книг, руководств, журналов, статей, сообщений по электронной почте, технических бюллетеней, отчетов о конференциях и курсов, из которых можно получить все необходимые сведения о последних изменениях в C++, его использовании, сервисных программах, библиотеках, новых трансляторах и т.д. Если вы серьезно рассчитываете на C++, стоит

получить доступ хотя бы к двум источникам информации, поскольку у каждого источника может быть своя позиция.

Большинство программных фрагментов, приведенных в книге, взяты непосредственно из текстов программ, которые были транслированы на машине DEC VAX 11/8550 под управлением 10-й версии системы UNIX [25]. Использовался транслятор, являющийся прямым потомком транслятора C++, созданного автором. Здесь описывается "чистый C ++", т.е. не используются никакие зависящие от реализации расширения. Следовательно, примеры должны идти при любой реализации языка. Однако, шаблоны типа и обработка особых ситуаций относятся к самым последним расширениям языка, и возможно, что ваш транслятор их не содержит.

Упражнения

Упражнения даются в конце каждой лекции. Чаще всего они предлагают написать программу. Решением может считаться программа, которая транслируется и правильно работает хотя бы на нескольких тестах. Упражнения могут значительно различаться по сложности, поэтому дается приблизительная оценка степени их сложности. Рост сложности экспоненциальный, так что, если на упражнение (*1) у вас уйдет пять минут, то (*2) может занять час, а (*3) - целый день. Однако время написания и отладки программы больше зависит от опыта читателя, чем от самого упражнения. На упражнение (*1) может потребоваться целый день, если перед запуском программы читателю придется ознакомиться с новой вычислительной системой. С другой стороны, тот, у кого под рукой окажется нужный набор программ, может сделать упражнение (*5) за один час.

Любую книгу по программированию на языке C можно использовать как источник дополнительных упражнений при изучении лекций 2 - 4. В книге Ахо ([1]) приведено много общих структур данных и алгоритмов в терминах абстрактных типов данных. Эту книгу также можно использовать как источник упражнений при изучении лекций 5 - 8. Однако, использованному в ней языку не достает функций-членов и производных классов. Поэтому определяемые пользователем типы на C ++ можно написать более элегантно.

Замечания по проекту языка

При разработке языка C++ одним из важнейших критериев выбора была простота. Когда возникал вопрос, что упростить: руководство по языку и другую документацию или транслятор, - то выбор делали в пользу первого. Огромное значение придавалось совместимости с языком C, что помешало удалить его синтаксис.

В C++ нет типов данных и элементарных операций высокого уровня. Например, не существует типа матрица с операцией обращения или типа строка с операцией конкатенации. Если пользователю понадобятся подобные типы, он может определить их в самом языке. Программирование на C++ по сути сводится к определению универсальных или зависящих от области приложения типов. Хорошо продуманный пользовательский тип отличается от встроенного типа только способом определения, но не способом применения.

Из языка исключались возможности, которые могут привести к накладным расходам памяти или времени выполнения, даже если они непосредственно не используются в программе. Например, было отвергнуто предложение хранить в каждом объекте некоторую служебную информацию. Если пользователь описал структуру, содержащую две величины, занимающие по 16 разрядов, то гарантируется, что она поместится в 32-х разрядный регистр.

Язык C++ проектировался для использования в довольно традиционной среде, а именно: в системе программирования C операционной системы UNIX. Но есть вполне обоснованные доводы в пользу использования C++ в более богатой программной среде. Такие возможности, как динамическая загрузка, развитые системы трансляции и базы данных для хранения определений типов, можно успешно использовать без ущерба для языка.

Типы C++ и механизмы упрятывания данных рассчитаны на определенный синтаксический анализ, проводимый транслятором для обнаружения случайной порчи данных. Они не обеспечивают секретности данных и защиты от умышленного нарушения правил доступа к ним. Однако, эти средства можно свободно использовать, не боясь накладных расходов памяти и времени выполнения программы.

Учтено, что конструкция языка активно используется тогда, когда она не только изящно записывается на нем, но и вполне по средствам обычным программам.

Историческая справка

Безусловно C++ многим обязан языку C [8], который сохраняется как его подмножество. Сохранены и все свойственные C средства низкого уровня, предназначенные для решения самых насущных задач системного программирования. C, в свою очередь, многим обязан своему предшественнику языку BCPL [13]. Комментарий языка BCPL был восстановлен в C++. Если читатель знаком с языком BCPL, то может заметить, что в C++ по-прежнему нет блока VALOF. Еще одним источником вдохновения был язык SIMULA-67 [2,3]; именно из него была заимствована концепция классов (вместе с производными классами и виртуальными функциями). Оператор inspect из SIMULA-67 намеренно не был включен в C++. Причина - желание способствовать модульности за счет использования виртуальных функций. Возможность в C++ перегрузки операций и свобода размещения описаний всюду, где может встречаться оператор, напоминают язык Алгол-68 [24].

С момента выхода в свет первого издания этой книги язык C++ подвергся существенным изменениям и уточнениям. В основном это касается разрешения неоднозначности при перегрузке, связывании и управлении памятью. Вместе с тем, были внесены незначительные изменения с целью увеличить совместимость с языком C. Были также введены некоторые обобщения и существенные расширения, как то: множественное наследование, функции-члены со спецификациями static и const, защищенные члены (protected), шаблоны типа и обработка особых ситуаций. Все эти расширения и доработки были нацелены на то, чтобы C++ стал языком, на котором можно создавать и использовать библиотеки. Все изменения описываются в [10,18,20,21 и 23].

Шаблоны типов появились частично из-за желания формализовать макросредства, а частично были инспирированы описанием генерических объектов в языке Ада (с учетом их достоинств и недостатков) и параметризированными модулями языка CLU. Механизм

обработки особых ситуаций появился отчасти под влиянием языков Ада и CLU [11], а отчасти под влиянием ML [26]. Другие расширения, введенные за период между 1985 и 1991 г.г. (такие как множественное наследование, статические функции-члены и чистые виртуальные функции), скорее появились в результате обобщения опыта программирования на C++, чем были почерпнуты из других языков.

Более ранние версии языка, получившие название "C с классами" [16], использовались, начиная с 1980 г. Этот язык возник потому, что автору потребовалось написать программы моделирования, управляемые прерываниями. Язык SIMULA-67 идеально подходит для этого, если не учитывать эффективность. Язык "C с классами" использовался для больших задач моделирования. Строгой проверке подверглись тогда возможности написания на нем программ, для которых критичны ресурсы времени и памяти. В этом языке недоставало перегрузки операций, ссылок, виртуальных функций и многих других возможностей. Впервые C++ вышел за пределы исследовательской группы, в которой работал автор, в июле 1983 г., однако тогда многие возможности C++ еще не были разработаны.

Название C++ (си плюс плюс) , было придумано Риком Маскитти летом 1983 г. Это название отражает эволюционный характер изменений языка C. Обозначение ++ относится к операции наращивания C. Чуть более короткое имя C+ является синтаксической ошибкой. Кроме того, оно уже было использовано как название совсем другого языка. Знатоки семантики C находят, что C++ хуже, чем ++C. Язык не получил названия D, поскольку он является расширением C, и в нем не делается попыток решить какие-либо проблемы за счет отказа от возможностей C. Еще одну интересную интерпретацию названия C++ можно найти в приложении к [12].

Изначально C++ был задуман для того, чтобы автору и его друзьям не надо было программировать на ассемблере, C или других современных языках высокого уровня. Основное его предназначение - упростить и сделать более приятным процесс программирования для отдельного программиста. До недавнего времени не было плана разработки C++ на бумаге. Проектирование, реализация и документирование шли параллельно. Никогда не существовало "проекта C++" или "Комитета по разработке C++". Поэтому язык развивался и продолжает развиваться

так, чтобы преодолеть все проблемы, с которыми столкнулись пользователи. Толчками к развитию служат также и обсуждения автором всех проблем с его друзьями и коллегами.

В связи с лавинообразным процессом увеличения числа пользователей C++, пришлось сделать следующие изменения. Примерно в 1987 г. стало очевидно, что работа по стандартизации C++ неизбежна и что следует незамедлительно приступить к созданию основы для нее [22]. В результате были предприняты целенаправленные действия, чтобы установить контакт между разработчиками C++ и большинством пользователей. Применялась обычная и электронная почта, а также было непосредственное общение на конференциях по C++ и других встречах.

Фирма AT&T Bell Laboratories внесла основной вклад в эту работу, предоставив автору право изучать версии справочного руководства по языку вместе с упоминавшимися разработчиками и пользователями. Не следует недооценивать этот вклад, т.к. многие из них работают в компаниях, которые можно считать конкурентами фирмы AT&T. Менее просвещенная компания могла бы просто ничего не делать, и в результате появилось бы несколько несогласованных версий языка. Около ста представителей из порядка 20 организаций изучали и комментировали то, что стало современной версией справочного руководства и исходными материалами для ANSI по стандартизации C ++. Их имена можно найти в "Аннотированном справочном руководстве по языку C++" [4]. Справочное руководство полностью вошло в настоящую книгу. Наконец, по инициативе фирмы Hewlett-Packard в декабре 1989 г. в составе ANSI был образован комитет X3J16. Ожидается, что работы по стандартизации C++ в ANSI (американский стандарт) станут составной частью работ по стандартизации силами ISO (Международной организации по стандартизации).

C++ развивался одновременно с развитием некоторых фундаментальных классов, представленных в данной книге. Например, автор разрабатывал классы complex, vector и stack, создавая одновременно возможность перегрузки операций. В результате этих же усилий и благодаря содействию Д. Шапиро появились строковые и списочные классы. Эти классы стали первыми библиотечными классами, которые начали активно использоваться. Библиотека task,

описываемая в [19] и в упражнении 13 из $6.8 стала частью самой первой программы, написанной на языке "C с классами". Эта программа и используемые в ней классы были созданы для моделирования в стиле Симулы. Библиотека task была существенно переработана Д. Шапиро и продолжает активно использоваться до настоящего времени. Потоковая библиотека, как указывалось в первом издании книги, была разработана и применена автором. Д. Шварц преобразовал ее в потоковую библиотеку ввода-вывода ($10), используя наряду с другими приемами метод манипуляторов Э.Кенига ($10.4.2). Класс map ($8.8) был предложен Э.Кенигом. Он же создал класс Pool ($13.10), чтобы использовать для библиотеки предложенный автором способ распределения памяти для классов ($5.5.6). На создание остальных шаблонов повлияли шаблоны Vector, Map, Slist и sort, представленные в лекции 8.

Сравнение языков C++ и C

Выбор C в качестве базового языка для C++ объясняется следующими его достоинствами:

1. универсальность, краткость и относительно низкий уровень;
2. адекватность большинству задач системного программирования;
3. он идет в любой системе и на любой машине;
4. полностью подходит для программной среды UNIX.

В C существуют свои проблемы, но в языке, разрабатываемом "с нуля" они появились бы тоже, а проблемы C, по крайней мере, хорошо известны. Более важно то, что ориентация на C позволила использовать язык "C с классами" как полезный (хотя и не очень удобный) инструмент в течение первых месяцев раздумий о введении в C классов в стиле Симулы.

C++ стал использоваться шире, но по мере роста его возможностей, выходящих за пределы C, вновь и вновь возникала проблема совместимости. Ясно, что отказавшись от части наследства C, можно избежать некоторых проблем (см., например, [15]). Это не было сделано по следующим причинам:

1. существуют миллионы строк программ на C, которые можно улучшить с помощью C++, но при условии, что полной переписи их на язык C++ не потребуется;
2. существуют миллионы строк библиотечных функций и служебных программ на C, которые можно было бы использовать в C++ при условиях совместимости обоих языков на стадии связывания и их большого синтаксического сходства;
3. существуют сотни тысяч программистов, знающих C; им достаточно овладеть только новыми средствами C++ и не надо изучать основ языка;
4. поскольку C и C++ будут использоваться одними и теми же людьми на одних и тех же системах многие годы, различия между языками должны быть либо минимальными, либо максимальными, чтобы свести к минимуму количество ошибок и недоразумений. Описание C++ было переработано так, чтобы гарантировать, что любая допустимая в обоих языках конструкция означала в них одно и то же.

Язык C сам развивался в последние несколько лет, что отчасти было связано с разработкой C++ [14]. Стандарт ANSI для C [27] содержит, например, синтаксис описания функций, позаимствованный из языка "C с классами". Происходит взаимное заимствование, например, тип указателя void* был придуман для ANSI C, а впервые реализован в C++. Как было обещано в первом издании этой книги, описание C++ было доработано, чтобы исключить неоправданные расхождения. Теперь C++ более совместим с языком C, чем это было вначале ($.18). В идеале C++ должен максимально приближаться к ANSI C, но не более [9]. Стопроцентной совместимости никогда не было и не будет, поскольку это нарушит надежность типов и согласованность использования встроенных и пользовательских типов, а эти свойства всегда были одними из главных для C++.

Для изучения C++ не обязательно знать C. Программирование на C способствует усвоению приемов и даже трюков, которые при программировании на C++ становятся просто ненужными. Например, явное преобразование типа (приведение) , в C++ нужно гораздо реже, чем в C (см. "Замечания для программистов на C" ниже). Тем не менее, хорошие программы на языке C по сути являются программами на C++. Например, все программы из классического описания C [8] являются

программами на C++. В процессе изучения C++ будет полезен опыт работы с любым языком со статическими типами.

Эффективность и структура

Развитие языка C++ происходило на базе языка C, и, за небольшим исключением, C был сохранен в качестве подмножества C++. Базовый язык C был спроектирован таким образом, что имеется очень тесная связь между типами, операциями, операторами и объектами, с которыми непосредственно работает машина, т.е. числами, символами и адресами. За исключением операций new, delete и throw, а также проверяемого блока, для выполнения операторов и выражений C++ не требуется скрытой динамической аппаратной или программной поддержки.

В C++ используется та же (или даже более эффективная) последовательность команд для вызова функций и возврата из них, что и в C. Если даже эти довольно эффективные операции становятся слишком дорогими, то вызов функции может быть заменен подстановкой ее тела, причем сохраняется удобная функциональная запись безо всяких расходов на вызов функции.

Первоначально язык C задумывался как конкурент ассемблера, способный вытеснить его из основных и наиболее требовательных к ресурсам задач системного программирования. В проекте C++ были приняты меры, чтобы успехи C в этой области не оказались под угрозой. Различие между двумя языками прежде всего состоит в степени внимания, уделяемого типам и структурам. Язык C выразителен и в то же время снисходителен по отношению к типам. Язык C++ еще более выразителен, но такой выразительности можно достичь лишь тогда, когда типам уделяют большое внимание. Когда типы объектов известны, транслятор правильно распознает такие выражения, в которых иначе программисту пришлось бы записывать операции с утомительными подробностями. Кроме того, знание типов позволяет транслятору обнаруживать такие ошибки, которые в противном случае были бы выявлены только при тестировании. Отметим, что само по себе использование строгой типизации языка для контроля параметров функции, защиты данных от незаконного доступа,

определения новых типов и операций не влечет дополнительных расходов памяти и увеличения времени выполнения программы.

В проекте C++ особое внимание уделяется структурированию программы. Это вызвано увеличением размеров программ со времени появления C. Небольшую программу (скажем, не более 1000 строк) можно заставить из упрямства работать, нарушая все правила хорошего стиля программирования. Однако, действуя так, человек уже не сможет справиться с большой программой. Если у вашей программы в 10 000 строк плохая структура, то вы обнаружите, что новые ошибки появляются в ней так же быстро, как удаляются старые. C++ создавался с целью, чтобы большую программу можно было структурировать таким образом, чтобы одному человеку не пришлось работать с текстом в 25000 строк. В настоящее время можно считать, что эта цель полностью достигнута.

Существуют, конечно, программы еще большего размера. Однако те из них, которые действительно используются, обычно можно разбить на несколько практически независимых частей, каждая из которых имеет значительно меньший упомянутого размер. Естественно, трудность написания и сопровождения программы определяется не только числом строк текста, но и сложностью предметной области. Так что приведенные здесь числа, которыми обосновывались наши соображения, не надо воспринимать слишком серьезно.

К сожалению, не всякую часть программы можно хорошо структурировать, сделать независимой от аппаратуры, достаточно понятной и т.д. В C++ есть средства, непосредственно и эффективно представляющие аппаратные возможности. Их использование позволяет избавиться от беспокойства о надежности и простоте понимания программы. Такие части программы можно скрывать, предоставляя надежный и простой интерфейс с ними.

Естественно, если C++ используется для большой программы, то это означает, что язык используют группы программистов. Полезную роль здесь сыграют свойственные языку модульность, гибкость и строго типизированные интерфейсы. В C++ есть такой же хороший набор средств для создания больших программ, как во многих языках. Но когда программа становится еще больше, проблемы по ее созданию и

сопровождению перемещаются из области языка в более глобальную область программных средств и управления проектом. Этим вопросам посвящены лекции 11 и лекции 12.

В этой книге основное внимание уделяется методам создания универсальных средств, полезных типов, библиотек и т.д. Эти методы можно успешно применять как для маленьких, так и для больших программ. Более того, поскольку все нетривиальные программы состоят из нескольких в значительной степени независимых друг от друга частей, методы программирования отдельных частей пригодятся как системным, так и прикладным программистам.

Может возникнуть подозрение, что запись программы с использованием подробной системы типов, увеличит размер текста. Для программы на C++ это не так: программа на C++, в которой описаны типы формальных параметров функций, определены классы и т.п., обычно бывает даже короче своего эквивалента на C, где эти средства не используются. Когда в программе на C++ используются библиотеки, она также оказывается короче своего эквивалента на C, если, конечно, он существует.

Философские замечания

Язык программирования решает две взаимосвязанные задачи: позволяет программисту записать подлежащие выполнению действия и формирует понятия, которыми программист оперирует, размышляя о своей задаче. Первой цели идеально отвечает язык, который очень "близок машине". Тогда со всеми ее основными "сущностями" можно просто и эффективно работать на этом языке, причем делая это очевидным для программиста способом. Именно это имели в виду создатели C. Второй цели идеально отвечает язык, который настолько "близок к поставленной задаче", что на нем непосредственно и точно выражаются понятия, используемые в решении задачи. Именно это имелось в виду, когда первоначально определялись средства, добавляемые к C.

Связь между языком, на котором мы думаем и программируем, а также между задачами и их решениями, которые можно представить в своем

воображении, довольно близка. По этой причине ограничивать возможности языка только поиском ошибок программиста - в лучшем случае опасно. Как и в случае естественных языков, очень полезно обладать, по крайней мере, двуязычием. Язык предоставляет программисту некоторые понятия в виде языковых инструментов; если они не подходят для задачи, их просто игнорируют. Например, если существенно ограничить понятие указателя, то программист будет вынужден для создания структур, указателей и т.п. использовать вектора и операции с целыми. Хороший проект программы и отсутствие в ней ошибок нельзя гарантировать только наличием или отсутствием определенных возможностей в языке.

Типизация языка должна быть особенно полезна для нетривиальных задач. Действительно, понятие класса в C++ проявило себя как мощное концептуальное средство.

Замечания о программировании на языке C++

Предполагается, что в идеальном случае разработка программы делится на три этапа: вначале необходимо добиться ясного понимания задачи, затем определить ключевые понятия, используемые для ее решения, и, наконец, полученное решение выразить в виде программы. Однако, детали решения и точные понятия, которые будут использоваться в нем, часто проясняются только после того, как их попытаются выразить в программе. Именно в этом случае большое значение приобретает выбор языка программирования.

Во многих задачах используются понятия, которые трудно представить в программе в виде одного из основных типов или в виде функции без связанных с ней статических данных. Такое понятие может представлять в программе класс. Класс - это тип; он определяет поведение связанных с ним объектов: их создание, обработку и уничтожение. Кроме этого, класс определяет реализацию объектов в языке, но на начальных стадиях разработки программы это не является и не должно являться главной заботой. Для написания хорошей программы надо составить такой набор классов, в котором каждый класс четко представляет одно понятие. Обычно это означает, что программист должен сосредоточиться на вопросах: Как создаются

объекты данного класса? Могут ли они копироваться и (или) уничтожаться? Какие операции можно определить над этими объектами? Если на эти вопросы удовлетворительных ответов не находится, то, скорее всего, это означает, что понятие не было достаточно ясно сформулировано. Тогда, возможно, стоит еще поразмышлять над задачей и предлагаемым решением, а не немедленно приступать к программированию, надеясь в процессе него найти ответы.

Проще всего работать с понятиями, которые имеют традиционную математическую форму представления: всевозможные числа, множества, геометрические фигуры и т.д. Для таких понятий полезно было бы иметь стандартные библиотеки классов, но к моменту написания книги их еще не было. В программном мире накоплено удивительное богатство из таких библиотек, но нет ни формального, ни фактического стандарта на них. Язык C++ еще достаточно молод, и его библиотеки не развились в такой степени, как сам язык.

Понятие не существует в вакууме, вокруг него всегда группируются связанные с ним понятия. Определить в программе взаимоотношения классов, иными словами, установить точные связи между используемыми в задаче понятиями, бывает труднее, чем определить каждый из классов сам по себе. В результате не должно получиться "каши" - когда каждый класс (понятие) зависит от всех остальных. Пусть есть два класса A и B. Тогда связи между ними типа "A вызывает функцию из B", "A создает объекты B", "A имеет член типа B" обычно не вызывают каких-либо трудностей. Связи же типа "A использует данные из B", как правило, можно вообще исключить.

Одно из самых мощных интеллектуальных средств, позволяющих справиться со сложностью, - это иерархическое упорядочение, т.е. упорядочение связанных между собой понятий в древовидную структуру, в которой самое общее понятие находится в корне дерева. Часто удается организовать классы программы как множество деревьев или как направленный ацикличный граф. Это означает, что программист определяет набор базовых классов, каждый из которых имеет свое множество производных классов. Набор операций самого общего вида для базовых классов (понятий) обычно определяется с помощью виртуальных функций (§6.5). Интерпретация этих операций,

по мере надобности, может уточняться для каждого конкретного случая, т.е. для каждого производного класса.

Естественно, есть ограничения и при такой организации программы. Иногда используемые в программе понятия не удается упорядочить даже с помощью направленного ацикличного графа. Некоторые понятия оказываются по своей природе взаимосвязанными. Циклические зависимости не вызовут проблем, если множество взаимосвязанных классов настолько мало, что в нем легко разобраться. Для представления на C++ множества взаимозависимых классов можно использовать дружественные классы ($5.4.1).

Если понятия программы нельзя упорядочить в виде дерева или направленного ацикличного графа, а множество взаимозависимых понятий не поддается локализации, то, по всей видимости, вы попали в такое затруднительное положение, выйти из которого не сможет помочь ни один из языков программирования. Если вам не удалось достаточно просто сформулировать связи между основными понятиями задачи, то, скорее всего, вам не удастся ее запрограммировать.

Еще один способ выражения общности понятий в языке предоставляют шаблоны типа. Шаблонный класс задает целое семейство классов. Например, шаблонный класс список задает классы вида "список объектов T", где T может быть произвольным типом. Таким образом, шаблонный тип указывает, как получается новый тип из заданного в качестве параметра. Самые типичные шаблонные классы - это контейнеры, в частности, списки, массивы и ассоциативные массивы.

Напомним, что можно легко и просто запрограммировать многие задачи, используя только простые типы, структуры данных, обычные функции и несколько классов из стандартных библиотек. Весь аппарат построения новых типов следует привлекать только тогда, когда он действительно необходим.

Вопрос "Как написать хорошую программу на C++?" очень похож на вопрос "Как пишется хорошая английская проза?". На него есть два ответа: "Нужно знать, что вы, собственно, хотите написать" и "Практика и подражание хорошему стилю". Оба совета пригодны для C++ в той же мере, что и для английского языка, и обоим достаточно трудно следовать.

Несколько полезных советов

Ниже представлен "свод правил", который стоит учитывать при изучении C++. Когда вы станете более опытными, то на базе этих правил сможете сформулировать свои собственные, которые будут более подходить для ваших задач и более соответствовать вашему стилю программирования. Сознательно выбраны очень простые правила, и в них опущены подробности. Не следует воспринимать их слишком буквально. Хорошая программа требует и ума, и вкуса, и терпения. С первого раза обычно она не получается, поэтому экспериментируйте! Итак, свод правил.

1. Когда вы пишете программу, то создаете конкретные представления тех понятий, которые использовались в решении поставленной задачи. Структура программы должна отражать эти понятия настолько явно, насколько это возможно.
    - Если вы считаете "нечто" отдельным понятием, то сделайте его классом.
    - Если вы считаете "нечто" существующим независимо, то сделайте его объектом некоторого класса.
    - Если два класса имеют нечто существенное, и оно является для них общим, то выразите эту общность с помощью базового класса.
    - Если класс является контейнером некоторых объектов, сделайте его шаблонным классом.
2. Если определяется класс, который не реализует математических объектов вроде матриц или комплексных чисел и не является типом низкого уровня наподобие связанного списка, то:
    - Не используйте глобальные данные.
    - Не используйте глобальные функции (не члены).
    - Не используйте общие данные-члены.
    - Не используйте функции `friend`.
    - Не обращайтесь к данным-членам другого объекта непосредственно.
    - Не заводите в классе "поле типа"; используйте виртуальные функции.
    - Используйте функции-подстановки только как средство значительной оптимизации.

Замечание для программистов на C

Чем лучше программист знает C, тем труднее будет для него при программировании на C++ отойти от стиля программирования на C. Так он теряет потенциальные преимущества C++. Поэтому советуем просмотреть раздел "Отличия от C" в справочном руководстве ($.18). Здесь мы только укажем на те места, в которых использование дополнительных возможностей C++ приводит к лучшему решению, чем программирование на чистом C. Макрокоманды практически не нужны в C++: используйте `const` ($2.5) или `enum` ($2.5.1), чтобы определить поименованные константы; используйте `inline` ($4.6.2), чтобы избежать расходов ресурсов, связанных с вызовом функций; используйте шаблоны типа ($8), чтобы задать семейство функций и типов. Не описывайте переменную, пока она действительно вам не понадобится, а тогда ее можно сразу инициализировать, ведь в C++ описание может появляться в любом месте, где допустим оператор. Не используйте `malloc()`, эту операцию лучше реализует `new` ($3.2.6). Объединения нужны не столь часто, как в C, поскольку альтернативность в структурах реализуется с помощью производных классов. Старайтесь обойтись без объединений, но если они все-таки нужны, не включайте их в основные интерфейсы; используйте безымянные объединения ($2.6.2). Старайтесь не использовать указателей типа `void*`, арифметических операций с указателями, массивов в стиле C и операций приведения. Если все-таки вы используете эти конструкции, скрывайте их достаточно надежно в какую-нибудь функцию или класс. Укажем, что связывание в стиле C возможно для функции на C++, если она описана со спецификацией `extern` "C" ($4.4).

Но гораздо важнее стараться думать о программе как о множестве взаимосвязанных понятий, представляемых классами и объектами, чем представлять ее как сумму структур данных и функций, что-то делающих с этими данными.

Краткий обзор C++

В этой лекции содержится краткий обзор основных концепций и конструкций языка C++. Он служит для беглого знакомства с языком. Подробное описание возможностей языка и методов программирования на нем дается в следующих лекциях. Разговор ведется в основном вокруг абстракции данных и объектно-ориентированного программирования, но перечисляются и основные возможности процедурного программирования.

1.1 Введение

Язык программирования C++ задумывался как язык, который будет:

- лучше языка C;
- поддерживать абстракцию данных;
- поддерживать объектно-ориентированное программирование.

В этой лекции объясняется смысл этих фраз без подробного описания конструкций языка.

§ 1.2 содержит неформальное описание различий "процедурного", "модульного" и "объектно-ориентированного" программирования. Приведены конструкции языка, которые существенны для каждого из перечисленных стилей программирования. Свойственный C стиль программирования обсуждается в разделах "процедурное программирование и "модульное программирование". Язык C++ - "лучший вариант C". Он лучше поддерживает такой стиль программирования, чем сам C, причем это делается без потери какой-либо общности или эффективности по сравнению с C. В то же время язык C является подмножеством C++. Абстракция данных и объектно-ориентированное программирование рассматриваются как "поддержка абстракции данных" и "поддержка объектно- ориентированного программирования". Первая базируется на возможности определять новые типы и работать с ними, а вторая - на возможности задавать иерархию типов.

§ 1.3 содержит описание основных конструкций для процедурного и

модульного программирования. В частности, определяются функции, указатели, циклы, ввод-вывод и понятие программы как совокупности раздельно транслируемых модулей. Подробно эти возможности описаны в лекции 2, 3 и 4.

§ 1.4 содержит описание средств, предназначенных для эффективной реализации абстракции данных. В частности, определяются классы, простейший механизм контроля доступа, конструкторы и деструкторы, перегрузка операций, преобразования пользовательских типов, обработка особых ситуаций и шаблоны типов. Подробно эти возможности описаны в лекциях 5, 7, 8 и 9.

§ 1.5 содержит описание средств поддержки объектно-ориентированного программирования. В частности, определяются производные классы и виртуальные функции, обсуждаются некоторые вопросы реализации. Все это подробно изложено в лекции 6.

§ 1.6 содержит описание определенных ограничений на пути совершенствования как языков программирования общего назначения вообще, так и C++ в частности. Эти ограничения связаны с эффективностью, с противоречащими друг другу требованиями разных областей приложения, проблемами обучения и необходимостью трансляции и выполнения программ в старых системах.

Если какой-то раздел окажется для вас непонятным, настоятельно советуем прочитать соответствующие лекции, а затем, ознакомившись с подробным описанием основных конструкций языка, вернуться к этой лекции. Она нужна для того, чтобы можно было составить общее представление о языке. В ней недостаточно сведений, чтобы немедленно начать программировать.

1.2 Парадигмы программирования

Объектно-ориентированное программирование - это метод программирования, способ написания "хороших" программ для множества задач. Если этот термин имеет какой-то смысл, то он должен подразумевать такой язык программирования, который предоставляет хорошие возможности для объектно-ориентированного стиля

программирования.

Здесь следует указать на важные различия. Говорят, что язык поддерживает некоторый стиль программирования, если в нем есть такие возможности, которые делают программирование в этом стиле удобным (достаточно простым, надежным и эффективным). Язык не поддерживает некоторый стиль программирования, если требуются большие усилия или даже искусство, чтобы написать программу в этом стиле. Однако это не означает, что язык запрещает писать программы в этом стиле. Действительно, можно писать структурные программы на Фортране и объектно-ориентированные программы на С, но это будет пустой тратой сил, поскольку данные языки не поддерживают указанных стилей программирования.

Поддержка языком определенной парадигмы (стиля) программирования явно проявляется в конкретных языковых конструкциях, рассчитанных на нее. Но она может проявляться в более тонкой, скрытой форме, когда отклонение от парадигмы диагностируется на стадии трансляции или выполнения программы. Самый очевидный пример - это контроль типов. Кроме того, языковая поддержка парадигмы может дополняться проверкой на однозначность и динамическим контролем. Поддержка может предоставляться и помимо самого языка, например, стандартными библиотеками или средой программирования.

Нельзя сказать, что один язык лучше другого только потому, что в нем есть возможности, которые в другом отсутствуют. Часто бывает как раз наоборот. Здесь более важно не то, какими возможностями обладает язык, а то, насколько имеющиеся в нем возможности поддерживают избранный стиль программирования для определенного круга задач. Поэтому можно сформулировать следующие требования к языку:

1. Все конструкции языка должны естественно и элегантно определяться в нем.
2. Для решения определенной задачи должна быть возможность использовать сочетания конструкций, чтобы избежать необходимости вводить для этой цели новую конструкцию.
3. Должно быть минимальное число неочевидных конструкций специального назначения.
4. Конструкция должна допускать такую реализацию, чтобы в

неиспользующей ее программе не возникло дополнительных расходов.

5. Пользователю достаточно знать только то множество конструкций, которое непосредственно используется в его программе.

Первое требование апеллирует к логике и эстетическому вкусу. Два следующих выражают принцип минимальности. Два последних можно иначе сформулировать так: "то, чего вы не знаете, не сможет нанести вам вреда".

С учетом ограничений, указанных в этих правилах, язык C++ проектировался для поддержки абстракции данных и объектно-ориентированного программирования в добавление к традиционному стилю C. Впрочем, это не значит, что язык требует какого-то одного стиля программирования от всех пользователей.

Теперь перейдем к конкретным стилям программирования и посмотрим каковы основные конструкции языка, их поддерживающие. Мы не собираемся давать полное описание этих конструкций.

1.2.1 Процедурное программирование

Первоначальной (и, возможно, наиболее используемой) парадигмой программирования было:

Определите, какие процедуры вам нужны; используйте лучшие из известных вам алгоритмов!

Ударение делалось на обработку данных с помощью алгоритма, производящего нужные вычисления. Для поддержки этой парадигмы языки предоставляли механизм передачи параметров и получения результатов функций. Литература, отражающая такой подход, заполнена рассуждениями о способах передачи параметров, о том, как различать параметры разных типов, о различных видах функций (процедуры, подпрограммы, макрокоманды, ...) и т.д. Первым процедурным языком был Фортран, а Алгол60, Алгол68, Паскаль и C продолжили это направление.

Типичным примером хорошего стиля в таком понимании может служить функция извлечения квадратного корня. Для заданного параметра она выдает результат, который получается с помощью понятных математических операций:

```
double sqrt ( double arg )
{
  // программа для вычисления квадратного корня
}

void some_function ()
{
    double root = sqrt ( 2 );
   // ..
}
```

Двойная наклонная черта // начинает комментарий, который продолжается до конца строки.

При такой организации программы, функции вносят определенный порядок в хаос различных алгоритмов.

1.2.2 Модульное программирование

Со временем при проектировании программ акцент сместился с организации процедур на организацию структур данных. Помимо всего прочего это вызвано и ростом размеров программ. Модулем обычно называют совокупность связанных процедур и тех данных, которыми они управляют. Парадигма программирования приобрела вид:

Определите, какие модули нужны; поделите программу так, чтобы данные были скрыты в этих модулях

Эта парадигма известна также как "принцип сокрытия данных". Если в языке нет возможности сгруппировать связанные процедуры вместе с данными, то он плохо поддерживает модульный стиль программирования. Теперь метод написания "хороших" процедур применяется для отдельных процедур модуля. Типичный пример модуля

- определение стека. Здесь необходимо решить такие задачи:

1. Предоставить пользователю интерфейс для стека (например, функции `push ()` и `pop ()`).
2. Гарантировать, что представление стека (например, в виде массива элементов) будет доступно лишь через интерфейс пользователя.
3. Обеспечивать инициализацию стека перед первым его использованием.

Язык Модула-2 прямо поддерживает эту парадигму, тогда как C только допускает такой стиль. Ниже представлен на C возможный внешний интерфейс модуля, реализующего стек:

```
// описание интерфейса для  модуля,
// реализующего стек символов:

void push ( char );
char pop ();
const int stack_size = 100;
```

Допустим, что описание интерфейса находится в файле stack.h, тогда реализацию стека можно определить следующим образом:

```
#include "stack.h"  // используем интерфейс стека

static char v [ stack_size ]; // ``static" означает локальный
// в данном файле/модуле
static char * p = v; // стек вначале пуст

void push ( char c )
{
//проверить на переполнение и поместить в стек
}

char pop ()
{
//проверить, не пуст ли стек, и считать из него
}
```

Вполне возможно, что реализация стека может измениться, например, если использовать для хранения связанный список. Пользователь в любом случае не имеет непосредственного доступа к реализации: `v` и `p` - статические переменные, т.е. переменные локальные в том модуле (файле), в котором они описаны. Использовать стек можно так:

```
#include "stack.h"       // используем интерфейс стека

void some_function ()
{
    push ( 'c' );
    char c = pop ();
    if ( c != 'c' )  error ( "невозможно" );
}
```

Поскольку данные есть единственная вещь, которую хотят скрывать, понятие упрятывания данных тривиально расширяется до понятия упрятывания информации, т.е. имен переменных, констант, функций и типов, которые тоже могут быть локальными в модуле. Хотя C++ и не предназначался специально для поддержки модульного программирования, классы поддерживают концепцию модульности (§ 5.4.3 и § 5.4.4). Помимо этого C++, естественно, имеет уже продемонстрированные возможности модульности, которые есть в C, т.е. представление модуля как отдельной единицы трансляции.

1.2.3 Абстракция данных

Модульное программирование предполагает группировку всех данных одного типа вокруг одного модуля, управляющего этим типом. Если потребуются стеки двух разных видов, можно определить управляющий ими модуль с таким интерфейсом:

```
class stack_id { /* ... */ };        // stack_id только тип
      // никакой информации о стеках
      // здесь не содержится

 stack_id create_stack ( int size );  // создать стек и возвратить
```

```
    // его идентификатор

void push ( stack_id, char );
char pop ( stack_id );

destroy_stack ( stack_id ); // уничтожение стека
```

Конечно такое решение намного лучше, чем хаос, свойственный традиционным, неструктурированным решениям, но моделируемые таким способом типы совершенно очевидно отличаются от "настоящих", встроенных. Каждый управляющий типом модуль должен определять свой собственный алгоритм создания "переменных" этого типа. Не существует универсальных правил присваивания идентификаторов, обозначающих объекты такого типа. У "переменных" таких типов не существует имен, которые были бы известны транслятору или другим системным программам, и эти "переменные" не подчиняются обычным правилам областей видимости и передачи параметров.

Тип, реализуемый управляющим им модулем, по многим важным аспектам существенно отличается от встроенных типов. Такие типы не получают той поддержки со стороны транслятора (разного вида контроль), которая обеспечивается для встроенных типов. Проблема здесь в том, что программа формулируется в терминах небольших (одно-два слова) дескрипторов объектов, а не в терминах самих объектов (`stack_id` может служить примером такого дескриптора). Это означает, что транслятор не сможет отловить глупые, очевидные ошибки, вроде тех, что допущены в приведенной ниже функции:

```
void f()
{
  stack_id s1;
  stack_id s2;

  s1 = create_stack ( 200 );
  // ошибка: забыли создать s2

  push ( s1,'a' );
  char c1 = pop ( s1 );
```

```
    destroy_stack ( s2 ); // неприятная ошибка

    // ошибка: забыли уничтожить s1

    s1 = s2;  // это присваивание является по сути
    // присваиванием указателей,
    // но здесь s2 используется после уничтожения
}
```

Иными словами, концепция модульности, поддерживающая парадигму упрятывания данных, не запрещает такой стиль программирования, но и не способствует ему.

В языках Ада, Clu, C++ и подобных им эта трудность преодолевается благодаря тому, что пользователю разрешается определять свои типы, которые трактуются в языке практически так же, как встроенные. Такие типы обычно называют абстрактными типами данных, хотя лучше, пожалуй, их называть просто пользовательскими. Более строгим определением абстрактных типов данных было бы их математическое определение. Если бы удалось его дать, то, что мы называем в программировании типами, было бы конкретным представлением действительно абстрактных сущностей. Как определить "более абстрактные" типы, показано в § 4.6. Парадигму же программирования можно выразить теперь так:

Определите, какие типы вам нужны; предоставьте полный набор операций для каждого типа.

Если нет необходимости в разных объектах одного типа, то стиль программирования, суть которого сводится к упрятыванию данных, и следование которому обеспечивается с помощью концепции модульности, вполне адекватен этой парадигме.

Арифметические типы, подобные типам рациональных и комплексных чисел, являются типичными примерами пользовательских типов:

```
class complex
{
```

```
    double re, im;
public:
    complex(double r, double i) { re=r; im=i; }
    complex(double r)  // преобразование double->complex
        { re=r; im=0; }
    friend complex operator+(complex, complex);
    friend complex operator-(complex, complex); // вычитание
    friend complex operator-(complex) // унарный минус
    friend complex operator*(complex, complex);
    friend complex operator/(complex, complex);
    // ...
};
```

Описание класса (т.е. определяемого пользователем типа) complex задает представление комплексного числа и набор операций с комплексными числами. Представление является частным (private): `re` и `im` доступны только для функций, указанных в описании класса `complex`. Подобные функции могут быть определены так:

```
complex operator + ( complex a1, complex a2 )
{
  return complex ( a1.re + a2.re, a1.im + a2.im );
}
```

и использоваться следующим образом:

```
void f ()
{
  complex a = 2.3;
  complex b = 1 / a;
  complex c = a + b * complex ( 1, 2.3 );
   // ...
  c = - ( a / b ) + 2;
}
```

Большинство модулей (хотя и не все) лучше определять как пользовательские типы.

1.2.4 Пределы абстракции данных

Абстрактный тип данных определяется как некий "черный ящик". После своего определения он по сути никак не взаимодействует с программой. Его никак нельзя приспособить для новых целей, не меняя определения. В этом смысле это негибкое решение. Пусть, например, нужно определить для графической системы тип `shape` (фигура). Пока считаем, что в системе могут быть такие фигуры: окружность (circle), треугольник (triangle) и квадрат (square). Пусть уже есть определения точки и цвета:

```
class point { /* ... */ };
class color { /* ... */ };
```

Тип shape можно определить следующим образом:

```
enum kind { circle, triangle, square };

class shape
{
  point center;
  color col;
  kind k;
  // представление фигуры
public:
  point where () { return center; }
  void move ( point to ) { center = to; draw (); }
  void draw ();
  void rotate ( int );
  // еще некоторые операции
};
```

"Поле типа" `k` необходимо для того, чтобы такие операции, как `draw ()` и `rotate ()`, могли определять, с какой фигурой они имеют дело (в языках вроде Паскаля можно использовать для этого запись с вариантами, в которой `k` является полем-дескриминантом). Функцию `draw ()` можно определить так:

```
void shape :: draw ()
{
```

```
    switch ( k )
    {
    case circle:
        // рисование окружности
        break;
    case triangle:
        // рисование треугольника
        break;
    case square:
        // рисование квадрата
        break;
    }
}
```

Это не функция, а кошмар. В ней нужно учесть все возможные фигуры, какие только есть. Поэтому она дополняется новыми операторами, как только в системе появляется новая фигура. Плохо то, что после определения новой фигуры нужно проверить и, возможно, изменить все старые операции класса. Поэтому, если вам недоступен исходный текст каждой операции класса, ввести новую фигуру в систему просто невозможно. Появление любой новой фигуры приводит к манипуляциям с текстом каждой существенной операции класса. Требуется достаточно высокая квалификация, чтобы справиться с этой задачей, но все равно могут появиться ошибки в уже отлаженных частях программы, работающих со старыми фигурами. Возможность выбора представления для конкретной фигуры сильно сужается, если требовать, чтобы все ее представления укладывались в уже заданный формат, специфицированный общим определением фигуры (т.е. определением типа `shape`).

1.2.5 Объектно-ориентированное программирование

Проблема состоит в том, что мы не различаем общие свойства фигур (например, фигура имеет цвет, ее можно нарисовать и т.д.) и свойства конкретной фигуры (например, окружность - это такая фигура, которая имеет радиус, она изображается с помощью функции, рисующей дуги и т.д.). Суть объектно-ориентированного программирования в том, что оно позволяет выражать эти различия и использует их. Язык, который

имеет конструкции для выражения и использования подобных различий, поддерживает объектно-ориентированное программирование. Все другие языки не поддерживают его. Здесь основную роль играет механизм наследования, заимствованный из языка Симула. Вначале определим класс, задающий общие свойства всех фигур:

```
class shape
{
  point center;
  color col;
  // ...
  public:
  point where () { return center; }
  void move ( point to ) { center = to; draw(); }
  virtual void draw ();
  virtual void rotate ( int );
  // ...
};
```

Те функции, для которых можно определить заявленный интерфейс, но реализация которых (т.е. тело с операторной частью) возможна только для конкретных фигур, отмечены служебным словом `virtual` (виртуальные). В Симуле и C++ виртуальность функции означает: "функция может быть определена позднее в классе, производном от данного". С учетом такого определения класса можно написать общие функции, работающие с фигурами:

```
void rotate_all ( shape v [], int size, int angle )
  // повернуть все элементы массива "v" размера "size"
  // на угол равный "angle"
  {
    int i = 0;
    while ( i<size )
    {
      v [ i ] . rotate ( angle );
      i = i + 1;
    }
  }
```

Для определения конкретной фигуры следует указать, прежде всего, что это - именно фигура и задать ее особые свойства (включая и виртуальные функции):

```
class circle : public shape
  {
     int radius;
  public:
     void draw () { /* ... */ };
     void rotate ( int ) {}  // да, пока пустая функция
  };
```

В языке C++ класс `circle` называется производным по отношению к классу shape, а класс `shape` называется базовым для класса circle. Возможна другая терминология, использующая названия "подкласс" и "суперкласс" для классов `circle` и `shape` соответственно. Теперь парадигма программирования формулируется так:

Определите, какой класс вам необходим; предоставьте полный набор операций для каждого класса; общность классов выразите явно с помощью наследования.

Если общность между классами отсутствует, вполне достаточно абстракции данных. Насколько применимо объектно-ориентированное программирование для данной области приложения, определяется степенью общности между разными типами, которая позволяет использовать наследование и виртуальные функции. В некоторых областях, таких, например, как интерактивная графика, есть широкий простор для объектно-ориентированного программирования. В других областях, в которых используются традиционные арифметические типы и вычисления над ними, трудно найти применение для более развитых стилей программирования, чем абстракция данных. Здесь средства, поддерживающие объектно-ориентированное программирование, очевидно, избыточны.

Нахождение общности среди отдельных типов системы представляет собой нетривиальный процесс. Степень такой общности зависит от способа проектирования системы. В процессе проектирования выявление общности классов должно быть постоянной целью. Она

достигается двумя способами: либо проектированием специальных классов, используемых как "кирпичи" при построении других, либо поиском похожих классов для выделения их общей части в один базовый класс.

Итак, мы указали, какую минимальную поддержку должен обеспечивать язык программирования для процедурного программирования, для упрятывания данных, абстракции данных и объектно-ориентированного программирования. Теперь несколько подробнее опишем средства языка, хотя и не самые существенные, но позволяющие более эффективно реализовать абстракцию данных и объектно-ориентированное программирование.

1.3 "Улучшенный С"

Минимальная поддержка процедурного программирования включает функции, арифметические операции, выбирающие операторы и циклы. Помимо этого должны быть предоставлены операции ввода-вывода. Базовые языковые средства C++ унаследовал от C (включая указатели), а операции ввода-вывода предоставляются библиотекой. Самая зачаточная концепция модульности реализуется с помощью механизма раздельной трансляции.

1.3.1 Программа и стандартный вывод

Самая маленькая программа на C++ выглядит так:

```
main () { }
```

В этой программе определяется функция, называемая `main`, которая не имеет параметров и ничего не делает. Фигурные скобки { и } используются в C++ для группирования операторов. В данном случае они обозначают начало и конец тела (пустого) функции main. В каждой программе на C++ должна быть своя функция `main()`, и программа начинается с выполнения этой функции.

Обычно программа выдает какие-то результаты. Вот программа, которая выдает приветствие Hello, World! (Всем привет!):

```
#include <iostream.h>

        int main ()
        {
cout << "Hello, World!\n";
        }
```

Строка #include <iostream.h> сообщает транслятору, что надо включить в программу описания, необходимые для работы стандартных потоков ввода- вывода, которые находятся в iostream.h. Без этих описаний выражение

```
cout << "Hello, World!\n"
```

не имело бы смысла. Операция << ("выдать") записывает свой второй параметр в первый параметр. В данном случае строка "Hello, World!\n" записывается в стандартный выходной поток cout. Строка - это последовательность символов, заключенная в двойные кавычки. Два символа: обратной дробной черты \ и непосредственно следующий за ним - обозначают некоторый специальный символ. В данном случае `\n` является символом конца строки (или перевода строки), поэтому он выдается после символов `Hello, world`!

Целое значение, возвращаемое функцией `main()`, если только оно есть, считается возвращаемым системе значением программы. Если ничего не возвращается, система получит какое-то "мусорное" значение.

1.3.2 Переменные и арифметические операции

Каждое имя и каждое выражение обязаны иметь тип. Именно тип определяет операции, которые могут выполняться над ними. Например, в описании

```
int inch;
```

говорится, что `inch` имеет тип `int`, т.е. `inch` является целой переменной.

Описание - это оператор, который вводит имя в программу. В описании указывается тип имени. Тип, в свою очередь, определяет как правильно использовать имя или выражение.

Основные типы, наиболее приближенные к "аппаратной реальности" машины, таковы:

char
short
int
long

Они представляют целые числа. Следующие типы:

float
double
long double

представляют числа с плавающей точкой. Переменная типа `char` имеет размер, нужный для хранения одного символа на данной машине (обычно это один байт). Переменная `int` имеет размер, необходимый для целой арифметики на данной машине (обычно это одно слово).

Следующие арифметические операции можно использовать над любым сочетанием перечисленных типов:

\+ (плюс, унарный и бинарный)
\- (минус, унарный и бинарный)
* (умножение)
/ (деление)
% (остаток от деления)

То же верно для операций отношения:

== (равно)
!= (не равно)
< (меньше чем)
<= (меньше или равно)
>= (больше или равно)

Для операций присваивания и арифметических операций в C++ выполняются все осмысленные преобразования основных типов, чтобы их можно было неограниченно использовать любые их сочетания:

```
double d;
int i;
short s;
// ...
d = d + i;
i = s * i;
```

Символ = обозначает обычное присваивание.

1.3.3 Указатели и массивы

Массив можно описать так:

```
char v [ 10 ]; // массив из 10 символов
```

Описание указателя имеет такой вид:

```
char * p; // указатель на символ
```

Здесь `[]` означает "массив из", а символ `*` означает "указатель на". Значение нижней границы индекса для всех массивов равно нулю, поэтому `v` имеет 10 элементов: `v [ 0 ] ... v [ 9 ]`. Переменная типа указатель может содержать адрес объекта соответствующего типа:

```
p = & v [ 3 ]; // p указывает на 4-й элемент массива v
```

Унарная операция & означает взятие адреса.

1.3.4 Условные операторы и циклы

В C++ есть традиционный набор выбирающих операторов и циклов.

Ниже приводятся примеры операторов `if`, `switch` и `while`.

В следующем примере показано преобразование дюйма в сантиметр и обратно. Предполагается, что во входном потоке значение в сантиметрах завершается символом `c`, а значение в дюймах - символом `i`:

```
#include <iostream.h>

int main ()
{
  const float fac = 2.54;
  float x, in, cm;
  char ch = 0;

  cout << "enter length: ";

  cin >> x;   // ввод числа с плавающей точкой
  cin >> ch;   // ввод завершающего символа

  if ( ch == 'i' )
  {      // дюйм
     in = x;
     cm = x * fac;
  }
  else if ( ch == 'c' )
  { // сантиметры
     in = x / fac;
     cm = x;
  }
  else
     in = cm = 0;

  cout << in << " in = " << cm << " cm\n";
}
```

Операция >> ("ввести из") используется как оператор ввода; `cin` является стандартным входным потоком. Тип операнда, расположенного справа от операции >>, определяет, какое значение

вводится; оно записывается в этот операнд.

Оператор `switch` (переключатель) сравнивает значение с набором констант. Проверку в предыдущем примере можно записать так:

```
switch ( ch )
{
case 'i':
    in = x;
    cm = x * fac;
    break;
case 'c':
    in = x / fac;
    cm = x;
    break;
        default:
   in = cm = 0;
   break;
       }
```

Операторы `break` используются для выхода из переключателя. Все константы вариантов должны быть различны. Если сравниваемое значение не совпадает ни с одной из них, выполняется оператор с меткой `default`. Вариант `default` может и отсутствовать.

Приведем запись, задающую копирование 10 элементов одного массива в другой:

```
int v1 [ 10 ];
int v2 [ 10 ];
// ...
for ( int i=0; i<10; i++ ) v1 [ i ] = v2 [ i ];
```

Словами это можно выразить так: "Начать с i равного нулю, и пока i меньше 10, копировать i-тый элемент и увеличивать i." Инкремент (++) переменной целого типа просто сводится к увеличению на 1.

1.3.5 Функции

Функция - это поименованная часть программы, которая может вызываться из других частей программы столько раз, сколько необходимо. Приведем программу, выдающую степени числа два:

```
extern float pow ( float, int );
  // pow () определена в другом месте

int main ()
{
  for ( int i=0; i<10; i++ ) cout << pow ( 2, i ) << '\n';
}
```

Первая строка является описанием функции. Она задает pow как функцию с параметрами типа `float` и `int`, возвращающую значение типа `float`. Описание функции необходимо для ее вызова, ее определение находится в другом месте.

При вызове функции тип каждого фактического параметра сверяется с типом, указанным в описании функции, точно так же, как если бы инициализировалась переменная описанного типа. Это гарантирует надлежащую проверку и преобразования типов. Например, вызов функции pow(12.3,"abcd") транслятор сочтет ошибочным, поскольку "abcd" является строкой, а не параметром типа `int`. В вызове `pow` (2,i) транслятор преобразует целую константу (целое 2) в число с плавающей точкой (`float`), как того требует функция. Функция pow может быть определена следующим образом:

```
float pow ( float x, int n )
    {
if ( n < 0 )
  error ( "ошибка: для pow () задан отрицательный показатель");
switch ( n )
{
case 0:  return 1;
case 1:  return x;
default: return x * pow ( x, n-1 );
}
    }
```

Первая часть определения функции задает ее имя, тип возвращаемого значения (если оно есть), а также типы и имена формальных параметров (если они существуют). Значение возвращается из функции с помощью оператора `return`.

Разные функции обычно имеют разные имена, но функциям, выполняющим сходные операции над объектами разных типов, лучше дать одно имя. Если типы параметров таких функций различны, то транслятор всегда может разобраться, какую функцию нужно вызывать. Например, можно иметь две функции возведения в степень: одну - для целых чисел, а другую - для чисел с плавающей точкой:

```
int pow ( int, int );
double pow ( double, double );
//...
x = pow ( 2,10 );    // вызов pow ( int, int )
y = pow ( 2.0, 10.0 );// вызов pow ( double, double )
```

Такое многократное использование имени называется перегрузкой имени функции или просто перегрузкой.

Параметры функции могут передаваться либо "по значению", либо "по ссылке". Рассмотрим определение функции, которая осуществляет взаимообмен значений двух целых переменных. Если используется стандартный способ передачи параметров по значению, то придется передавать указатели:

```
void swap ( int * p, int * q )
        {
 int t = * p;
 * p = * q;
 * q = t;
        }
```

Унарная операция * называется косвенностью (или операцией разыменования), она выбирает значение объекта, на который настроен указатель. Функцию можно вызывать следующим образом:

```
void f ( int i, int j )
        {
```

```
swap ( & i, & j );
        }
```

Если использовать передачу параметра по ссылке, можно обойтись без явных операций с указателем:

```
void swap (int & r1, int & r2 )
        {
int t = r1;
r1 = r2;
r2 = t;
        }

        void g ( int i, int j )
        {
swap ( i, j );
        }
```

Для любого типа `T` запись T& означает "ссылка на T". Ссылка служит синонимом той переменной, которой она инициализировалась. Отметим, что перегрузка допускает сосуществование двух функций `swap` в одной программе.

1.3.6 Модули

Программа C++ почти всегда состоит из нескольких раздельно транслируемых "модулей". Каждый "модуль" обычно называется исходным файлом, но иногда - единицей трансляции. Он состоит из последовательности описаний типов, функций, переменных и констант. Описание `extern` позволяет из одного исходного файла ссылаться на функцию или объект, определенные в другом исходном файле. Например:

```
extern "C" double sqrt ( double );
extern ostream cout;
```

Самый распространенный способ обеспечить согласованность описаний внешних переменных во всех исходных файлах - поместить

такие описания в специальные файлы, называемые заголовочными. Заголовочные файлы можно включать во все исходные файлы, в которых требуются описания внешних переменных. Например, описание функции `sqrt` хранится в заголовочном файле стандартных математических функций с именем `math.h`, поэтому, если нужно извлечь квадратный корень из 4, можно написать:

```
#include <math.h>
//...
x = sqrt ( 4 );
```

Поскольку стандартные заголовочные файлы могут включаться во многие исходные файлы, в них нет описаний, дублирование которых могло бы вызвать ошибки. Так, тело функции присутствует в таких файлах, если только это функция-подстановка, а инициализаторы указаны только для констант. Не считая таких случаев, заголовочный файл обычно служит хранилищем для типов, он предоставляет интерфейс между раздельно транслируемыми частями программы.

В команде включения заключенное в угловые скобки имя файла (в нашем примере - <math.h>) ссылается на файл, находящийся в стандартном каталоге включаемых файлов. Часто это - каталог `/usr/include/CC`. Файлы, находящиеся в других каталогах, обозначаются своими путевыми именами, взятыми в кавычки. Поэтому в следующих командах:

```
#include "math1.h"
#include "/usr/bs/math2.h"
```

включаются файл math1.h из текущего каталога пользователя и файл math2.h из каталога `/usr/bs`.

Приведем небольшой законченный пример, в котором строка определяется в одном файле, а печатается в другом. В файле header.h определяются нужные типы:

```
// header.h

extern char * prog_name;
extern void f ();
```

Файл main.c является основной программой:

```
// main.c

#include "header.h"
char * prog_name = "примитивный, но законченный пример";
int main ()
{
f ();
}
```

а строка печатается функцией из файла f.c:

```
// f.c

#include <stream.h>
#include "header.h"
void f ()
{
cout << prog_name << '\n';
}
```

При запуске транслятора C++ и передаче ему необходимых файлов-параметров в различных реализациях могут использоваться разные расширения имен для программ на C++. На машине автора трансляция и запуск программы выглядит так:

```
$ CC main.c f.c -o silly
$ silly

примитивный, но законченный пример

$
```

Кроме раздельной трансляции концепцию модульности в C++ поддерживают классы.

1.4 Поддержка абстракции данных

Поддержка программирования с абстракцией данных в основном

сводится к возможности определить набор операций (функции и операции) над типом. Все обращения к объектам этого типа ограничиваются операциями из заданного набора. Однако, имея такие возможности, программист скоро обнаруживает, что для удобства определения и использования новых типов нужны еще некоторые расширения языка. Хорошим примером такого расширения является перегрузка операций.

1.4.1 Инициализация и удаление

Когда представление типа скрыто, необходимо дать пользователю средства для инициализации переменных этого типа. Простейшее решение - до использования переменной вызывать некоторую функцию для ее инициализации.

Например:

```
class vector
{
// ...
public:
   void init ( int size );  // вызов init () перед первым
       // использованием объекта vector
   // ...
};

void f ()
{
  vector v;
  // пока v нельзя использовать
  v.init ( 10 );
  // теперь можно
}
```

Но это некрасивое и чреватое ошибками решение. Будет лучше, если создатель типа определит для инициализации переменных некоторую специальную функцию. Если такая функция есть, то две независимые операции размещения и инициализации переменной совмещаются в

одной (иногда ее называют инсталляцией или просто построением). Функция инициализации называется конструктором. Конструктор выделяется среди всех прочих функций данного класса тем, что имеет такое же имя, как и сам класс. Если объекты некоторого типа строятся нетривиально, то нужна еще одна дополнительная операция для удаления их после последнего использования. Функция удаления в C++ называется деструктором. Деструктор имеет то же имя, что и его класс, но перед ним стоит символ ~ (в C++ этот символ используется для операции дополнения). Приведем пример:

```
class vector
 {
   int  sz;  // число элементов
   int * v;   // указатель на целые
 public:
   vector ( int );    // конструктор
   ~vector ();      // деструктор
   int& operator [] ( int index ); // операция индексации
 };
```

Конструктор класса `vector` можно использовать для контроля над ошибками и выделения памяти:

```
vector::vector ( int s )
 {
  if ( s <= 0 )
     error ( "недопустимый размер вектора" );
  sz = s;
  v = new int [ s ];  // разместить массив из s целых
 }
```

Деструктор класса `vector` освобождает использовавшуюся память:

```
vector::~vector ()
 {
  delete [] v;      // освободить массив, на который
// настроен указатель v
 }
```

От реализации C++ не требуется освобождения выделенной с помощью

new памяти, если на нее больше не ссылается ни один указатель (иными словами, не требуется автоматическая "сборка мусора"). Взамен этого можно без вмешательства пользователя определить в классе собственные функции управления памятью. Это типичный способ применения конструкторов и деструкторов, хотя есть много не связанных с управлением памятью применений этих функций.

1.4.2 Присваивание и инициализация

Для многих типов задача управления ими сводится к построению и уничтожению связанных с ними объектов, но есть типы, для которых этого мало. Иногда необходимо управлять всеми операциями копирования. Вернемся к классу `vector`:

```
void f()
{
vector v1 ( 100 );
vector v2 = v1;   // построение нового вектора v2,
      // инициализируемого v1
v1 = v2;// v2 присваивается v1
 // ...
}
```

Должна быть возможность определить интерпретацию операций инициализации `v2` и присваивания `v1`. Например, в описании:

```
class vector
{
  int * v;
  int sz;
public:
  // ...
  void operator = ( const vector & );  // присваивание
  vector ( const vector & );// инициализация
};
```

указывается, что присваивание и инициализация объектов типа `vector` должны выполняться с помощью определенных

пользователем операций.

Присваивание можно определить так:

```
void vector::operator = ( const vector & a )
 // контроль размера и копирование элементов
{
if ( sz != a.sz )
  error ( "недопустимый размер вектора для =" );
for ( int i = 0; i < sz; i++ ) v [ i ] = a.v [ i ];
}
```

Поскольку эта операция использует для присваивания "старое значение" вектора, операция инициализации должна задаваться другой функцией, например, такой:

```
vector::vector ( const vector & a )
 // инициализация вектора значением другого вектора
{
sz = a.sz;    // размер тот же
v = new int [ sz ];     // выделить память для массива
for ( int i = 0; i < sz; i++ )    //копирование элементов
   v [ i ] = a.v [ i ];
}
```

В языке C++ конструктор вида `T` (const T&) называется конструктором копирования для типа `T`. Любую инициализацию объектов типа `T` он выполняет с помощью значения некоторого другого объекта типа `T`. Помимо явной инициализации конструкторы вида `T` (const T&) используются для передачи параметров по значению и получения возвращаемого функцией значения.

1.4.3 Шаблоны типа

Зачем программисту может понадобиться определить такой тип, как вектор целых чисел? Как правило, ему нужен вектор из элементов, тип которых неизвестен создателю класса `Vector`. Следовательно, надо суметь определить тип вектора так, чтобы тип элементов в этом

определении участвовал как параметр, обозначающий "реальные" типы элементов:

```
template < class T > class Vector
 { // вектор элементов типа T
  T * v;
  int sz;
public:
  Vector ( int s )
  {
     if ( s <= 0 )
error ( "недопустимый для Vector размер" );
     v = new T [ sz = s ];
// выделить память для массива s типа T
  }
  T & operator [] ( int i );
  int size () { return sz; }
  // ...
};
```

Таково определение шаблона типа. Он задает способ получения семейства сходных классов. В нашем примере шаблон типа `Vector` показывает, как можно получить класс вектор для заданного типа его элементов. Это описание отличается от обычного описания класса наличием начальной конструкции `template<class T>`, которая и показывает, что описывается не класс, а шаблон типа с заданным параметром-типом (здесь он используется как тип элементов). Теперь можно определять и использовать вектора разных типов:

```
void f ()
 {
 Vector < int > v1 ( 100 );     // вектор из 100 целых
 Vector < complex > v2 ( 200 ); // вектор из 200
       // комплексных чисел
 v2 [ i ] = complex ( v1 [ x ], v1 [ y ] );
 // ...
 }
```

Возможности, которые реализует шаблон типа, иногда называются

параметрическими типами или генерическими объектами. Оно сходно с возможностями, имеющимися в языках Clu и Ада. Использование шаблона типа не влечет за собой каких-либо дополнительных расходов времени по сравнению с использованием класса, в котором все типы указаны непосредственно.

1.4.4 Обработка особых ситуаций

По мере роста программ, а особенно при активном использовании библиотек появляется необходимость стандартной обработки ошибок (или, в более широком смысле, "особых ситуаций"). Языки Ада, Алгол-68 и Clu поддерживают стандартный способ обработки особых ситуаций.

Снова вернемся к классу `vector`. Что нужно делать, когда операции индексации передано значение индекса, выходящее за границы массива? Создатель класса `vector` не знает, на что рассчитывает пользователь в таком случае, а пользователь не может обнаружить подобную ошибку (если бы мог, то эта ошибка вообще не возникла бы). Выход такой: создатель класса обнаруживает ошибку выхода за границу массива, но только сообщает о ней неизвестному пользователю. Пользователь сам принимает необходимые меры.

Например:

```
class vector {
 // определение типа возможных особых ситуаций
        class range { };
 // ...
};
```

Вместо вызова функции ошибки в функции `vector::operator[]()` можно перейти на ту часть программы, в которой обрабатываются особые ситуации. Это называется "запустить особую ситуацию" ("throw the exception"):

```
int & vector::operator [] ( int i )
        {
```

```
if ( i < 0 || sz <= i ) throw range ();
return v [ i ];
        }
```

В результате из стека будет выбираться информация, помещаемая туда при вызовах функций, до тех пор, пока не будет обнаружен обработчик особой ситуации с типом `range` для класса вектор (`vector::range`); он и будет выполняться.

Обработчик особых ситуаций можно определить только для специального блока:

```
void f ( int i )
{
  try
  {
  // в этом блоке обрабатываются особые ситуации
  // с помощью определенного ниже обработчика
  vector v ( i );
  // ...
  v [ i + 1 ] = 7;  // приводит к особой ситуации range
  // ...
  g ();         // может привести к особой ситуации range
    // на некоторых векторах
  }
  catch ( vector::range )
  {
     error ( "f (): vector range error" );
     return;
  }
}
```

Использование особых ситуаций делает обработку ошибок более упорядоченной и понятной.

1.4.5 Преобразования типов

Определяемые пользователем преобразования типа, например, такие,

как преобразование числа с плавающей точкой в комплексное, которое необходимо для конструктора `complex` (double), оказались очень полезными в C++. Программист может задавать эти преобразования явно, а может полагаться на транслятор, который выполняет их неявно в том случае, когда они необходимы и однозначны:

```
complex a = complex ( 1 );
complex b = 1;      // неявно: 1 -> complex ( 1 )
a = b + complex ( 2 );
a = b + 2;// неявно: 2 -> complex ( 2)
```

Преобразования типов нужны в C++ потому, что арифметические операции со смешанными типами являются нормой для языков, используемых в числовых задачах. Кроме того, большая часть пользовательских типов, используемых для "вычислений" (например, матрицы, строки, машинные адреса) допускает естественное преобразование в другие типы (или из других типов).

Преобразования типов способствуют более естественной записи программы:

```
complex a = 2;
complex b = a + 2;  // это означает: operator + ( a, complex ( 2  ))
b = 2 + a;// это означает: operator + ( complex ( 2 ), a )
```

В обоих случаях для выполнения операции "+" нужна только одна функция, а ее параметры единообразно трактуются системой типов языка. Более того, класс `complex` описывается так, что для естественного и беспрепятственного обобщения понятия числа нет необходимости что-то изменять для целых чисел.

1.4.6 Множественные реализации

Основные средства, поддерживающие объектно-ориентированное программирование, а именно: производные классы и виртуальные функции,- можно использовать и для поддержки абстракции данных, если допустить несколько реализаций одного типа. Вернемся к примеру со стеком:

```
template < class T >
 class stack
 {
    public:
    virtual void push ( T ) = 0; // чистая виртуальная функция
    virtual T pop () = 0;      // чистая виртуальная функция
 };
```

Обозначение `=0` показывает, что для виртуальной функции не требуется никакого определения, а класс `stack` является абстрактным, т.е. он может использоваться только как базовый класс. Поэтому стеки можно использовать, но не создавать:

```
class cat { /* ... */ };
stack < cat > s;    // ошибка: стек - абстрактный класс

void some_function ( stack <cat> & s, cat kitty ) // нормально
{
 s.push ( kitty );
 cat c2 = s.pop ();
 // ...
}
```

Поскольку интерфейс стека ничего не сообщает о его представлении, от пользователей стека полностью скрыты детали его реализации.

Можно предложить несколько различных реализаций стека. Например, стек может быть массивом:

```
template < class T >
 class astack : public stack < T >
 {
 // истинное представление объекта типа стек
 // в данном случае - это массив
 // ...
 public:
 astack ( int size );
 ~astack ();
```

```
void push ( T );
T pop ();
};
```

Можно реализовать стек как связанный список:

```
template < class T >
 class lstack : public stack < T >
 {
 // ...
 };
```

Теперь можно создавать и использовать стеки:

```
void g ()
{
 lstack < cat > s1 ( 100 );
 astack < cat > s2 ( 100 );

 cat Ginger;
 cat Snowball;

 some_function ( s1, Ginger );
 some_function ( s2, Snowball );
 }
```

О том, как представлять стеки разных видов, должен беспокоиться только тот, кто их создает (т.е. функция `g()`), а пользователь стека (т.е. автор функции `some_function()`) полностью огражден от деталей их реализации. Платой за подобную гибкость является то, что все операции над стеками должны быть виртуальными функциями.

1.5 Поддержка объектно-ориентированного программирования

Поддержку объектно-ориентированного программирования обеспечивают классы вместе с механизмом наследования, а также механизм вызова функций-членов в зависимости от истинного типа объекта (дело в том, что возможны случаи, когда этот тип неизвестен на

стадии трансляции). Особенно важную роль играет механизм вызова функций-членов. Не менее важны средства, поддерживающие абстракцию данных (о них мы говорили ранее). Все доводы в пользу абстракции данных и базирующихся на ней методов, которые позволяют естественно и красиво работать с типами, действуют и для языка, поддерживающего объектно-ориентированное программирование. Успех обоих методов зависит от способа построения типов, от того, насколько они просты, гибки и эффективны. Метод объектно-ориентированного программирования позволяет определять более общие и гибкие пользовательские типы по сравнению с теми, которые получаются, если использовать только абстракцию данных.

1.5.1 Механизм вызова

Основное средство поддержки объектно-ориентированного программирования - это механизм вызова функции-члена для данного объекта, когда истинный тип его на стадии трансляции неизвестен. Пусть, например, есть указатель `p`. Как происходит вызов `p->rotate(45)`? Поскольку C++ базируется на статическом контроле типов, задающее вызов выражение имеет смысл только при условии, что функция rotate() уже была описана. Далее, из обозначения `p->rotate()` мы видим, что p является указателем на объект некоторого класса, а `rotate` должна быть членом этого класса. Как и при всяком статическом контроле типов проверка корректности вызова нужна для того, чтобы убедиться (насколько это возможно на стадии трансляции), что типы в программе используются непротиворечивым образом. Тем самым гарантируется, что программа свободна от многих видов ошибок.

Итак, транслятору должно быть известно описание класса, аналогичное тем, что приводились в § 1.2.5:

```
class shape
{
 // ...
 public:
// ...
virtual void rotate ( int );
```

```
// ...
        };
```

а указатель p должен быть описан, например, так:

```
T * p;
```

где `T` - класс `shape` или производный от него класс. Тогда транслятор видит, что класс объекта, на который настроен указатель p, действительно имеет функцию `rotate()`, а функция имеет параметр типа `int`. Значит, `p->rotate(45)` корректное выражение.

Поскольку `shape::rotate()` была описана как виртуальная функция, нужно использовать механизм вызова виртуальной функции. Чтобы узнать, какую именно из функций `rotate` следует вызвать, нужно до вызова получить из объекта некоторую служебную информацию, которая была помещена туда при его создании. Как только установлено, какую функцию надо вызвать, допустим `circle::rotate`, происходит ее вызов с уже упоминавшимся контролем типа. Обычно в качестве служебной информации используется таблица адресов функций, а транслятор преобразует имя `rotate` в индекс этой таблицы. С учетом этой таблицы объект типа `shape` можно представить так:

```
center
  vtbl:
  color   &X::draw
 &Y::rotate
        ...
   ...
```

Функции из таблицы виртуальных функций `vtbl` позволяют правильно работать с объектом даже в тех случаях, когда в вызывающей функции неизвестны ни таблица `vtbl`, ни расположение данных в части объекта, обозначенной ... . Здесь как `X` и `Y` обозначены имена классов, в которые входят вызываемые функции. Для объекта `circle` оба имени `X` и `Y` есть `circle`. Вызов виртуальной функции может быть по сути столь же эффективен, как вызов обычной функции.

1.5.2 Проверка типа

Необходимость контроля типа при обращениях к виртуальным функциям может оказаться определенным ограничением для разработчиков библиотек. Например, хорошо бы предоставить пользователю класс "стек чего-угодно". Непосредственно в C++ это сделать нельзя. Однако, используя шаблоны типа и наследование, можно приблизиться к той эффективности и простоте проектирования и использования библиотек, которые свойственны языкам с динамическим контролем типов. К таким языкам относится, например, язык Smalltalk, на котором можно описать "стек чего-угодно". Рассмотрим определение стека с помощью шаблона типа:

```
template < class T > class stack
{
  T * p;
  int sz;
public:
  stack ( int );
  ~stack ();

  void push ( T );
  T & pop ();
};
```

Не ослабляя статического контроля типов, можно использовать такой стек для хранения указателей на объекты типа `plane` (самолет):

```
stack < plane * > cs ( 200 );

void f ()
{
 cs.push ( new Saab900 );   // Ошибка при трансляции :
     // требуется plane*, а передан car*
 cs.push ( new Saab37B );
     // прекрасно: Saab 37B - на самом
     // деле самолет, т.е. типа plane
 cs.pop () -> takeoff ();
```

```
  cs.pop () -> takeoff ();
}
```

Если статического контроля типов нет, приведенная выше ошибка обнаружится только при выполнении программы:

```
// пример  динамическое контроля типа
// вместо статического; это не C++
Stack s; // стек может хранить указатели на объекты
// произвольного типа
void f ()
{
  s.push ( new Saab900 );
  s.push ( new Saab37B );
  s.pop () -> takeoff ();  // прекрасно: Saab 37B - самолет
  s.pop () -> takeoff ();  // динамическая ошибка:
    // машина не может взлететь
}
```

Для способа определения, допустима ли операция над объектом, обычно требуется больше дополнительных расходов, чем для механизма вызова виртуальных функций в C++.

Рассчитывая на статический контроль типов и вызов виртуальных функций, мы приходим к иному стилю программирования, чем надеясь только на динамический контроль типов. Класс в C++ задает строго определенный интерфейс для множества объектов этого и любого производного класса, тогда как в Smalltalk класс задает только минимально необходимое число операций, и пользователь вправе применять незаданные в классе операции. Иными словами, класс в C++ содержит точное описание операций, и пользователю гарантируется, что только эти операции транслятор сочтет допустимыми.

1.5.3 Множественное наследование

Если класс `A` является базовым классом для `B`, то `B` наследует атрибуты `A`. т.е. `B` содержит `A` плюс еще что-то. С учетом этого становится очевидно, что хорошо, когда класс `B` может наследовать из двух базовых

классов A1 и A2. Это называется множественным наследованием.

Приведем некий типичный пример множественного наследования. Пусть есть два библиотечных класса displayed и task. Первый представляет задачи, информация о которых может выдаваться на экран с помощью некоторого монитора, а второй - задачи, выполняемые под управлением некоторого диспетчера. Программист может создавать собственные классы, например, такие:

```
class my_displayed_task: public displayed, public task
{
// текст пользователя
    };

    class my_task: public task {
// эта задача не изображается
// на экране, т.к. не содержит класс displayed
// текст пользователя
    };

    class my_displayed: public displayed
    {
// а это не задача
// т.к. не содержит класс task
// текст пользователя
};
```

Если наследоваться может только один класс, то пользователю доступны только два из трех приведенных классов. В результате либо получается дублирование частей программы, либо теряется гибкость, а, как правило, происходит и то, и другое. Приведенный пример проходит в C++ безо всяких дополнительных расходов времени и памяти по сравнению с программами, в которых наследуется не более одного класса. Статический контроль типов от этого тоже не страдает.

Все неоднозначности выявляются на стадии трансляции:

```
class task
{
```

```
public:
  void trace ();
  // ...
};

class displayed
{
public:
  void trace ();
  // ...
};

class my_displayed_task:public displayed, public task
{
  // в этом классе trace () не определяется
};

void g ( my_displayed_task * p )
{
  p -> trace ();  // ошибка: неоднозначность
}
```

В этом примере видны отличия C++ от объектно-ориентированных диалектов языка Лисп, в которых есть множественное наследование. В этих диалектах неоднозначность разрешается так: или считается существенным порядок описания, или считаются идентичными объекты с одним и тем же именем в разных базовых классах, или используются комбинированные способы, когда совпадение объектов для базовых классов сочетается с более сложным способом для производных классов. В C++ неоднозначность, как правило, разрешается введением еще одной функции:

```
class my_displayed_task:public displayed, public task
  {
     // ...
  public:
     void trace ()
     {
       // текст пользователя
```

```
        displayed::trace ();  // вызов trace () из displayed
        task::trace ();       // вызов trace () из task
      }
      // ...
  };

  void g ( my_displayed_task * p )
  {
      p -> trace ();  // теперь нормально
  }
```

1.5.4 Инкапсуляция

Пусть члену класса (неважно функции-члену или члену, представляющему данные) требуется защита от "несанкционированного доступа". Как разумно ограничить множество функций, которым такой член будет доступен? Очевидный ответ для языков, поддерживающих объектно-ориентированное программирование, таков: доступ имеют все операции, которые определены для этого объекта, иными словами, все функции-члены. Например:

```
class window
{
// ...
protected:
   Rectangle inside;
   // ...
};

class dumb_terminal : public window
{
   // ...
public:
   void prompt ();
   // ...
};
```

Здесь в базовом классе window член `inside` типа Rectangle

описывается как защищенный (protected), но функции-члены производных классов, например, `dumb_terminal::prompt()`, могут обратиться к нему и выяснить, с какого вида окном они работают. Для всех других функций член `window::inside` недоступен.

В таком подходе сочетается высокая степень защищенности (действительно, вряд ли вы "случайно" определите производный класс) с гибкостью, необходимой для программ, которые создают классы и используют их иерархию (действительно, "для себя" всегда можно в производных классах предусмотреть доступ к защищенным членам).

Неочевидное следствие из этого: нельзя составить полный и окончательный список всех функций, которым будет доступен защищенный член, поскольку всегда можно добавить еще одну, определив ее как функцию-член в новом производном классе. Для метода абстракции данных такой подход часто бывает мало приемлемым. Если язык ориентируется на метод абстракции данных, то очевидное для него решение - это требование указывать в описании класса список всех функций, которым нужен доступ к члену. В C++ для этой цели используется описание частных (private) членов. Оно использовалось и в приводившихся описаниях классов complex и shape.

Важность инкапсуляции, т.е. заключения членов в защитную оболочку, резко возрастает с ростом размеров программы и увеличивающимся разбросом областей приложения. В § 6.6 более подробно обсуждаются возможности языка по инкапсуляции.

1.6 Пределы совершенства

Язык C++ проектировался как "лучший C", поддерживающий абстракцию данных и объектно-ориентированное программирование. При этом он должен быть пригодным для большинства основных задач системного программирования.

Основная трудность для языка, который создавался в расчете на методы упрятывания данных, абстракции данных и объектно-ориентированного программирования, в том, что для того, чтобы быть языком общего назначения, он должен:

- идти на традиционных машинах;
- сосуществовать с традиционными операционными системами и языками;
- соперничать с традиционными языками программирования в эффективности выполнения программы;
- быть пригодным во всех основных областях приложения.

Это значит, что должны быть возможности для эффективных числовых операций (арифметика с плавающей точкой без особых накладных расходов, иначе пользователь предпочтет Фортран) и средства такого доступа к памяти, который позволит писать на этом языке драйверы устройств. Кроме того, надо уметь писать вызовы функций в достаточно непривычной записи, принятой для обращений в традиционных операционных системах. Наконец, должна быть возможность из языка, поддерживающего объектно-ориентированное программирование, вызывать функции, написанные на других языках, а из других языков вызывать функцию на этом языке, поддерживающем объектно-ориентированное программирование.

Далее, нельзя рассчитывать на широкое использование искомого языка программирования как языка общего назначения, если реализация его целиком полагается на возможности, которые отсутствуют в машинах с традиционной архитектурой.

Если не вводить в язык возможности низкого уровня, то придется для основных задач большинства областей приложения использовать некоторые языки низкого уровня, например C или ассемблер. Но C++ проектировался с расчетом, что в нем можно сделать все, что допустимо на C, причем без увеличения времени выполнения. Вообще, C++ проектировался, исходя из принципа, что не должно возникать никаких дополнительных затрат времени и памяти, если только этого явно не пожелает сам программист.

Язык проектировался в расчете на современные методы трансляции, которые обеспечивают проверку согласованности программы, ее эффективность и компактность представления. Основным средством борьбы со сложностью программ видится, прежде всего, строгий контроль типов и инкапсуляция. Особенно это касается больших программ, создаваемых многими людьми. Пользователь может не

являться одним из создателей таких программ, и может вообще не быть программистом. Поскольку никакую настоящую программу нельзя написать без поддержки библиотек, создаваемых другими программистами, последнее замечание можно отнести практически ко всем программам.

C++ проектировался для поддержки того принципа, что всякая программа есть модель некоторых существующих в реальности понятий, а класс является конкретным представлением понятия, взятого из области приложения (§ 12.2). Поэтому классы пронизывают всю программу на C++, и налагаются жесткие требования на гибкость понятия класса, компактность объектов класса и эффективность их использования. Если работать с классами будет неудобно или слишком накладно, то они просто не будут использоваться, и программы выродятся в программы на "лучшем C". Значит пользователь не сумеет насладиться теми возможностями, ради которых, собственно, и создавался язык.

Описания и константы

В данной лекции описаны основные типы (char, int, float и т.д.) и способы построения на их основе новых типов (функций, векторов, указателей и т.д.). Описание вводит в программу имя, указав его тип и, возможно, начальное значение. В этой лекции вводятся такие понятия, как описание и определение, типы, область видимости имен, время жизни объектов. Даются обозначения литеральных констант C++ и способы задания символических констант. Приводятся примеры, которые просто демонстрируют возможности языка. Более осмысленные примеры, иллюстрирующие возможности выражений и операторов языка C++, будут приведены в следующей лекции. В этой лекции лишь упоминаются средства для определения пользовательских типов и операций над ними. Они обсуждаются в лекциях 5 и 7.

2.1 Описания

Имя (идентификатор) следует описать прежде, чем оно будет использоваться в программе на C++. Это означает, что нужно указать его тип, чтобы транслятор знал, к какого вида объектам относится имя. Ниже приведены несколько примеров, иллюстрирующих все разнообразие описаний:

```
char ch;
int count = 1;
char* name = "Njal";
struct complex { float re, im; };
complex cvar;
extern complex sqrt(complex);
extern int error_number;
typedef complex point;
float real(complex* p) { return p->re; };
const double pi = 3.1415926535897932385;
struct user;
template<class T> abs(T a) { return a<0 ? -a : a; }
enum beer { Carlsberg, Tuborg, Thor };
```

Из этих примеров видно, что роль описаний не сводится лишь к

привязке типа к имени. Большинство указанных описаний одновременно являются определениями, т.е. они создают объект, на который ссылается имя. Для `ch`, `count`, `name` и `cvar` таким объектом является элемент памяти соответствующего размера. Этот элемент будет использоваться как переменная, и говорят, что для него отведена память. Для `real` подобным объектом будет заданная функция. Для константы `pi` объектом будет число 3.14159265358979323 85. Для `complex` объектом будет новый тип. Для `point` объектом является тип `complex`, поэтому `point` становится синонимом `complex`. Следующие описания уже не являются определениями:

```
extern complex sqrt(complex);
extern int error_number;
struct user;
```

Это означает, что объекты, введенные ими, должны быть определены где-то в другом месте программы. Тело функции `sqrt` должно быть указано в каком-то другом описании. Память для переменной `error_number` типа `int` должна выделяться в результате другого описания `error_number`. Должно быть и какое-то другое описание типа `user`, из которого можно понять, что это за тип. В программе на языке C++ должно быть только одно определение каждого имени, но описаний может быть много. Однако все описания должны быть согласованы по типу вводимого в них объекта. Поэтому в приведенном ниже фрагменте содержатся две ошибки:

```
int count;
int count;          // ошибка: переопределение

extern int error_number;
extern short error_number; // ошибка: несоответствие типов
```

Зато в следующем фрагменте нет ни одной ошибки (об использовании `extern`):

```
extern int error_number;
extern int error_number;
```

В некоторых описаниях указываются "значения" объектов, которые они

определяют:

```
struct complex { float re, im; };
typedef complex point;
float real(complex* p) { return  p->re };
const double pi = 3.1415926535897932385;
```

Для типов, функций и констант "значение" остается неизменным; для данных, не являющихся константами, начальное значение может впоследствии изменяться:

```
int count = 1;
char* name = "Bjarne";
//...
count = 2;
name = "Marian";
```

Из всех определений только следующее не задает значения:

```
char ch;
```

Всякое описание, которое задает значение, является определением.

2.1.1 Область видимости

Описанием определяется область видимости имени. Это значит, что имя может использоваться только в определенной части текста программы. Если имя описано в функции (обычно его называют "локальным именем"), то область видимости имени простирается от точки описания до конца блока, в котором появилось это описание. Если имя не находится в описании функции или класса (его обычно называют "глобальным именем"), то область видимости простирается от точки описания до конца файла, в котором появилось это описание. Описание имени в блоке может скрывать описание в объемлющем блоке или глобальное имя; т.е. имя может быть переопределено так, что оно будет обозначать другой объект внутри блока. После выхода из блока прежнее значение имени (если оно было) восстанавливается. Приведем пример:

```
int x;          // глобальное x

void f()
{
    int x;      // локальное x скрывает глобальное x
    x = 1;      // присвоить локальному x
    {
        int x;  // скрывает первое локальное x
        x = 2;  // присвоить второму локальному x
    }
    x = 3;      // присвоить первому локальному x
}

int* p = &x;    // взять адрес глобального x
```

В больших программах не избежать переопределения имен. К сожалению, человек легко может проглядеть такое переопределение. Возникающие из-за этого ошибки найти непросто, возможно потому, что они достаточно редки. Следовательно, переопределение имен следует свести к минимуму. Если вы обозначаете глобальные переменные или локальные переменные в большой функции такими именами, как `i` или `x`, то сами напрашиваетесь на неприятности.

Есть возможность с помощью операции разрешения области видимости :: обратиться к скрытому глобальному имени, например:

```
int x;

void f2()
{
  int x = 1;  // скрывает глобальное x
  ::x = 2;    // присваивание глобальному x
}
```

Возможность использовать скрытое локальное имя отсутствует.

Область видимости имени начинается в точке его описания (по окончании описателя, но еще до начала инициализатора). Это означает, что имя можно использовать даже до того, как задано его начальное значение. Например:

```
int x;

void f3()
{
int x = x;    // ошибочное присваивание
}
```

Такое присваивание недопустимо и лишено смысла. Если вы попытаетесь транслировать эту программу, то получите предупреждение: "использование до задания значения". Вместе с тем, не применяя оператора ::, можно использовать одно и то же имя для обозначения двух различных объектов блока. Например:

```
int x = 11;

 void f4()        // извращенный пример
 {
   int y = x;   //  глобальное x
   int x = 22;
   y = x;       // локальное x
}
```

Переменная y инициализируется значением глобального `x`, т.е. 11, а затем ей присваивается значение локальной переменной `x`, т.е. 22. Имена формальных параметров функции считаются описанными в самом большом блоке функции, поэтому в описании ниже есть ошибка:

```
void f5(int x)
 {
  int x;       // ошибка
 }
```

Здесь x определено дважды в одной и той же области видимости. Это хотя и не слишком редкая, но довольно тонкая ошибка.

2.1.2 Объекты и адреса

Можно выделять память для "переменных", не имеющих имен, и

использовать эти переменные. Возможно даже присваивание таким странно выглядящим "переменным", например, `*p[a+10]=7`. Следовательно, есть потребность именовать "нечто хранящееся в памяти". Можно привести подходящую цитату из справочного руководства: "Любой объект - это некоторая область памяти, а адресом называется выражение, ссылающееся на объект или функцию". Слову адрес (*`lvalue - left value`*, т.е. величина слева) первоначально приписывался смысл "нечто, что может в присваивании стоять слева". Адрес может ссылаться и на константу. Адрес, который не был описан со спецификацией `const`, называется изменяемым адресом.

2.1.3 Время жизни объектов

Если только программист не вмешается явно, объект будет создан при появлении его определения и уничтожен, когда исчезнет из области видимости. Объекты с глобальными именами создаются, инициализируются (причем только один раз) и существуют до конца программы. Если локальные объекты описаны со служебным словом `static`, то они также существуют до конца программы. Инициализация их происходит, когда в первый раз управление "проходит через" описание этих объектов, например:

```
int a = 1;

void f()
{
  int b = 1;   // инициализируется при каждом вызове f()
  static int c = a;  // инициализируется только один раз
  cout << " a = " << a++
   << " b = " << b++
   << " c = " << c++ << '\n';
}

int main()
{
  while (a < 4) f();
}
```

Здесь программа выдаст такой результат:

```
a = 1 b = 1 c = 1
a = 2 b = 1 c = 2
a = 3 b = 1 c = 3
```

Из примеров этой лекции для краткости изложения исключена макрокоманда `#include <iostream>`. Она нужна лишь в тех из них, которые выдают результат.

Операция "++" является инкрементом, т. е. a++ означает: добавить 1 к переменной a.

Глобальная переменная или локальная переменная `static`, которая не была явно инициализирована, инициализируется неявно нулевым значением. Используя операции `new` и `delete`, программист может создавать объекты, временем жизни которых он управляет сам.

2.2 Имена

Имя (идентификатор) является последовательностью букв или цифр. Первый символ должен быть буквой. Буквой считается и символ подчеркивания _. Язык C++ не ограничивает число символов в имени. Но в реализацию входят программные компоненты, которыми создатель транслятора управлять не может (например, загрузчик), а они, к сожалению, могут устанавливать ограничения. Кроме того, некоторые системные программы, необходимые для выполнения программы на C ++, могут расширять или сужать множество символов, допустимых в идентификаторе. Расширения (например, использование `$` в имени) могут нарушить переносимость программы. Нельзя использовать в качестве имен служебные слова C++, например:

```
hello   this_is_a_most_unusially_long_name
DEFINED foO   bAr   u_name   HorseSense
var0    var1  CLASS  _class   ___
```

Теперь приведем примеры последовательностей символов, которые не могут использоваться как идентификаторы:

```
012     a fool  $sys    class   3var
pay.due foo~bar .name   if
```

Заглавные и строчные буквы считаются различными, поэтому Count и count - разные имена. Но выбирать имена, почти не отличающиеся друг от друга, неразумно. Все имена, начинающиеся с символа подчеркивания, резервируются для использования в самой реализации или в тех программах, которые выполняются совместно с рабочей, поэтому крайне легкомысленно вставлять такие имена в свою программу.

При разборе программы транслятор всегда стремится выбрать самую длинную последовательность символов, образующих имя, поэтому var10 - это имя, а не идущие подряд имя var и число 10. По той же причине `elseif` - одно имя (служебное), а не два служебных имени `else` и `if`.

2.3 Типы

С каждым именем (идентификатором) в программе связан тип. Он задает те операции, которые могут применяться к имени (т.е. к объекту, который обозначает имя), а также интерпретацию этих операций. Приведем примеры:

```
int error_number;
float real(complex* p);
```

Поскольку переменная `error_number` описана как `int` (целое), ей можно присваивать, а также можно использовать ее значения в арифметических выражениях. Функцию `real` можно вызывать с параметром, содержащим адрес `complex`. Можно получать адреса и переменной, и функции. Некоторые имена, как в нашем примере `int` и `complex`, являются именами типов. Обычно имя типа нужно, чтобы задать в описании типа некоторое другое имя. Кроме того, имя типа может использоваться в качестве операнда в операциях `sizeof` (с ее помощью определяют размер памяти, необходимый для объектов этого типа) и `new` (с ее помощью можно разместить в свободной памяти объект этого типа). Например:

```
int main()
{
  int* p = new int;
  cout << "sizeof(int) = " << sizeof(int) '\n';
}
```

Еще имя типа может использоваться в операции явного преобразования одного типа к другому, например:

```
float f;
char* p;
//...
long ll = long(p);     // преобразует p в long
int i = int(f); // преобразует f в int
```

2.3.1 Основные типы

Основные типы C++ представляют самые распространенные единицы памяти машин и все основные способы работы с ними. Это:

```
char
short int
int
long int
```

Перечисленные типы используются для представления различного размера целых. Числа с плавающей точкой представлены типами:

```
float
double
long double
```

Следующие типы могут использоваться для представления беззнаковых целых, логических значений, разрядных массивов и т.д.:

```
unsigned char
unsigned short int
unsigned int
unsigned long int
```

Ниже приведены типы, которые используются для явного задания знаковых типов:

```
signed char
signed short int
signed int
signed long int
```

Поскольку по умолчанию значения типа `int` считаются знаковыми, то соответствующие типы с signed являются синонимами типов без этого служебного слова. Но тип `signed char` представляет особый интерес: все 3 типа - `unsigned char`, `signed char` и просто `char` считаются различными. Для краткости (и это не влечет никаких последствий) слово `int` можно не указывать в многословных типах, т.е. `long` означает `long int`, `unsigned` - `unsigned int`. Вообще, если в описании не указан тип, то предполагается, что это `int`. Например, ниже даны два определения объекта типа `int`:

```
const a = 1;     // небрежно, тип не указан
 static x;       // тот же случай
```

Все же обычно пропуск типа в описании в надежде, что по умолчанию это будет тип int, считается дурным стилем. Он может вызвать тонкий и нежелательный эффект.

Для хранения символов и работы с ними наиболее подходит тип `char`. Обычно он представляет байт из 8 разрядов. Размеры всех объектов в C ++ кратны размеру `char`, и по определению значение `sizeof(char)` тождественно 1. В зависимости от машины значение типа `char` может быть знаковым или беззнаковым целым. Конечно, значение типа `unsigned char` всегда беззнаковое, и, задавая явно этот тип, мы улучшаем переносимость программы. Однако, использование `unsigned char` вместо `char` может снизить скорость выполнения программы. Естественно, значение типа `signed char` всегда знаковое.

В язык введено несколько целых, несколько беззнаковых типов и несколько типов с плавающей точкой, чтобы программист мог полнее использовать возможности системы команд. У многих машин

значительно различаются размеры выделяемой памяти, время доступа и скорость вычислений для значений различных основных типов. Как правило, зная особенности конкретной машины, легко выбрать оптимальный основной тип (например, один из типов `int`) для данной переменной. Однако, написать действительно переносимую программу, использующую такие возможности низкого уровня, непросто. Для размеров основных типов выполняются следующие соотношения:

```
1==sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)

sizeof(float)<=sizeof(double)<=sizeof(long double)

sizeof(I)==sizeof(signed I)==sizeof(unsigned I)
```

Здесь I может быть типа `char`, `short`, `int` или `long`. Помимо этого гарантируется, что `char` представлен не менее, чем 8 разрядами, `short` - не менее, чем 16 разрядами и `long` - не менее, чем 32 разрядами. Тип `char` достаточен для представления любого символа из набора символов данной машины. Но это означает только то, что тип `char` может представлять целые в диапазоне 0..127. Предположить большее - рискованно.

Типы беззнаковых целых больше всего подходят для таких программ, в которых память рассматривается как массив разрядов. Но, как правило, использование `unsigned` вместо `int`, не дает ничего хорошего, хотя таким образом рассчитывали выиграть еще один разряд для представления положительных целых. Описывая переменную как `unsigned`, нельзя гарантировать, что она будет только положительной, поскольку допустимы неявные преобразования типа, например:

```
unsigned surprise = -1;
```

Это определение допустимо (хотя компилятор может выдать предупреждение о нем).

2.3.2 Неявное преобразование типа

В присваивании и выражении основные типы могут совершенно свободно использоваться совместно. Значения преобразовываются всюду, где это возможно, таким образом, чтобы информация не терялась.

Все-таки есть ситуации, когда информация может быть потеряна или даже искажена. Потенциальным источником таких ситуаций становятся присваивания, в которых значение одного типа присваивается значению другого типа, причем в представлении последнего используется меньше разрядов. Допустим, что следующие присваивания выполняются на машине, в которой целые представляются в дополнительном коде, и символ занимает 8 разрядов:

```
int i1 = 256+255;
char ch = i1        // ch == 255
int i2 = ch;        // i2 == ?
```

В присваивании `ch=i1` теряется один разряд (и самый важный!), а когда мы присваиваем значение переменной `i2`, у переменной `ch` значение "все единицы", т.е. 8 единичных разрядов. Но какое значение примет `i2`? На машине DEC VAX, в которой char представляет знаковые значения, это будет -1, а на машине Motorola 68K, в которой `char` - беззнаковый, это будет 255. В C++ нет динамических средств контроля подобных ситуаций, а контроль на этапе трансляции вообще слишком сложен, поэтому надо быть осторожными.

2.3.3 Производные типы

Исходя из основных (и определенных пользователем) типов, можно с помощью следующих операций описания:

```
*    указатель
&    ссылка
[]   массив
()   функция
```

а также с помощью определения структур, задать другие, производные типы. Например:

```
int* a;
float v[10];
char* p[20];   // массив из 20 символьных указателей
void f(int);
struct str { short length; char* p; };
```

Ключевая идея состоит в том, что описание объекта производного типа должно отражать его использование, например:

```
int v[10];     // описание вектора
i = v[3];      // использование элемента вектора

int* p;        // описание указателя
i = *p;        // использование указуемого объекта
```

Обозначения, используемые для производных типов, достаточно трудны для понимания лишь потому, что операции * и & являются префиксными, а [] и () - постфиксными. Поэтому в задании типов, если приоритеты операций не отвечают цели, надо ставить скобки. Например, приоритет операции [] выше, чем у *, и мы имеем:

```
int* v[10];       // массив указателей
int (*p)[10];     // указатель массива
```

Большинство людей просто запоминает, как выглядят наиболее часто употребляемые типы. Можно описать сразу несколько имен в одном описании. Тогда оно содержит вместо одного имени список отделяемых друг от друга запятыми имен. Например, можно так описать две переменные целого типа:

```
int x, y;    // int x; int y;
```

Когда мы описываем производные типы, не надо забывать, что операции описаний применяются только к данному имени (а вовсе не ко всем остальным именам того же описания). Например:

```
int* p, y;       // int* p; int y; НО НЕ int* y;
int x, *p;       // int x; int* p;
int v[10], *p;   // int v[10]; int* p;
```

Но такие описания запутывают программу, и, возможно, их следует избегать.

2.3.4 Тип void

Тип void синтаксически эквивалентен основным типам, но использовать его можно только в производном типе. Объектов типа `void` не существует. С его помощью задаются указатели на объекты неизвестного типа или функции, не возвращающие значение.

```
void f();    // f не возвращает значения
void* pv;    // указатель на объект неизвестного типа
```

Указатель произвольного типа можно присваивать переменной типа void*. На первый взгляд этому трудно найти применение, поскольку для `void*` недопустимо косвенное обращение (разыменование). Однако, именно на этом ограничении основывается использование типа `void*`. Он приписывается параметрам функций, которые не должны знать истинного типа этих параметров. Тип `void*` имеют также бестиповые объекты, возвращаемые функциями.

Для использования таких объектов нужно выполнить явную операцию преобразования типа. Такие функции обычно находятся на самых нижних уровнях системы, которые управляют аппаратными ресурсами. Приведем пример:

```
void* malloc(unsigned size);
void free(void*);

void f()          // распределение памяти в стиле Си
{
  int* pi = (int*)malloc(10*sizeof(int));
  char* pc = (char*)malloc(10);
  //...
  free(pi);
  free(pc);
}
```

Обозначение: (тип) выражение - используется для задания операции преобразования выражения к типу, поэтому перед присваиванием `pi` тип `void*`, возвращаемый в первом вызове `malloc()`, преобразуется в тип `int`. Пример записан в архаичном стиле.

2.3.5 Указатели

Для большинства типов `T` указатель на `T` имеет тип `T*`. Это значит, что переменная типа `T*` может хранить адрес объекта типа `T`. Указатели на массивы и функции, к сожалению, требуют более сложной записи:

```
int* pi;
char** cpp;              // указатель на указатель на char
int (*vp)[10];           // указатель на массив из 10 целых
int (*fp)(char, char*); // указатель на функцию с параметрами
                         // char  и char*,  возвращающую int
```

Главная операция над указателями - это косвенное обращение (разыменование), т.е. обращение к объекту, на который настроен указатель. Эту операцию обычно называют просто косвенностью. Операция косвенности * является префиксной унарной операцией. Например:

```
char c1 = 'a';
char* p = &c1;    // p содержит адрес c1
char c2 = *p;    // c2 = 'a'
```

Переменная, на которую указывает `p`,- это `c1`, а значение, которое хранится в `c1`, равно 'a'. Поэтому присваиваемое `c2` значение *p есть 'a'. Над указателями можно выполнять и некоторые арифметические операции. Ниже в качестве примера представлена функция, подсчитывающая число символов в строке, заканчивающейся нулевым символом (который не учитывается):

```
int strlen(char* p)
 {
   int i = 0;
   while (*p++) i++;
```

```
    return i;
  }
```

Можно определить длину строки по-другому: сначала найти ее конец, а затем вычесть адрес начала строки из адреса ее конца.

```
int strlen(char* p)
{
char* q = p;
while (*q++) ;
return q-p-1;
}
```

2.3.6 Массивы

Для типа `T T[size]` является типом "массива из `size` элементов типа T". Элементы индексируются от 0 до `size-1`. Например:

```
float v[3];   // массив из трех чисел с плавающей точкой:
              //     v[0], v[1], v[2]
int a[2][5];  // два массива, из пяти целых каждый
char* vpc;    // указатель на char
```

Можно следующим образом записать цикл, в котором печатаются целые значения прописных букв:

```
extern "C" int strlen(const char*);  // из <string.h>

char alpha[] = "abcdefghijklmnopqrstuvwxyz";

main()
{
  int sz = strlen(alpha);

  for (int i=0; i<sz; i++) {
    char ch = alpha[i];
      cout << '\''<< ch << '\''
        << " = " <<int(ch)
```

```
            << " = 0" << oct(ch)
            << " = 0x" << hex(ch) << '\n';
    }
}
```

Здесь функции `oct()` и `hex()` выдают свой параметр целого типа в восьмеричном и шестнадцатеричном виде соответственно. Обе функции описаны в `<iostream.h>`. Для подсчета числа символов в `alpha` используется функция `strlen()` из `<string.h>`, но вместо нее можно было использовать размер массива alpha. Для множества символов ASCII результат будет таким:

```
'a' = 97 = 0141 = 0x61
'b' = 98 = 0142 = 0x62
'c' = 99 = 0143 = 0x63
...
```

Отметим, что не нужно указывать размер массива `alpha`: транслятор установит его, подсчитав число символов в строке, заданной в качестве инициализатора. Задание массива символов в виде строки инициализатора - это удобный, но к сожалению, единственный способ подобного применения строк. Присваивание строки массиву недопустимо, поскольку в языке присваивание массивам не определено, например:

```
char v[9];
v = "a string";          // ошибка
```

Классы позволяют реализовать представление строк с большим набором операций.

Очевидно, что строки пригодны только для инициализации символьных массивов; для других типов приходится использовать более сложную запись. Впрочем, она может использоваться и для символьных массивов. Например:

```
int  v1[] = { 1, 2, 3, 4 };
int  v2[] = { 'a', 'b', 'c', 'd' };

char v3[] = { 1, 2, 3, 4 };
```

```
char v4[] = { 'a', 'b', 'c', 'd' };
```

Здесь `v3` и `v4` - массивы из четырех (а не пяти) символов; `v4` не оканчивается нулевым символом, как того требуют соглашение о строках и большинство библиотечных функций. Используя такой массив `char` мы сами готовим почву для будущих ошибок.

Многомерные массивы представлены как массивы массивов. Однако нельзя при задании граничных значений индексов использовать, как это делается в некоторых языках, запятую. Запятая - это особая операция для перечисления выражений. Можно попробовать задать такое описание:

```
int bad[5,2];    // ошибка
```

или такое

```
int v[5][2];
int bad = v[4,1];      // ошибка
int good = v[4][1];    // правильно
```

Ниже описывается массив из двух элементов, каждый из которых является, в свою очередь, массивом из 5 элементов типа char:

```
char v[2][5];
```

В следующем примере первый массив инициализируется пятью первыми буквами алфавита, а второй - пятью младшими цифрами.

```
char v[2][5] = {
      { 'a', 'b', 'c', 'd', 'e' },
      { '0', '1', '2', '3', '4' }
  };

  main() {
     for (int i = 0; i<2; i++) {
       for (int j = 0; j<5; j++)
          cout << "v[" << i << "][" << j
              << "]=" << v[i][j] << "  ";
          cout << '\n';
```

```
    }
  }
```

В результате получим:

```
v[0][0]=a v[0][1]=b v[0][2]=c v[0][3]=d v[0][4]=e
v[1][0]=0 v[1][1]=1 v[1][2]=2 v[1][3]=3 v[1][4]=4
```

2.3.7 Указатели и массивы

Указатели и массивы в языке Си++ тесно связаны. Имя массива можно использовать как указатель на его первый элемент, поэтому пример с массивом `alpha` можно записать так:

```
int main()
 {
  char alpha[] = "abcdefghijklmnopqrstuvwxyz";
  char* p = alpha;
   char ch;

   while (ch = *p++)
       cout << ch << " = " << int (ch)
           << " = 0" << oct(ch) << '\n';
}
```

Можно также задать описание `p` следующим образом:

```
char* p = &alpha[0];
```

Эта эквивалентность широко используется при вызовах функций с параметром-массивом, который всегда передается как указатель на его первый элемент. Таким образом, в следующем примере в обоих вызовах `strlen` передается одно и то же значение:

```
void f()
{
  extern "C" int strlen(const char*);  // из <string.h>
  char v[] = "Annemarie";
```

```
  char* p = v;
  strlen(p);
  strlen(v);
}
```

Но в том и загвоздка, что обойти это нельзя: не существует способа так описать функцию, чтобы при ее вызове массив `v` копировался. Результат применения к указателям арифметических операций +, -, ++ или -- зависит от типа указуемых объектов. Если такая операция применяется к указателю `p` типа `T*`, то считается, что p указывает на массив объектов типа `T`. Тогда `p+1` обозначает следующий элемент этого массива, а `p-1` - предыдущий элемент. Отсюда следует, что значение (адрес) `p+1` будет на `sizeof(T)` байтов больше, чем значение `p`. Поэтому в следующей программе

```
main()
    {
 char cv[10];
 int iv[10];

 char* pc = cv;
 int* pi = iv;

 cout << "char* " << long(pc+1)-long(pc) << '\n';
 cout << "int*  " << long(pi+1)-long(pi) << '\n';
    }
```

с учетом того, что на машине автора (Macintosh) символ занимает один байт, а целое - четыре байта, получим:

```
char* 1
int*  4
```

Перед вычитанием указатели были явной операцией преобразованы к типу `long`. Он использовался для преобразования вместо "очевидного" типа `int`, поскольку в некоторых реализациях языка C++ указатель может не поместиться в тип `int` (т.е. `sizeof(int) <(sizeof(char*))`). Вычитание указателей определено только в том случае, когда они оба указывают на один и тот же массив (хотя в

языке нет возможностей гарантировать этот факт). Результат вычитания одного указателя из другого равен числу (целое) элементов массива, находящихся между этими указателями. Можно складывать с указателем или вычитать из него значение целого типа; в обоих случаях результатом будет указатель. Если получится значение, не являющееся указателем на элемент того же массива, на который был настроен исходный указатель (или указателем на следующий за массивом элемент), то результат использования такого значения неопределен. Приведем пример:

```
void f()
{
  int v1[10];
  int v2[10];

  int i = &v1[5]-&v1[3];    // 2
  i = &v1[5]-&v2[3];    // неопределенный результат

   int* p = v2+2;           // p == &v2[2]
       p = v2-2;           // *p неопределено
}
```

Как правило, сложных арифметических операций с указателями не требуется и лучше всего их избегать. Следует сказать, что в большинстве реализаций языка C++ нет контроля над границами массивов. Описание массива не является самодостаточным, поскольку необязательно в нем будет храниться число элементов массива. Понятие массива в C является, по сути, понятием языка низкого уровня. Классы помогают развить его.

2.3.8 Структуры

Массив представляет собой совокупность элементов одного типа, а структура является совокупностью элементов произвольных (практически) типов. Например:

```
struct address {
    char* name;        // имя "Jim Dandy"
```

```
    long number;      // номер дома 61
    char* street;     // улица "South Street"
    char* town;       // город "New Providence"
    char* state[2];   // штат 'N' 'J'
     int  zip;        // индекс 7974
};
```

Здесь определяется новый тип, называемый `address`, который задает почтовый адрес. Определение не является достаточно общим, чтобы учесть все случаи адресов, но оно вполне пригодно для примера. Обратите внимание на точку с запятой в конце определения: это один из немногих в C++ случаев, когда после фигурной скобки требуется точка с запятой, поэтому про нее часто забывают. Переменные типа `address` можно описывать точно так же, как и любые другие переменные, а с помощью операции . (точка) можно обращаться к отдельным членам структуры. Например:

```
address jd;
jd.name = "Jim Dandy";
jd.number = 61;
```

Инициализировать переменные типа `struct` можно так же, как массивы. Например:

```
address jd = {
  "Jim Dandy",
   61, "South Street",
   "New Providence", {'N','J'}, 7974
};
```

Но лучше для этих целей использовать конструктор. Отметим, что jd.state нельзя инициализировать строкой "NJ". Ведь строки оканчиваются нулевым символом '\0', значит в строке "NJ" три символа, а это на один больше, чем помещается в jd.state. К структурным объектам часто обращаются с помощью указателей, используя операцию ->. Например:

```
void print_addr(address* p)
{
 cout << p->name << '\n'
```

```
        << p->number << ' ' << p->street << '\n'
        << p->town << '\n'
        << p->state[0] << p->state[1]
       << ' ' << p->zip << '\n';
}
```

Объекты структурного типа могут быть присвоены, переданы как фактические параметры функций и возвращены функциями в качестве результата. Например:

```
address current;

address set_current(address next)
{
  address prev = current;
  current = next;
  return prev;
}
```

Другие допустимые операции, например, такие, как сравнение (== и !=), неопределены. Однако пользователь может сам определить эти операции (см. лекцию 7).

Размер объекта структурного типа не обязательно равен сумме размеров всех его членов. Это происходит по той причине, что на многих машинах требуется размещать объекты определенных типов, только выравнивая их по некоторой зависящей от системы адресации границе (или просто потому, что работа при таком выравнивании будет более эффективной). Типичный пример - это выравнивание целого по условной границе. В результате выравнивания могут появиться "дырки" в структуре. Так, на уже упоминавшейся машине автора sizeof(address) равно 24, а не 22, как можно было ожидать. Следует также упомянуть, что тип можно использовать сразу после его появления в описании, еще до того, как будет завершено все описание. Например:

```
struct link{
        link* previous;
        link* successor;
 };
```

Однако новые объекты типа структуры нельзя описать до тех пор, пока не появится ее полное описание. Поэтому описание

```
struct no_good {
  no_good member;
};
```

является ошибочным (транслятор не в состоянии установить размер `no_good`). Чтобы позволить двум (или более) структурным типам ссылаться друг на друга, можно просто описать имя одного из них как имя некоторого структурного типа. Например:

```
struct list;        // будет определено позднее

 struct link {
    link* pre;
    link* suc;
    list* member_of;
 };

struct list {
     link* head;
};
```

Если бы не было первого описания `list`, описание члена `link` привело бы к синтаксической ошибке. Можно также использовать имя структурного типа еще до того, как тип будет определен, если только это использование не предполагает знания размера структуры. Например:

```
class S;        // 'S' - имя некоторого типа

extern S a;

S f();

void g(S);
```

Но приведенные описания можно использовать лишь после того, как тип `S` будет определен:

```
void h()
{
  S a;       // ошибка: S - неописано
  f();       // ошибка: S - неописано
  g(a);      // ошибка: S - неописано
}
```

2.3.9 Эквивалентность типов

Два структурных типа считаются различными даже тогда, когда они имеют одни и те же члены. Например, ниже определены различные типы:

```
struct s1 { int a; };
struct s2 { int a; };
```

В результате имеем:

```
s1 x;
s2 y = x;    // ошибка: несоответствие типов
```

Кроме того, структурные типы отличаются от основных типов, поэтому получим:

```
s1 x;
int i = x;    // ошибка: несоответствие типов
```

Есть, однако, возможность, не определяя новый тип, задать новое имя для типа. В описании, начинающемся служебным словом `typedef`, описывается не переменная указанного типа, а вводится новое имя для типа. Приведем пример:

```
typedef char* Pchar;
Pchar p1, p2;
char* p3 = p1;
```

Это просто удобное средство сокращения записи.

2.3.10 Ссылки

Ссылку можно рассматривать как еще одно имя объекта. В основном ссылки используются для задания параметров и возвращаемых функциями значений , а также для перегрузки операций. Запись X& обозначает ссылку на `X`. Например:

```
int i = 1;
int& r = i;    // r и i ссылаются на одно и то же целое
int x = r;     // x = 1
r = 2;         // i = 2;
```

Ссылка должна быть инициализирована, т.е. должно быть нечто, что она может обозначать. Следует помнить, что инициализация ссылки совершенно отличается от операции присваивания. Хотя можно указывать операции над ссылкой, ни одна из них на саму ссылку не действует, например,

```
int ii = 0;
int& rr = ii;
rr++;          // ii увеличивается на 1
```

Здесь операция `++` допустима, но `rr++` не увеличивает саму ссылку `rr` ; вместо этого `++` применяется к целому, т.е. к переменной `ii`. Следовательно, после инициализации значение ссылки не может быть изменено: она всегда указывает на тот объект, к которому была привязана при ее инициализации. Чтобы получить указатель на объект, обозначаемый ссылкой `rr`, можно написать `&rr`. Очевидной реализацией ссылки может служить постоянный указатель, который используется только для косвенного обращения. Тогда инициализация ссылки будет тривиальной, если в качестве инициализатора указан адрес (т.е. объект, адрес которого можно получить). Инициализатор для типа `T` должен быть адресом. Однако, инициализатор для `&T` может быть и не адресом, и даже не типом `T`. В таких случаях делается следующее:

1. во-первых, если необходимо, применяется преобразование типа;
2. затем получившееся значение помещается во временную

переменную;

3. наконец, адрес этой переменной используется в качестве инициализатора ссылки.

Пусть имеются описания:

```
double& dr = 1;          // ошибка: нужен адрес
const double& cdr = 1;   // нормально
```

Это интерпретируется так:

```
double* cdrp;    // ссылка, представленная как указатель
double temp;
temp = double(1);
cdrp = &temp;
```

Ссылки на переменные и ссылки на константы различаются по следующей причине: в первом случае создание временной переменной чревато ошибками, поскольку присваивание этой переменной означает присваивание временной переменной, которая могла к этому моменту исчезнуть. Естественно, что во втором случае подобных проблем не существует и ссылки на константы часто используются как параметры функций. Ссылка может использоваться для функции, которая изменяет значение своего параметра. Например:

```
void incr(int& aa) { aa++; }

void f()
{
 int x = 1;
 incr(x);     // x = 2
}
```

По определению передача параметров имеет ту же семантику, что и инициализация, поэтому при вызове функции `incr` ее параметр aa становится другим именем для `x`. Лучше, однако, избегать изменяющих свои параметры функций, чтобы не запутывать программу. В большинстве случаев предпочтительнее, чтобы функция возвращала результат явным образом, или чтобы использовался параметр типа указателя:

```
int next(int p) { return p+1; }
void inc(int* p) { (*p)++; }

void g()
{
  int x = 1;
  x = next(x);     // x = 2
  inc(&x);  // x = 3
}
```

Кроме перечисленного, с помощью ссылок можно определить функции, используемые как в правой, так и в левой частях присваивания. Наиболее интересное применение это обычно находит при определении нетривиальных пользовательских типов. В качестве примера определим простой ассоциативный массив. Начнем с определения структуры `pair`:

```
struct pair {
    char* name;  // строка
    int val;     // целое
  };
```

Идея заключается в том, что со строкой связывается некоторое целое значение. Нетрудно написать функцию поиска `find()`, которая работает со структурой данных, представляющей ассоциативный массив. В нем для каждой отличной от других строки содержится структура `pair` (пара: строка и значение). В данном примере - это просто массив. Чтобы сократить пример, используется предельно простой, хотя и неэффективный алгоритм:

```
const int large = 1024;
static pair vec[large+1];

pair* find(const char* p)
      /*
// работает со множеством пар "pair":
// ищет p, если находит, возвращает его "pair",
// в противном случае возвращает неиспользованную "pair"
      */
```

```
    {
  for (int i=0; vec[i].name; i++)
     if (strcmp(p,vec[i].name)==0) return &vec[i];

  if (i == large) return &vec[large-1];

  return &vec[i];
        }
```

Эту функцию использует функция `value()`, которая реализует массив целых, индексируемый строками (хотя привычнее строки индексировать целыми):

```
  int& value(const char* p)
  {
    pair* res = find(p);
    if (res->name == 0) {  // до сих пор строка не встречалась,
      // значит надо инициализировать
      res->name = new char[strlen(p)+1];
      strcpy(res->name,p);
      res->val = 0;      // начальное значение равно 0
    }
    return res->val;
        }
```

Для заданного параметра (строки) `value()` находит объект, представляющий целое (а не просто значение соответствующего целого) и возвращает ссылку на него. Эти функции можно использовать, например, так:

```
  const int MAX = 256;    // больше длины самого длинного слова

  main()
  // подсчитывает частоту слов во входном потоке
      {
    char buf[MAX];

    while (cin>>buf) value(buf)++;
```

```
  for (int i=0; vec[i].name; i++)
      cout << vec[i].name << ": " << vec [i].val<< '\n';
  }
```

В цикле `while` из стандартного входного потока `cin` читается по одному слову и записывается в буфер `buf` (см. лекцию 10), при этом каждый раз значение счетчика, связанного со считываемой строкой, увеличивается. Счетчик отыскивается в ассоциативном массиве `vec` с помощью функции `find()`. В цикле `for` печатается получившаяся таблица различных слов из `cin` вместе с их частотой. Имея входной поток

```
aa bb bb aa aa bb aa aa
```

программа выдает:

```
aa: 5
bb: 3
```

С помощью шаблонного класса и перегруженной операции `[ ]` достаточно просто довести массив из этого примера до настоящего ассоциативного массива.

2.4 Литералы

В C++ можно задавать значения всех основных типов: символьные константы, целые константы и константы с плавающей точкой. Кроме того, нуль (0) можно использовать как значение указателя произвольного типа, а символьные строки являются константами типа `char[]`. Есть возможность определить символические константы. Символическая константа - это имя, значение которого в его области видимости изменять нельзя. В C++ символические константы можно задать тремя способами:

1. добавив служебное слово `const` в определении, можно связать с именем любое значение произвольного типа;
2. множество целых констант можно определить как перечисление;
3. константой является имя массива или функции.

2.4.1 Целые константы

Целые константы могут появляться в четырех обличьях: десятичные, восьмеричные, шестнадцатеричные и символьные константы. Десятичные константы используются чаще всего и выглядят естественно:

```
0   1234   976   12345678901234567890
```

Десятичная константа имеет тип `int`, если она умещается в память, отводимую для `int`, в противном случае ее тип `long`. Транслятор должен предупреждать о константах, величина которых превышает выбранный формат представления чисел. Константа, начинающаяся с нуля, за которым следует `x` (`0x`), является шестнадцатеричным числом (с основанием 16), а константа, которая начинается с нуля, за которым следует цифра, является восьмеричным числом (с основанием 8). Приведем примеры восьмеричных констант:

```
0      02      077      0123
```

Их десятичные эквиваленты равны соответственно: 0, 2, 63, 83. В шестнадцатеричной записи эти константы выглядят так:

```
0x0    0x2    0x3f    0x53
```

Буквы a, b, c, d, e и f или эквивалентные им заглавные буквы используются для представления чисел 10, 11, 12, 13, 14 и 15, соответственно. Восьмеричная и шестнадцатеричная формы записи наиболее подходят для задания набора разрядов, а использование их для обычных чисел может дать неожиданный эффект. Например, на машине, в которой int представляется как 16-разрядное число в дополнительном коде, 0xffff есть отрицательное десятичное число -1. Если бы для представления целого использовалось большее число разрядов, то это было бы числом 65535.

Окончание U может использоваться для явного задания констант типа `unsigned`. Аналогично, окончание `L` явно задает константу типа `long`. Например:

```
void f(int);
void f(unsigned int);
void f(long int);

void g()
{
  f(3);     // вызов f(int)
  f(3U);    // вызов f(unsigned int)
  f(3L);    // вызов f(long int)
}
```

2.4.2 Константы с плавающей точкой

Константы с плавающей точкой имеют тип `double`. Транслятор должен предупреждать о таких константах, значение которых не укладывается в формат, выбранный для представления чисел с плавающей точкой. Приведем примеры констант с плавающей точкой:

```
1.23   .23    0.23   1.   1.0    1.2e10   1.23e-15
```

Отметим, что внутри константы с плавающей точкой не должно быть пробелов. Например, 65.43 e-21 не является константой с плавающей точкой, транслятор распознает это как четыре отдельные лексемы:

```
65.43   e   -   21
```

что вызовет синтаксическую ошибку.

Если нужна константа с плавающей точкой типа `float`, то ее можно получить, используя окончание `f`:

```
3.14159265f   2.0f   2.997925f
```

2.4.3 Символьные константы

Символьной константой является символ, заключенный в одиночные кавычки, например, 'a' или '0'. Символьные константы можно считать

константами, которые дают имена целым значениям символов из набора, принятого на машине, на которой выполняется программа. Это необязательно тот же набор символов, который есть на машине, где программа транслировалась. Таким образом, если вы запускаете программу на машине, использующей набор символов ASCII, то значение '0' равно 48, а если машина использует код EBCDIC, то оно будет равно 240. Использование символьных констант вместо их десятичного целого эквивалента повышает переносимость программ. Некоторые специальные комбинации символов, начинающиеся с обратной дробной черты, имеют стандартные названия:

Конец строки	NL(LF)	\n
Горизонтальная табуляция	HT	\t
Вертикальная табуляция	VT	\v
Возврат	BS	\b
Возврат каретки	CR	\r
Перевод формата	FF	\f
Сигнал	BEL	\a
Обратная дробная черта	\	\\
Знак вопроса	?	\?
Одиночная кавычка	'	\'
Двойная кавычка	"	\"
Нулевой символ	NUL	\0
Восьмеричное число	ooo	\ooo
Шестнадцатеричное число	hhh	\xhhh

Несмотря на их вид, все эти комбинации задают один символ. Тип символьной константы - `char`. Можно также задавать символ с помощью восьмеричного числа, представленного одной, двумя или тремя восьмеричными цифрами (перед цифрами идет \) или с помощью шестнадцатеричного числа (перед шестнадцатеричными цифрами идет `\x`). Число шестнадцатеричных цифр в такой последовательности неограничено. Последовательность восьмеричных или шестнадцатеричных цифр завершается первым символом, не являющимся такой цифрой. Приведем примеры:

'\6'	'\x6'	6	ASCII ack
'\60'	'\x30'	48	ASCII '0'
'\137'	'\x05f'	95	ASCII '_'

Этим способом можно представить любой символ из набора символов машины. В частности, задаваемые таким образом символы можно включать в символьные строки (см. следующий раздел). Заметим, что если для символов используется числовая форма задания, то нарушается переносимость программы между машинами с различными наборами символов.

2.4.4 Строки

Строка - это последовательность символов, заключенная в двойные кавычки:

```
"это строка"
```

Каждая строка содержит на один символ больше, чем явно задано: все строки оканчиваются нулевым символом ('\0'), имеющим значение 0. Поэтому

```
sizeof("asdf")==5;
```

Типом строки считается "массив из соответствующего числа символов", поэтому тип "asdf" есть `char[5]`. Пустая строка записывается как "" и имеет тип `char[1]`. Отметим, что для любой строки `s` выполняется `strlen(s)==sizeof(s)-1`, поскольку функция `strlen()` не учитывает завершающий символ '\0'.

Внутри строки можно использовать для представления невидимых символов специальные комбинации `с` \. В частности, в строке можно задать сам символ двойной кавычки " или символ \. Чаще всего из таких символов оказывается нужным символ конца строки '\n', например:

```
cout << "звуковой сигнал в конце сообщения\007\n"
```

Здесь 7 - это значение в ASCII символа BEL (сигнал), который в переносимом виде обозначается как \a.

Нет возможности задать в строке "настоящий" символ конца строки:

```
"это не строка,
```

```
а синтаксическая ошибка"
```

Для большей наглядности программы длинные строки можно разбивать пробелами, например:

```
char alpha[] = "abcdefghijklmnopqrstuvwxyz"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Подобные, подряд идущие, строки будут объединяться в одну, поэтому массив `alpha` можно эквивалентным образом инициализировать с помощью одной строки:

```
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

В строке можно задавать символ '\0', но большинство программ не ожидает после него встречи с какими-либо еще символами. Например, строку "asdf\000hjkl" стандартные функции `strcpy()` и `strlen()` будут рассматривать как строку "asdf".

Если вы задаете в строке последовательностью восьмеричных цифр числовую константу, то разумно указать все три цифры. Запись этой строки и так не слишком проста, чтобы еще и раздумывать, относится ли цифра к числу или является отдельным символом. Для шестнадцатеричных констант используйте два разряда. Рассмотрим следующие примеры:

```
char v1[] = "a\x0fah\0129";    // 'a' '\xfa' 'h' '\12' '9'
char v2[] = "a\xfah\129";      // 'a' '\xfa' 'h' '\12' '9'
char v3[] = "a\xfad\127";      // 'a' '\xfad' '\127'
```

2.4.5 Нуль

Нуль (0) имеет тип `int`. Благодаря стандартным преобразованиям 0 можно использовать как константу целого типа, или типа с плавающей точкой, или типа указателя. Нельзя разместить никакой объект, если вместо адреса указан 0. Какой из типов нуля использовать, определяется контекстом. Обычно (но необязательно) нуль представляется последовательностью разрядов "все нули" подходящей длины.

2.5 Поименованные константы

Добавив к описанию объекта служебное слово const, можно превратить этот объект из переменной в константу, например:

```
const int model = 90;
const int v[] = { 1, 2, 3, 4 };
```

Поскольку константе нельзя ничего присвоить, она должна быть инициализирована. Описывая какой-либо объект как const, мы гарантируем, что его значение не изменяется в области видимости:

```
model = 200;        // ошибка
model++;            // ошибка
```

Отметим, что спецификация `const` скорее ограничивает возможности использования объекта, чем указывает, где следует размещать объект. Может быть вполне разумным и даже полезным описание функции с типом возвращаемого значения `const`:

```
const char* peek(int i)    // вернуть указатель на строку-константу
{
  return hidden[i];
}
```

Приведенную функцию можно было бы использовать для передачи строки, защищенной от записи, в другую программу, где она будет читаться. Вообще говоря, транслятор может воспользоваться тем фактом, что объект является `const`, для различных целей (конечно, это зависит от "разумности" транслятора). Самое очевидное - это то, что для константы не нужно отводить память, поскольку ее значение известно транслятору. Далее, инициализатор для константы, как правило (но не всегда) является постоянным выражением, которое можно вычислить на этапе трансляции. Однако, для массива констант обычно приходится отводить память, поскольку в общем случае транслятор не знает, какой элемент массива используется в выражении. Но и в этом случае на многих машинах возможна оптимизация, если поместить такой массив в защищенную от записи память.

Задавая указатель, мы имеем дело с двумя объектами: с самим указателем и с указуемым объектом. Если в описании указателя есть "префикс" `const`, то константой объявляется сам объект, но не указатель на него, например:

```
const char* pc = "asdf";    // указатель на константу
pc[3] = 'a';                // ошибка
pc = "ghjk";                // нормально
```

Чтобы описать как константу сам указатель, а не указуемый объект, нужно использовать операцию * перед `const`. Например:

```
char *const cp = "asdf";    // указатель-константа
cp[3] = 'a';                // нормально
cp = "ghjk";                // ошибка
```

Чтобы сделать константами и указатель, и объект, надо оба объявить const, например:

```
const char *const cpc = "asdf";  // указатель-константа на const
cpc[3] = 'a';                    // ошибка
cpc = "ghjk";                    // ошибка
```

Объект может быть объявлен константой при обращении к нему с помощью указателя, и в то же время быть изменяемым, если обращаться к нему другим способом. Особенно это удобно использовать для параметров функции. Описав параметр-указатель функции как `const`, мы запрещаем изменять в ней указуемый объект, например:

```
char* strcpy(char* p, const char* q); // не может изменять *q
```

Указателю на константу можно присвоить адрес переменной, т.к. это не принесет вреда. Однако, адрес константы нельзя присваивать указателю без спецификации `const`, иначе станет возможным менять ее значение, например:

```
int a = 1;
const int c = 2;
const int* p1 = &c;    // нормально
const int* p2 = &a;    // нормально
```

```
int* p3 = &c;          // ошибка
*p3 = 7;               // меняет значение c
```

2.5.1. Перечисления

Есть способ связывания имен с целыми константами, который часто более удобен, чем описание с `const`. Например:

```
enum { ASM, AUTO, BREAK };
```

Здесь определены три целых константы, которые называются элементами перечисления, и им присвоены значения. Поскольку по умолчанию значения элементов перечисления начинаются с 0 и идут в возрастающем порядке, то приведенное перечисление эквивалентно определениям:

```
const ASM = 0;
const AUTO = 1;
const BREAK = 2;
```

Перечисление может иметь имя, например:

```
enum keyword { ASM, AUTO, BREAK };
```

Имя перечисления становится новым типом. С помощью стандартных преобразований тип перечисления может неявно приводиться к типу `int`. Обратное преобразование (из типа int в перечисление) должно быть задано явно. Например:

```
void f()
  {
    keyword k = ASM;
     int i = ASM;
        k = i     // ошибка
        k = keyword(i);
        i = k;
         k = 4;    // ошибка
  }
```

Последнее преобразование поясняет, почему нет неявного преобразования из `int` в перечисление: большинство значений типа `int` не имеет представления в данном перечислении. Описав переменную с типом `keyword` вместо очевидного `int`, мы дали как пользователю, так и транслятору определенную информацию о том, как будет использоваться эта переменная. Например, для следующего оператора

```
keyword key;

switch (key) {
case ASM:
  // выполнить что-либо
break;
case BREAK:
  // выполнить что-либо
 break;
}
```

транслятор может выдать предупреждение, поскольку из трех возможных значений типа `keyword` используются только два.

Значения элементов перечисления можно задавать и явно. Например:

```
enum int16 {
    sign=0100000,
    most_significant=040000,
    least_significant=1
};
```

Задаваемые значения необязательно должны быть различными, положительными и идти в возрастающем порядке.

2.6. Экономия памяти

В процессе создания нетривиальной программы рано или поздно наступает момент, когда требуется больше памяти, чем можно выделить или запросить. Есть два способа выжать еще некоторое количество памяти:

1. паковать в байты переменные с малыми значениями;
2. использовать одну и ту же память для хранения разных объектов в разное время.

Первый способ реализуется с помощью полей, а второй - с помощью объединений. И те, и другие описываются ниже. Поскольку назначение этих конструкций связано в основном с оптимизацией программы, и поскольку, как правило, они непереносимы, программисту следует хорошенько подумать, прежде чем использовать их. Часто лучше изменить алгоритм работы с данными, например, больше использовать динамически выделяемую память, чем заранее отведенную статическую память.

2.6.1 Поля

Кажется расточительным использовать для признака, принимающего только два значения (например: да, нет) тип `char`, но объект типа char является в C++ наименьшим объектом, который может независимо размещаться в памяти. Однако, есть возможность собрать переменные с малым диапазоном значений воедино, определив их как поля структуры. Член структуры является полем, если в его определении после имени указано число разрядов, которое он должен занимать. Допустимы безымянные поля. Они не влияют на работу с поименованными полями, но могут улучшить размещение полей в памяти для конкретной машины:

```
struct sreg {
        unsigned enable : 1;
        unsigned page : 3;
        unsigned : 1;       // не используется
        unsigned mode : 2;
        unsigned : 4;       // не используется
        unsigned access : 1;
        unsigned length : 1;
        unsigned non_resident : 1;
};
```

Приведенная структура описывает разряды нулевого регистра состояния

DEC PDP11/45 (предполагается, что поля в слове размещаются слева направо). Этот пример показывает также другое возможное применение полей: давать имена тем частям объекта, размещение которых определено извне. Поле должно иметь целый тип, и оно используется аналогично другим объектам целого типа. Но есть исключение: нельзя брать адрес поля. В ядре операционной системы или в отладчике тип `sreg` мог бы использоваться следующим образом:

```
sreg* sr0 = (sreg*)0777572;
//...
if (sr0->access) {        // нарушение прав доступа
    // разобраться в ситуации
   sr0->access = 0;
}
```

Тем не менее, применяя поля для упаковки нескольких переменных в один байт, мы необязательно сэкономим память. Экономится память для данных, но на большинстве машин одновременно возрастает объем команд, нужных для работы с упакованными данными. Известны даже такие программы, которые значительно сокращались в объеме, если двоичные переменные, задаваемые полями, преобразовывались в переменные типа `char`! Кроме того, доступ к `char` или `int` обычно происходит намного быстрее, чем доступ к полю. Поля - это просто удобная краткая форма задания логических операций для извлечения или занесения информации в части слова.

2.6.2. Объединения

Рассмотрим таблицу имен, в которой каждый элемент содержит имя и его значение. Значение может задаваться либо строкой, либо целым числом:

```
struct entry {
    char* name;
    char  type;
    char* string_value;  // используется если type == 's'
    int   int_value;     // используется если type == 'i'
};
```

```
void print_entry(entry* p)
{
 switch(p->type) {
 case 's':
     cout << p->string_value;
     break;
 case 'i':
     cout << p->int_value;
     break;
 default:
     cerr << "type corrupted\n";
     break;
 }
}
```

Поскольку переменные `string_value` и `int_value` никогда не могут использоваться одновременно, очевидно, что часть памяти пропадает впустую. Это можно легко исправить, описав обе переменные как члены объединения, например, так:

```
struct entry {
char* name;
char  type;
union {
  char* string_value; // используется если type == 's'
  int   int_value;    // используется если type == 'i'
};
   };
```

Теперь гарантируется, что при выделении памяти для `entry` члены `string_value` и `int_value` будут размещаться с одного адреса, и при этом не нужно менять все части программы, работающие с `entry`. Из этого следует, что все члены объединения вместе занимают такой же объем памяти, какой занимает наибольший член объединения.

Надежный способ работы с объединением заключается в том, чтобы выбирать значение с помощью того же самого члена, который его записывал. Однако, в больших программах трудно гарантировать, что объединение используется только таким способом, а в результате

использования не того члена объединения могут возникать трудно обнаруживаемые ошибки. Но можно встроить объединение в такую структуру, которая обеспечит правильную связь между значением поля типа и текущим типом члена объединения.

Иногда объединения используют для "псевдопреобразований" типа (в основном на это идут программисты, привыкшие к языкам, в которых нет средств преобразования типов, и в результате приходится обманывать транслятор). Приведем пример такого "преобразования" int в int* на машине VAX, которое достигается простым совпадением разрядов:

```
struct fudge {
    union {
      int  i;
      int* p;
    };
};

fudge a;
a.i = 4095;
int* p = a.p;    // некорректное использование
```

В действительности это вовсе не преобразование типа, т.к. на одних машинах `int` и `int*` занимают разный объем памяти, а на других целое не может размещаться по адресу, задаваемому нечетным числом. Такое использование объединений не является переносимым, тогда как существует переносимый способ задания явного преобразования типа.

Иногда объединения используют специально, чтобы избежать преобразования типов. Например, можно использовать `fudge`, чтобы узнать, как представляется указатель 0:

```
fudge.p = 0;
int i = fudge.i;    // i необязательно должно быть 0
```

Объединению можно дать имя, то есть можно сделать его полноправным типом. Например, `fudge` можно описать так:

```
union fudge {
```

```
    int i;
    int* p;
};
```

и использовать (некорректно) точно так же, как и раньше. Вместе с тем, поименованные объединения можно использовать и вполне корректным и оправданным способом.

Выражения и операторы

C++ имеет сравнительно небольшой набор операторов, который позволяет создавать гибкие структуры управления, и богатый набор операций для работы с данными. Основные их возможности показаны в этой лекции на одном завершенном примере. Затем приводится сводка выражений, и подробно обсуждаются операции преобразования типа и размещение в свободной памяти. Далее дана сводка операторов, а в конце лекции обсуждается выделение текста пробелами и использование комментариев.

3.1 Калькулятор

Мы познакомимся с выражениями и операторами на примере программы калькулятора. Калькулятор реализует четыре основных арифметических действия в виде инфиксных операций над числами с плавающей точкой. В качестве упражнения предлагается добавить к калькулятору переменные. Допустим, входной поток имеет вид:

```
r=2.5
area=pi*r*r
```

(здесь `pi` имеет предопределенное значение). Тогда программа калькулятора выдаст:

```
2.5
19.635
```

Результат вычислений для первой входной строки равен 2.5, а результат для второй строки - это 19.635.

Программа калькулятора состоит из четырех основных частей: анализатора, функции ввода, таблицы имен и драйвера. По сути - это транслятор в миниатюре, в котором анализатор проводит синтаксический анализ, функция ввода обрабатывает входные данные и проводит лексический анализ, таблица имен хранит постоянную информацию, нужную для работы, а драйвер выполняет инициализацию, вывод результатов и обработку ошибок. К такому калькулятору можно добавить много других полезных возможностей, но

программа его и так достаточно велика (200 строк), а введение новых возможностей только увеличит ее объем, не давая дополнительной информации для изучения C++.

3.1.1 Анализатор

Грамматика языка калькулятора определяется следующими правилами:

```
программа:
END // END - это конец ввода
список-выражений END

    список-выражений:
выражение PRINT// PRINT - это '\n' или ';'
выражение PRINT список-выражений

    выражение:
выражение + терм
выражение - терм
терм

    терм:
терм / первичное
терм * первичное
первичное

    первичное:
NUMBER// число с плавающей запятой в C++
NAME// имя в языке C++ за исключением '_'
NAME = выражение
- первичное
( выражение )
```

Иными словами, программа есть последовательность строк, а каждая строка содержит одно или несколько выражений, разделенных точкой с запятой. Основные элементы выражения - это числа, имена и операции *, /, +, - (унарный и бинарный минус) и =. Имена необязательно описывать до использования.

Для синтаксического анализа используется метод, обычно называемый рекурсивным спуском. Это распространенный и достаточно очевидный метод. В таких языках как C++, то есть в которых операция вызова не сопряжена с большими накладными расходами, этот метод эффективен. Для каждого правила грамматики имеется своя функция, которая вызывает другие функции. Терминальные символы (например, END, NUMBER, + и -) распознаются лексическим анализатором `get_token()`. Нетерминальные символы распознаются функциями синтаксического анализатора `expr()`, `term()` и `prim()`. Как только оба операнда выражения или подвыражения стали известны, оно вычисляется. В настоящем трансляторе в этот момент создаются команды, вычисляющие выражение. Анализатор использует для ввода функцию `get_token()`. Значение последнего вызова `get_token()` хранится в глобальной переменной `curr_tok`. Переменная `curr_tok` принимает значения элементов перечисления `token_value`:

```
enum token_value {
    NAME,NUMBER,    END,
    PLUS='+', MINUS='-', MUL='*',   DIV='/',
    PRINT=';', ASSIGN='=', LP='(',    RP=')'
  };
  token_value curr_tok;
```

Для всех функций анализатора предполагается, что `get_token()` уже была вызвана, и поэтому в `curr_tok` хранится следующая лексема, подлежащая анализу. Это позволяет анализатору заглядывать на одну лексему вперед. Каждая функция анализатора всегда читает на одну лексему больше, чем нужно для распознавания того правила, для которого она вызывалась. Каждая функция анализатора вычисляет "свое" выражение и возвращает его результат. Функция `expr()` обрабатывает сложение и вычитание. Она состоит из одного цикла, в котором распознанные термы складываются или вычитаются:

```
double expr()// складывает  и вычитает
{
  double left = term();

  for(;;)  // ``вечно"
```

```
 switch(curr_tok) {
 case PLUS:
get_token();// случай '+'
left += term();
break;
 case MINUS:
get_token();// случай '-'
left -= term();
break;
default:
return left;
}
}
```

Сама по себе эта функция делает немного. Как принято в высокоуровневых функциях больших программ, она выполняет задание, вызывая другие функции. Отметим, что выражения вида 2-3+4 вычисляются как (2-3)+4, что предопределяется правилами грамматики. Непривычная запись `for(;;)` - это стандартный способ задания бесконечного цикла, и его можно обозначить словом "вечно". Это вырожденная форма оператора `for`, и альтернативой ей может служить оператор `while(1)`. Оператор `switch` выполняется повторно до тех пор, пока не перестанут появляться операции + или - , а тогда по умолчанию выполняется оператор `return` (default).

Операции += и -= используются для выполнения операций сложения и вычитания. Можно написать эквивалентные присваивания: `left=left+term()` и `left=left-term()`. Однако вариант `left+=term()` и `left-=term()` не только короче, но и более четко определяет требуемое действие. Для бинарной операции `@` выражение `x@=y` означает `x=x@y`, за исключением того, что `x` вычисляется только один раз. Это применимо к бинарным операциям:

```
+   -   *   /   %   &   |   ^   <<   >>
```

поэтому возможны следующие операции присваивания:

```
+=   -=   *=   /=   %=   &=   |=   ^=   <<=   >>=
```

Каждая операция является отдельной лексемой, поэтому a + =1 содержит синтаксическую ошибку (из-за пробела между + и =). Расшифровка операций следующая: % - взятие остатка, &, | и ^ - разрядные логические операции И, ИЛИ и Исключающее ИЛИ; << и >> сдвиг влево и сдвиг вправо. Функции `term()` и `get_token()` должны быть описаны до определения `expr()`. В лекции 4 рассматривается построение программы в виде совокупности файлов. За одним исключением, все программы калькулятора можно составить так, чтобы в них все объекты описывались только один раз и до их использования. Исключением является функция `expr()`, которая вызывает функцию `term()`, а она, в свою очередь, вызывает `prim()`, и уже та, наконец, вызывает `expr()`. Этот цикл необходимо как-то разорвать, для чего вполне подходит заданное до определения `prim()` описание:

```
double expr();    // это описание необходимо
```

Функция `term()` справляется с умножением и делением аналогично тому, как функция `expr()` со сложением и вычитанием:

```
double term() // умножает и делит
    {
 double left = prim();

 for(;;)
  switch(curr_tok)    {
  case MUL:
   get_token(); // случай '*'
   left *= prim();
   break;
  case DIV:
   get_token(); // случай '/'
   double d = prim();
   if (d == 0) return error("деление на 0");
   left /= d;
   break;
  default:
   return left;
  }
```

```
}
```

Проверка отсутствия деления на нуль необходима, поскольку результат деления на нуль неопределен и, как правило, приводит к катастрофе.

Функция `error()` будет рассмотрена позже. Переменная d появляется в программе там, где она действительно нужна, и сразу же инициализируется. Во многих языках описание может находиться только в начале блока. Но такое ограничение может искажать естественную структуру программы и способствовать появлению ошибок. Чаще всего не инициализированные локальные переменные свидетельствуют о плохом стиле программирования. Исключение составляют те переменные, которые инициализируются операторами ввода, и переменные типа массива или структуры, для которых нет традиционной инициализации с помощью одиночных присваиваний. Следует напомнить, что = является операцией присваивания, тогда как == есть операция сравнения.

Функция `prim`, обрабатывающая первичное, во многом похожа на функции `expr` и `term()`. Но раз мы дошли до низа в иерархии вызовов, то в ней кое-что придется сделать. Цикл для нее не нужен:

```
double number_value;
char name_string[256];

double prim()    // обрабатывает первичное
{
 switch (curr_tok) {
 case NUMBER:  // константа с плавающей точкой
get_token();
return number_value;
 case NAME:
if (get_token() == ASSIGN) {
   name* n = insert(name_string);
   get_token();
   n->value = expr();
   return n->value;
}
return look(name_string)->value;
```

```
  case MINUS:   // унарный минус
  get_token();
  return -prim();
  case LP:
  get_token();
  double e = expr();
  if (curr_tok != RP) return error("требуется )");
  get_token();
  return e;
 case END:
  return 1;
  default:
  return error("требуется первичное");
  }
    }
```

Когда появляется NUMBER (то есть константа с плавающей точкой), возвращается ее значение. Функция ввода `get_token()` помещает значение константы в глобальную переменную `number_value`. Если в программе используются глобальные переменные, то часто это указывает на то, что структура не до конца проработана, и поэтому требуется некоторая оптимизация. Именно так обстоит дело в данном случае. В идеале лексема должна состоять из двух частей: значения, определяющего вид лексемы (в данной программе это `token_value`), и (если необходимо) собственно значения лексемы. Здесь же имеется только одна простая переменная `curr_tok`, поэтому для хранения последнего прочитанного значения NUMBER требуется глобальная переменная `number_value`. Такое решение проходит потому, что калькулятор во всех вычислениях вначале выбирает одно число, а затем считывает другое из входного потока.

Если последнее значение NUMBER хранится в глобальной переменной `number_value`, то строковое представление последнего значения NAME хранится в `name_string`. Перед тем, как что-либо делать с именем, калькулятор должен заглянуть вперед, чтобы выяснить, будет ли ему присваиваться значение, или же будет только использоваться существующее его значение. В обоих случаях надо обратиться к таблице имен. Она состоит из записей, имеющих вид:

```
struct name {
char* string;
name* next;
double value;
    };
```

Член `next` используется только служебными функциями, работающими с таблицей:

```
name* look(const char*);
   name* insert(const char*);
```

Обе функции возвращают указатель на ту запись `name`, которая соответствует их параметру-строке. Функция `look()` "ругается", если имя не было занесено в таблицу. Это означает, что в калькуляторе можно использовать имя без предварительного описания, но в первый раз оно может появиться только в левой части присваивания.

3.1.2 Функция ввода

Получение входных данных - часто самая запутанная часть программы. Причина кроется в том, что программа должна взаимодействовать с пользователем, то есть "мириться" с его прихотями, учитывать принятые соглашения и предусматривать кажущиеся редкими ошибки. Попытки заставить человека вести себя более удобным для машины образом, как правило, рассматриваются как неприемлемые, что справедливо. Задача ввода для функции низкого уровня состоит в последовательном считывании символов и составлении из них лексемы, с которой работают уже функции более высокого уровня. В этом примере низкоуровневый ввод делает функция `get_token()`. К счастью, написание низкоуровневой функции ввода достаточно редкая задача. В хороших системах есть стандартные функции для таких операций.

Правила ввода для калькулятора были специально выбраны несколько громоздкими для потоковых функций ввода. Незначительные изменения в определениях лексем превратили бы `get_token()` в обманчиво простую функцию.

Первая сложность состоит в том, что символ конца строки '\n' важен для калькулятора, но потоковые функции ввода воспринимают его как символ обобщенного пробела. Иначе говоря, для этих функций '\n' имеет значение только как символ, завершающий лексему. Поэтому приходится анализировать все обобщенные пробелы (пробел, табуляция и т.п.). Это делается в операторе `do`, который эквивалентен оператору `while`, за исключением того, что тело оператора `do` всегда выполняется хотя бы один раз:

```
char ch;

    do {    // пропускает пробелы за исключением '\n'
if(!cin.get(ch)) return curr_tok = END;
    } while (ch!='\n' && isspace(ch));
```

Функция `cin.get(ch)` читает один символ из стандартного входного потока в `ch`. Значение условия `if(!cin.get(ch))` - ложь, если из потока cin нельзя получить ни одного символа. Тогда возвращается лексема END, чтобы закончить работу калькулятора. Операция `!` `(NOT)` нужна потому, что в случае успешного считывания `get()` возвращает ненулевое значение.

Функция-подстановка `isspace()` из <ctype.h> проверяет, не является ли ее параметр обобщенным пробелом. Она возвращает ненулевое значение, если является, и нуль в противном случае. Проверка реализуется как обращение к таблице, поэтому для скорости лучше вызывать `isspace()`, чем проверять самому. То же можно сказать о функциях isalpha(), isdigit() и `isalnum()`, которые используются в `get_token()`.

После пропуска обобщенных пробелов следующий считанный символ определяет, какой будет начинающаяся с него лексема. Прежде, чем привести всю функцию, рассмотрим некоторые случаи отдельно. Лексемы '\n' и ';', завершающие выражение, обрабатываются следующим образом:

```
switch (ch) {
 case ';':
 case '\n':
```

```
    cin >> ws;    // пропуск обобщенного пробела
    return curr_tok=PRINT;
```

Необязательно снова пропускать пробел, но, сделав это, мы избежим повторных вызовов функции `get_token()`. Переменная `ws`, описанная в файле <stream.h>, используется только как приемник ненужных пробелов. Ошибка во входных данных, а также конец ввода не будут обнаружены до следующего вызова функции `get_token()`. Обратите внимание, как несколько меток выбора помечают одну последовательность операторов, заданную для этих вариантов. Для обоих символов ('\n' и ';') возвращается лексема PRINT, и она же помещается в `curr_tok`.

Числа обрабатываются следующим образом:

```
  case '0': case '1': case '2': case '3': case '4':
   case '5': case '6': case '7': case '8': case '9':
   case '.':
   cin.putback(ch);
   cin >> number_value;
   return curr_tok=NUMBER;
```

Размещать метки вариантов горизонтально, а не вертикально,- не самый лучший способ, поскольку такой текст труднее читать; но писать строку для каждой цифры утомительно. Поскольку оператор >> может читать константу с плавающей точкой типа `double`, программа тривиальна: прежде всего начальный символ (цифра или точка) возвращается назад в `cin`, а затем константу можно считать в `number_value`. Имя, т.е. лексема NAME, определяется как буква, за которой может идти несколько букв или цифр:

```
  if (isalpha(ch)) {
   char* p = name_string;
   *p++ = ch;
   while (cin.get(ch) && isalnum(ch)) *p++ = ch;
   cin.putback(ch);
   *p = 0;
   return curr_tok=NAME;
   }
```

Этот фрагмент программы заносит в `name_string` строку, оканчивающуюся нулевым символом. Функции `isalpha()` и `isalnum()` определены в <ctype.h>. Результат `isalnum(c)` ненулевой, если `c` - буква или цифра, и нулевой в противном случае.

Приведем, наконец, функцию ввода полностью:

```
token_value get_token()
 {
char ch;

do {    // пропускает обобщенные пробелы за исключением '\n'
   if(!cin.get(ch)) return curr_tok = END;
} while (ch!='\n' && isspace(ch));

switch (ch) {
case ';':
case '\n':
    cin >> ws;    // пропуск обобщенного пробела
    return curr_tok=PRINT;
case '*':
case '/':
case '+':
case '-':
case '(':
case ')':
case '=':
    return curr_tok=token_value(ch);
case '0': case '1': case '2': case '3': case '4':
case '5': case '6': case '7': case '8': case '9':
case '.':
    cin.putback(ch);
    cin >> number_value;
    return curr_tok=NUMBER;
  default: // NAME, NAME= или ошибка
    if (isalpha(ch)) {
char* p = name_string;
*p++ = ch;
while (cin.get(ch) && isalnum(ch)) *p++ = ch;
```

```
cin.putback(ch);
*p = 0;
return curr_tok=NAME;
    }
    error("недопустимая лексема");
    return curr_tok=PRINT;
}
}
```

Преобразование операции в значение лексемы для нее тривиально, поскольку в перечислении `token_value` лексема операции была определена как целое (код символа операции).

3.1.3 Таблица имен

Есть функция поиска в таблице имен:

```
name* look(char* p, int ins =0);
```

Второй ее параметр показывает, была ли символьная строка, обозначающая имя, ранее занесена в таблицу. Инициализатор `=0` задает стандартное значение параметра, которое используется, если функция `look()` вызывается только с одним параметром. Это удобно, так как можно писать `look("sqrt2")`, что означает `look("sqrt2",0)`, т.е. поиск, а не занесение в таблицу. Чтобы было так же удобно задавать операцию занесения в таблицу, определяется вторая функция:

```
inline name* insert(const char* s) { return look(s,1); }
```

Как ранее упоминалось, записи в этой таблице имеют такой тип:

```
struct name {
char* string;
name* next;
double value;
};
```

Член next используется для связи записей в таблице. Собственно

таблица - это просто массив указателей на объекты типа name:

```
const TBLSZ = 23;
name* table[TBLSZ];
```

Поскольку по умолчанию все статические объекты инициализируются нулем, такое тривиальное описание таблицы `table` обеспечивает также и нужную инициализацию.

Для поиска имени в таблице функция `look()` использует простой хэш-код (записи, в которых имена имеют одинаковый хэш-код, связываются вместе):

```
int ii = 0;// хэш-код
const char* pp = p;
while (*pp) ii = ii<<1 ^ *pp++;
if (ii < 0) ii = -ii;
ii %= TBLSZ;
```

Иными словами, с помощью операции `^` ("исключающее ИЛИ") все символы входной строки `p` поочередно добавляются к ii. Разряд в результате `x^y` равен 1 тогда и только тогда, когда эти разряды в операндах `x` и `y` различны. До выполнения операции `^` значение `ii` сдвигается на один разряд влево, чтобы использовался не только один байт `ii`. Эти действия можно записать таким образом:

```
ii <<= 1;
ii ^= *pp++;
```

Для хорошего хэш-кода лучше использовать операцию `^`, чем `+`. Операция сдвига важна для получения приемлемого хэш-кода в обоих случаях.

Операторы

```
if (ii < 0) ii = -ii;
ii %= TBLSZ;
```

гарантируют, что значение `ii` будет из диапазона 0...TBLSZ-1. Напомним, что % - это операция взятия остатка. Ниже полностью

приведена функция look:

```
#include <string.h>

 name* look(const char* p, int ins =0)
 {
int ii = 0;// хэш-код
const char* pp = p;
while (*pp) ii = ii<<1 ^ *pp++;
if (ii < 0) ii = -ii;
ii %= TBLSZ;

for (name* n=table[ii]; n; n=n->next)    // поиск
   if (strcmp(p,n->string) == 0) return n;

if (ins == 0) error("имя не найдено");

name* nn = new name;  // занесение
nn->string = new char[strlen(p)+1];
strcpy(nn->string,p);
nn->value = 1;
nn->next = table[ii];
table[ii] = nn;
return nn;
 }
```

После вычисления хэш-кода `ii` идет простой поиск имени по членам next. Имена сравниваются с помощью стандартной функции сравнения строк `strcmp()`. Если имя найдено, то возвращается указатель на содержащую его запись, а в противном случае заводится новая запись с этим именем.

Добавление нового имени означает создание нового объекта name в свободной памяти с помощью операции `new`, его инициализацию и включение в список имен. Последнее выполняется как занесение нового имени в начало списка, поскольку это можно сделать даже без проверки того, есть ли список вообще. Символьная строка имени также размещается в свободной памяти. Функция `strlen()` указывает, сколько памяти нужно для строки, операция new отводит нужную

память, а функция `strcpy()` копирует в нее строку. Все строковые функции описаны в <string.h>:

```
extern int strlen(const char*);
extern int strcmp(const char*, const char*);
extern char* strcpy(char*, const char*);
```

3.1.4 Обработка ошибок

Поскольку программа достаточно проста, не надо особо беспокоиться об обработке ошибок. Функция error просто подсчитывает число ошибок, выдает сообщение о них и возвращает управление обратно:

```
int no_of_errors;

double error(const char* s)
{
cerr << "error: " << s << "\n";
no_of_errors++;
return 1;
}
```

Небуферизованный выходной поток cerr обычно используется именно для выдачи сообщений об ошибках. Управление возвращается из `error()` потому, что ошибки, как правило, встречаются посреди вычисления выражения. Значит надо либо полностью прекращать вычисления, либо возвращать значение, которое не должно вызвать последующих ошибок. Для простого калькулятора больше подходит последнее. Если бы функция `get_token()` отслеживала номера строк, то функция `error()` могла бы указывать пользователю приблизительное место ошибки. Это было бы полезно при неинтерактивной работе с калькулятором. Часто после появления ошибки программа должна завершиться, поскольку не удалось предложить разумный вариант ее дальнейшего выполнения. Завершить ее можно с помощью вызова функции `exit()`, которая заканчивает работу с выходными потоками и завершает программу, возвращая свой параметр в качестве ее результата. Более радикальный способ завершения программы - это вызов функции `abort()`, которая

прерывает выполнение программы немедленно или сразу же после сохранения информации для отладчика (сброс оперативной памяти).

Более тонкие приемы обработки ошибок можно предложить, если ориентироваться на особые ситуации, но предложенное решение вполне приемлемо для игрушечного калькулятора в 200 строк.

3.1.5 Драйвер

Когда все части программы определены, нужен только драйвер, чтобы инициализировать и запустить процесс. В нашем примере с этим справится функция `main()`:

```
int main()
 {
 // вставить предопределенные имена:
 insert("pi")->value = 3.1415926535897932385;
 insert("e")->value = 2.7182818284590452354;

 while (cin) {
  get_token();
  if (curr_tok == END) break;
  if (curr_tok == PRINT) continue;
  cout << expr() << '\n';
 }
 return no_of_errors;
 }
```

Принято, что функция `main()` возвращает нуль, если программа завершается нормально, и ненулевое значение, если происходит иначе. Ненулевое значение возвращается как число ошибок. Оказывается, вся инициализация сводится к занесению предопределенных имен в таблицу.

В цикле main читаются выражения и выдаются результаты. Это делает одна строка:

```
cout << expr() << '\n';
```

Проверка `cin` при каждом проходе цикла гарантирует завершение программы, даже если что-то случится с входным потоком, а проверка на лексему END нужна для нормального завершения цикла, когда функция `get_token()` обнаружит конец файла. Оператор `break` служит для выхода из ближайшего объемлющего оператора `switch` или цикла (т.е. оператора `for`, `while` или `do`). Проверка на лексему PRINT (т.е. на '\n' и ';') снимает с функции `expr()` обязанность обрабатывать пустые выражения. Оператор `continue` эквивалентен переходу на конец цикла, поэтому в нашем случае фрагмент:

```
while (cin) {
// ...
if (curr_tok == PRINT) continue;
cout << expr() << '\n';
}
```

эквивалентен фрагменту:

```
while (cin) {
// ...
if (curr_tok == PRINT) goto end_of_loop;
cout << expr() << '\n';
end_of_loop: ;
}
```

3.1.6 Параметры командной строки

Когда программа калькулятора уже была написана и отлажена, выяснилось, что неудобно вначале запускать ее, вводить выражение, а затем выходить из калькулятора. Тем более, что обычно нужно просто вычислить одно выражение. Если это выражение задать как параметр командной строки запуска калькулятора, то можно сэкономить несколько нажатий клавиши. Как уже было сказано, выполнение программы начинается вызовом `main()`. При этом вызове `main()` получает два параметра: число параметров (обычно называемый `argc`) и массив строк параметров (обычно называемый `argv`). Параметры - это символьные строки, поэтому `argv` имеет тип `char*[argc+1]`. Имя программы (в том виде, как оно было задано в командной строке)

передается в `argv[0]`, поэтому `argc` всегда не меньше единицы. Например, для командной строки

```
dc 150/1.1934
```

параметры имеют значения:

```
argc    2
 argv[0]    "dc"
 argv[1]    "150/1.1934"
 argv[2]    0
```

Добраться до параметров командной строки просто; проблема в том, как использовать их так, чтобы не менять саму программу. В данном случае это оказывается совсем просто, поскольку входной поток может быть настроен на символьную строку вместо файла. Например, можно определить cin так, чтобы символы читались из строки, а не из стандартного входного потока:

```
int main(int argc, char* argv[])
 {
switch(argc) {
 case 1:   // считывать из стандартного входного потока
    break;
 case 2:   // считывать из строки параметров
  cin = *new istream(argv[1],strlen(argv[1]));
  break;
 default:
  error("слишком много параметров");
  return 1;
 }

// дальше прежний вариант main
 }
```

При этом `istrstream` - это функция `istream`, которая считывает символы из строки, являющейся ее первым параметром. Чтобы использовать `istrstream` нужно включить в программу файл <strstream.h>, а не обычный <iostream.h>. В остальном же программа осталась без изменений, кроме добавления параметров в функцию

`main()` и использования их в операторе `switch`. Можно легко изменить функцию `main()` так, чтобы она могла принимать несколько параметров из командной строки. Однако это не слишком нужно, тем более, что можно нескольких выражений передать как один параметр:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

Кавычки необходимы потому, что символ ';' служит в системе UNIX разделителем команд. В других системах могут быть свои соглашения о параметрах командной строки.

3.2 Сводка операций

Здесь приводится краткая сводка операций и несколько примеров. Каждая операция сопровождается одним или несколькими характерными для нее именами и примером ее использования. В этих примерах `class_name` обозначает имя класса, `member` - имя члена, `object` - выражение, задающее объект класса, `pointer` - выражение, задающее указатель, expr - просто выражение, а *`lvalue`* (адрес) - выражение, обозначающее не являющийся константой объект. Обозначение (type) задает имя типа в общем виде (с возможным добавлением *, () и т.д.). Если оно указано без скобок, существуют ограничения.

Порядок применения унарных операций и операций присваивания "справа налево", а всех остальных операций - "слева направо". То есть, `a=b=c` означает `a=(b=c)`, `a+b+c` означает `(a+b)+c`, и `*p++` означает `*(p++)`, а не `(*p)++`.

Операции C++
:: Разрешение области видимости class_name :: member
:: Глобальное :: name
.Выбор члена object . member
-> Выбор членаpointer -> member
[] Индексирование pointer [expr]
() Вызов функции expr (expr_list)

() Структурное значениеtype (expr_list)
sizeof Размер объекта sizeof expr
sizeof Размер типа sizeof (type)

++ Постфиксный инкремент lvalue ++
++ Префиксный инкремент++ lvalue
-- Постфиксный декремент lvalue --
-- Префиксный декремент-- lvalue
~Дополнение ~ expr
!Логическое НЕ ! expr
-Унарный минус - expr
+Унарный плюс+ expr
&Взятие адреса & lvalue
*Косвенность * expr
newСоздание (размещение) new type
delete Уничтожение (освобождение) delete pointer
delete[] Уничтожение массива delete[] pointer
() Приведение(преобразование)типа (type) expr

. *Выбор члена косвенный object . pointer-to-member
->*Выбор члена косвенный pointer -> pointer-to-member

---

*Умножениеexpr * expr
/Деление expr / expr
%Остаток от деления expr % expr

+Сложение (плюс) expr + expr
-Вычитание (минус) expr - expr

Все операции таблицы, находящиеся между двумя ближайшими друг к другу горизонтальными чертами, имеют одинаковый приоритет. Приоритет операций уменьшается при движении "сверху вниз". Например, `a+b*c` означает `a+(b*c)`, так как * имеет приоритет выше, чем +; а выражение `a+b-c` означает `(a+b)-c`, поскольку + и - имеют одинаковый приоритет, и операции + и - применяются "слева направо".

Операции C++ (продолжение)

==========

<< Сдвиг влево expr << expr
>> Сдвиг вправо expr >> expr

<Меньше expr < expr
<= Меньше или равно expr <= expr
>Больше expr > expr
>= Больше или равно expr >= expr

== Равно expr == expr
!= Не равно expr != expr

&Поразрядное Иexpr & expr

---

^Поразрядное исключающее ИЛИ expr ^ expr

|Поразрядное включающее ИЛИexpr | expr

&& Логическое И expr && expr

|| Логическое ИЛИ expr || expr

? :Операция условия expr? expr : expr

=Простое присваивание lvalue = expr
*= Присваивание с умножением lvalue *= expr
/= Присваивание с делением lvalue /= expr
%= Присваивание с взятием lvalue %= expr
остатка от деления
+= Присваивание со сложением lvalue += expr
-= Присваивание с вычитанием lvalue -= expr
<<=Присваивание со сдвигом влево lvalue <<= expr
>>=Присваивание со сдвигом вправо lvalue >>= expr
&= Присваивание с поразрядным И lvalue &= expr
|= Присваивание с поразряднымlvalue |= expr
включающим ИЛИ
^= Присваивание с поразряднымlvalue ^= expr
исключающим ИЛИ

Запятая (последовательность)	expr , expr

3.2.1 Скобки

Синтаксис языка C++ перегружен скобками, и разнообразие их применений способно сбить с толку. Они выделяют фактические параметры при вызове функций, имена типов, задающих функции, а также служат для разрешения конфликтов между операциями с одинаковым приоритетом. К счастью, последнее встречается не слишком часто, поскольку приоритеты и порядок применения операций определены так, чтобы выражения вычислялись "естественным образом" (т.е. наиболее распространенным образом). Например, выражение

```
if (i<=0 || max<i)   // ...
```

означает следующее: "Если `i` меньше или равно нулю, или если `max` меньше i". То есть, оно эквивалентно

```
if ( (i<=0) || (max<i) )    // ...
```

но не эквивалентно допустимому, хотя и бессмысленному выражению

```
if (i <= (0||max) < i)    // ...
```

Тем не менее, если программист не уверен в указанных правилах, следует использовать скобки, причем некоторые предпочитают для надежности писать более длинные и менее элегантные выражения, как:

```
if ( (i<=0) || (max<i) )   // ...
```

При усложнении подвыражений скобки используются чаще. Не надо, однако, забывать, что сложные выражения являются источником ошибок. Поэтому, если у вас появится ощущение, что в этом выражении нужны скобки, лучше разбейте его на части и введите дополнительную переменную. Бывают случаи, когда приоритеты операций не приводят к "естественному" порядку вычислений. Например, в выражении

```
if (i&mask == 0)    // ловушка! & применяется после ==
```

не происходит маскирование `i (i&mask)`, а затем проверка результата на 0. Поскольку `y ==` приоритет выше, чем у &, это выражение эквивалентно `i&(mask==0)`. В этом случае скобки играют важную роль:

```
if ((i&mask) == 0) // ...
```

Имеет смысл привести еще одно выражение, которое вычисляется совсем не так, как мог бы ожидать неискушенный пользователь:

```
if (0 <= a <= 99) // ...
```

Оно допустимо, но интерпретируется как `(0<=a)<=99`, и результат первого сравнения равен или 0, или 1, но не значению a (если, конечно, a не есть 1). Проверить, попадает ли a в диапазон 0...99, можно так:

```
if (0<=a && a<=99) // ...
```

Среди новичков распространена ошибка, когда в условии вместо == (равно) используют = (присвоить):

```
if (a = 7)   // ошибка: присваивание константы в условии
 // ...
```

Она вполне объяснима, поскольку в большинстве языков "=" означает "равно". Для транслятора не составит труда сообщать об ошибках подобного рода.

3.2.2 Порядок вычислений

Порядок вычисления подвыражений, входящих в выражение, не всегда определен. Например:

```
int i = 1;
v[i] = i++;
```

Здесь выражение может вычисляться или как `v[1]=1`, или как

`v[2]=1`. Если нет ограничений на порядок вычисления подвыражений, то транслятор получает возможность создавать более оптимальный код. Транслятору следовало бы предупреждать о двусмысленных выражениях, но к сожалению большинство из них не делает этого.

Для операций

&& || ,

гарантируется, что их левый операнд вычисляется раньше правого операнда. Например, в выражении `b=(a=2,a+1)` `b` присвоится значение 3. Отметим, что операция запятая отличается по смыслу от той запятой, которая используется для разделения параметров при вызове функций. Пусть есть выражения:

```
f1(v[i],i++); // два параметра
f2( (v[i],i++) )   // один параметр
```

Вызов функции `f1` происходит с двумя параметрами: `v[i]` и `i++`, но порядок вычисления выражений параметров неопределен. Зависимость вычисления значений фактических параметров от порядка вычислений - далеко не лучший стиль программирования. К тому же программа становится непереносимой. Вызов `f2` происходит с одним параметром, являющимся выражением, содержащим операцию запятая: `(v[i], i++)`. Оно эквивалентно `i++`.

Скобки могут принудительно задать порядок вычисления. Например, `a*(b/c)` может вычисляться как `(a*b)/c` (если только пользователь видит в этом какое-то различие). Заметим, что для значений с плавающей точкой результаты вычисления выражений `a*(b/c)` и `(a*b)/c` могут различаться весьма значительно.

3.2.3 Инкремент и декремент

Операция `++` явно задает инкремент в отличие от неявного его задания с помощью сложения и присваивания. По определению `++lvalue` означает `lvalue+=1`, что, в свою очередь означает

lvalue=lvalue+1 при условии, что содержимое *lvalue* не вызывает побочных эффектов. Выражение, обозначающее операнд инкремента, вычисляется только один раз. Аналогично обозначается операция декремента (`--`). Операции `++` и `--` могут использоваться как префиксные и постфиксные операции. Значением `++x` является новое (т. е. увеличенное на 1) значение `x`. Например, `y=++x` эквивалентно `y=(x+=1)`. Напротив, значение `x++` равно прежнему значению x. Например, `y=x++` эквивалентно `y=(t=x,x+=1,t)`, где `t` - переменная того же типа, что и `x`.

Напомним, что операции инкремента и декремента указателя эквивалентны сложению 1 с указателем или вычитанию 1 из указателя, причем вычисление происходит в элементах массива, на который настроен указатель. Так, результатом `p++` будет указатель на следующий элемент. Для указателя `p` типа `T*` следующее соотношение верно по определению:

```
long(p+1) == long(p) + sizeof(T);
```

Чаще всего операции инкремента и декремента используются для изменения переменных в цикле. Например, копирование строки, оканчивающейся нулевым символом, задается следующим образом:

```
inline void cpy(char* p, const char* q)
 {
while (*p++ = *q++) ;
 }
```

Язык C++ (подобно C) имеет как сторонников, так и противников именно из-за такого сжатого, использующего сложные выражения стиля программирования. Оператор

```
while (*p++ = *q++) ;
```

вероятнее всего, покажется невразумительным для незнакомых с C. Имеет смысл повнимательнее посмотреть на такие конструкции, поскольку для C и C++ они не является редкостью.

Сначала рассмотрим более традиционный способ копирования массива символов:

```
int length = strlen(q)
 for (int i = 0; i<=length; i++) p[i] = q[i];
```

Это неэффективное решение: строка оканчивается нулем; единственный способ найти ее длину - это прочитать ее всю до нулевого символа; в результате строка читается и для установления ее длины, и для копирования, то есть дважды. Поэтому попробуем такой вариант:

```
for (int i = 0; q[i] !=0 ; i++) p[i] = q[i];
 p[i] = 0;   // запись нулевого символа
```

Поскольку `p` и `q` - указатели, можно обойтись без переменной i, используемой для индексации:

```
while (*q !=0) {
    *p = *q;
    p++;// указатель на следующий символ
    q++;// указатель на следующий символ
 }
 *p = 0; // запись нулевого символа
```

Поскольку операция постфиксного инкремента позволяет сначала использовать значение, а затем уже увеличить его, можно переписать цикл так:

```
while (*q != 0) {
     *p++ = *q++;
}
*p = 0;   // запись нулевого символа
```

Отметим, что результат выражения `*p++ = *q++` равен `*q`. Следовательно, можно переписать наш пример и так:

```
while ((*p++ = *q++) != 0)  { }
```

В этом варианте учитывается, что *q равно нулю только тогда, когда `*q` уже скопировано в `*p`, поэтому можно исключить завершающее присваивание нулевого символа. Наконец, можно еще более сократить запись этого примера, если учесть, что пустой блок не нужен, а операция "!= 0" избыточна, т.к. результат условного выражения и так

всегда сравнивается с нулем. В результате мы приходим к первоначальному варианту, который вызывал недоумение:

```
while (*p++ = *q++) ;
```

Неужели этот вариант труднее понять, чем приведенные выше? Только неопытным программистам на C++ или C! Будет ли последний вариант наиболее эффективным по затратам времени и памяти? Если не считать первого варианта с функцией `strlen()`, то это неочевидно. Какой из вариантов окажется эффективнее, определяется как спецификой системы команд, так и возможностями транслятора. Наиболее эффективный алгоритм копирования для вашей машины можно найти в стандартной функции копирования строк из файла <string.h>:

```
int strcpy(char*, const char*);
```

3.2.4 Поразрядные логические операции

Поразрядные логические операции

```
&  |  ^  ~  >>  <<
```

применяются к целым, то есть к объектам типа `char`, `short`, `int`, `long` и к их беззнаковым аналогам. Результат операции также будет целым.

Чаще всего поразрядные логические операции используются для работы с небольшим по величине множеством данных (массивом разрядов). В этом случае каждый разряд беззнакового целого представляет один элемент множества, и число элементов определяется количеством разрядов. Бинарная операция & интерпретируется как пересечение множеств, операция | как объединение, а операция ^ как разность множеств. С помощью перечисления можно задать имена элементам множества. Ниже приведен пример, заимствованный из <iostream.h>:

```
class ios {
 public:
 enum io_state {
goodbit=0, eofbit=1, failbit=2, badbit=4
```

```
};
// ...
};
```

Состояние потока можно установить следующим присваиванием:

```
cout.state = ios::goodbit;
```

Уточнение именем ios необходимо, потому что определение `io_state` находится в классе `ios`, а также чтобы не возникло коллизий, если пользователь заведет свои имена наподобие `goodbit`.

Проверку на корректность потока и успешное окончание операции можно задать так:

```
if (cout.state&(ios::badbit|ios::failbit))  // ошибка в потоке
```

Еще одни скобки необходимы потому, что операция & имеет более высокий приоритет, чем операция "|". Функция, обнаружившая конец входного потока, может сообщать об этом так:

```
cin.state |= ios::eofbit;
```

Операция |= используется потому, что в потоке уже могла быть ошибка (т.е. state==ios::badbit), и присваивание

```
cin.state =ios::eofbit;
```

могло бы затереть ее признак. Установить отличия в состоянии двух потоков можно следующим способом:

```
ios::io_state diff = cin.state^cout.state;
```

Для таких типов, как `io_state`, нахождение различий не слишком полезная операция, но для других сходных типов она может оказаться весьма полезной. Например, полезно сравнение двух разрядных массива, один из которых представляет набор всех возможных обрабатываемых прерываний, а другой - набор прерываний, ожидающих обработки.

Отметим, что использование полей может служить удобным и более

лаконичным способом работы с частями слова, чем сдвиги и маскирование. С частями слова можно работать и с помощью поразрядных логических операций. Например, можно выделить средние 16 разрядов из средины 32-разрядного целого:

```
unsigned short middle(int a) { return (a>>8)&0xffff; }
```

Только не путайте поразрядные логические операции с просто логическими операциями:

```
&&   ||   !
```

Результатом последних может быть 0 или 1, и они в основном используются в условных выражениях операторов `if`, `while` или `for`. Например, !0 (не нуль) имеет значение 1, тогда как ~0 (дополнение нуля) представляет собой набор разрядов "все единицы", который обычно является значением -1 в дополнительном коде.

3.2.5 Преобразование типа

Иногда бывает необходимо явно преобразовать значение одного типа в значение другого. Результатом явного преобразования будет значение указанного типа, полученное из значения другого типа. Например:

```
float r = float(1);
```

Здесь перед присваиванием целое значение 1 преобразуется в значение с плавающей точкой 1.0f. Результат преобразования типа не является адресом, поэтому ему присваивать нельзя (если только тип не является ссылкой).

Существуют два вида записи явного преобразования типа: традиционная запись, как операция приведения в C, например, (double)a и функциональная запись, например, `double(a)`. Функциональную запись нельзя использовать для типов, которые не имеют простого имени. Например, чтобы преобразовать значение в тип указателя, надо или использовать приведение

```
char* p = (char*)0777;
```

или определить новое имя типа:

```
typedef char* Pchar;
char* p = Pchar(0777);
```

По мнению автора, функциональная запись в нетривиальных случаях предпочтительнее. Рассмотрим два эквивалентных примера:

```
Pname n2 = Pbase(n1->tp)->b_name; // функциональная запись
 Pname n3 = ((Pbase)n2->tp)->b_name;   // запись с приведением
```

Поскольку операция -> имеет больший приоритет, чем операция приведения, последнее выражение выполняется так:

```
((Pbase)(n2->tp))->b_name
```

Используя явное преобразование в тип указателя можно выдать данный объект за объект произвольного типа. Например, присваивание

```
any_type* p = (any_type*)&some_object;
```

позволит обращаться к некоторому объекту (`some_object`) через указатель p как к объекту произвольного типа (`any_type`). Тем не менее, если `some_object` в действительности имеет тип не `any_type`, могут получиться странные и нежелательные результаты.

Если преобразование типа не является необходимым, его вообще следует избегать. Программы, в которых есть такие преобразования, обычно труднее понимать, чем программы, их не имеющие. В то же время программы с явно заданными преобразованиями типа понятнее, чем программы, которые обходятся без таких преобразований, потому что не вводят типов для представления понятий более высокого уровня. Так, например, поступают программы, управляющие регистром устройства с помощью сдвига и маскирования целых, вместо того, чтобы определить подходящую структуру (struct) и работать непосредственно с ней. Корректность явного преобразования типа часто существенно зависит от того, насколько программист понимает, как язык работает с объектами различных типов, и какова специфика данной реализации языка. Приведем пример:

```
int i = 1;
  char* pc = "asdf";
  int* pi = &i;
  i = (int)pc;
  pc = (char*)i; // осторожно: значение pc может измениться.
   // На некоторых машинах sizeof(int)
   // меньше, чем sizeof(char*)
 pi = (int*)pc;
 pc = (char*)pi; // осторожно: pc может измениться
   // На некоторых машинах char* имеет не такое
   // представление, как int*
```

Для многих машин эти присваивания ничем не грозят, но для некоторых результат может быть плачевным. В лучшем случае подобная программа будет переносимой. Обычно без особого риска можно предположить, что указатели на различные структуры имеют одинаковое представление. Далее, произвольный указатель можно присвоить (без явного преобразования типа) указателю типа `void*`, а `void*` может быть явно преобразован обратно в указатель произвольного типа.

В языке C++ явные преобразования типа оказывается излишними во многих случаях, когда в C (и других языках) они требуются. Во многих программах можно вообще обойтись без явных преобразований типа, а во многих других они могут быть локализованы в нескольких подпрограммах.

3.2.6 Свободная память

Именованный объект является либо статическим, либо автоматическим. Статический объект размещается в памяти в момент запуска программы и существует там до ее завершения. Автоматический объект размещается в памяти всякий раз, когда управление попадает в блок, содержащий определение объекта, и существует только до тех пор, пока управление остается в этом блоке. Тем не менее, часто бывает удобно создать новый объект, который существует до тех пор, пока он не станет ненужным. В частности, бывает удобно создать объект, который можно использовать после возврата из функции, где он был создан. Подобные объекты

создает операция `new`, а операция `delete` используется для их уничтожения в дальнейшем. Про объекты, созданные операцией `new`, говорят, что они размещаются в свободной памяти. Примерами таких объектов являются узлы деревьев или элементы списка, которые входят в структуры данных, размер которых на этапе трансляции неизвестен. Давайте рассмотрим в качестве примера набросок транслятора, который строится аналогично программе калькулятора. Функции синтаксического анализа создают из представлений выражений дерево, которое будет в дальнейшем использоваться для генерации кода. Например:

```
struct enode {
  token_value oper;
  enode* left;
  enode* right;
 };

enode* expr()
{
enode* left = term();

for(;;)
switch(curr_tok) {
  case PLUS:
  case MINUS:
 get_token();
 enode* n = new enode;
 n->oper = curr_tok;
 n->left = left;
 n->right = term();
 left = n;
 break;
 default:
return left;
}
}
```

Генератор кода может использовать дерево выражений, например так:

```
void generate(enode* n)
{
switch (n->oper) {
case PLUS:
  // соответствующая генерация
  delete n;
}
}
```

Объект, созданный с помощью операции `new`, существует, до тех пор, пока он не будет явно уничтожен операцией `delete`. После этого память, которую он занимал, вновь может использоваться `new`. Обычно нет никакого "сборщика мусора", ищущего объекты, на которые никто не ссылается, и предоставляющего занимаемую ими память операции `new` для повторного использования. Операндом `delete` может быть только указатель, который возвращает операция `new`, или нуль. Применение `delete` к нулю не приводит ни к каким действиям. Операция `new` может также создавать массивы объектов, например:

```
char* save_string(const char* p)
{
  char* s = new char[strlen(p)+1];
  strcpy(s,p);
  return s;
}
```

Отметим, что для перераспределения памяти, отведенной операцией `new`, операция `delete` должна уметь определять размер размещенного объекта. Например:

```
int main(int argc, char* argv[])
{
  if (argc < 2) exit(1);
  char* p = save_string(arg[1]);
  delete[] p;
}
```

Чтобы добиться этого, приходится под объект, размещаемый стандартной операцией `new`, отводить немного больше памяти, чем под

статический (обычно, больше на одно слово). Простой оператор `delete` уничтожает отдельные объекты, а операция `delete[]` используется для уничтожения массивов.

Операции со свободной памятью реализуются функциями:

```
void* operator new(size_t);
void operator delete(void*);
```

Здесь `size_t` - беззнаковый целочисленный тип, определенный в <stddef.h>.

Стандартная реализация функции operator `new()` не инициализирует предоставляемую память.

Что случится, когда операция new не сможет больше найти свободной памяти для размещения? Поскольку даже виртуальная память небесконечна, такое время от времени происходит. Так, запрос вида:

```
char* p = new char [100000000];
```

обычно не проходит нормально. Когда операция `new` не может выполнить запрос, она вызывает функцию, которая была задана как параметр при обращении к функции `set_new_handler()` из <new.h>. Например, в следующей программе:

```
#include <iostream.h>
#include <new.h>
#include <stdlib.h>

void out_of_store()
{
 cerr << "operator new failed: out of store\n";
 exit(1);
}

int main()
{
set_new_handler(&out_of_store);
char* p = new char[100000000];
```

```
cout << "done, p = " << long(p) << '\n';
}
```

скорее всего, будет напечатано не "done", а сообщение:

```
operator new failed: out of store
// операция new не прошла: нет памяти
```

С помощью функции `new_handler` можно сделать нечто более сложное, чем просто завершить программу. Если известен алгоритм операций `new` и `delete` (например, потому, что пользователь определил свои функции `operator new` и `operator delete`), то обработчик `new_handler` может попытаться найти свободную память для `new`. Другими словами, пользователь может написать свой "сборщик мусора", тем самым сделав вызов операции `delete` необязательным. Однако такая задача, безусловно, не под силу новичку.

По традиции операция new просто возвращает указатель 0, если не удалось найти достаточно свободной памяти. Реакция же на это `new_handler` не была установлена. Например, следующая программа:

```
#include <stream.h>

 main()
  {
char* p = new char[100000000];
cout << "done, p = " << long(p) << '\n';
   }
```

выдаст

```
done, p = 0
```

Память не выделена, и вам сделано предупреждение! Отметим, что, задав реакцию на такую ситуацию в функции `new_handler`, пользователь берет на себя проверку: исчерпана ли свободная память. Она должна выполняться при каждом обращении в программе к `new` (если только пользователь не определил собственные функции для размещения объектов пользовательских типов).

3.3 Сводка операторов

Здесь дается сводка операторов и несколько примеров.

```
---------------------------------------------------------------
Синтаксис операторов
---------------------------------------------------------------

  оператор:
описание
{ список-операторов opt }
выражение opt ;

if ( выражение )  оператор
if ( выражение )  оператор else оператор
switch ( выражение ) оператор

while ( выражение ) оператор
do  оператор  while ( выражение )
for (начальный-оператор-for; выражение opt; выражение opt) оператор

case  выражение-константа : оператор
default : оператор
break ;
continue ;

return  выражение opt ;

goto  идентификатор ;
идентификатор : оператор

список-операторов:
 оператор
 список-операторов  оператор

начальный-оператор-for:
 описание
 выражение opt ;
---------------------------------------------------------------
```

Обратите внимание, что описание является оператором, но нет операторов присваивания или вызова функции (они относятся к выражениям).

3.3.1 Выбирающие операторы

Значение можно проверить с помощью операторов `if` или `switch`:

```
if ( выражение ) оператор

if ( выражение ) оператор else оператор

switch ( выражение ) оператор
```

В языке C++ среди основных типов нет отдельного булевского (тип со значениями истина, ложь). Все операции отношений:

```
== != < > <= >=
```

дают в результате целое 1, если отношение выполняется, и 0 в противном случае. Обычно определяют константы TRUE как 1 и FALSE как 0.

В операторе `if`, если выражение имеет ненулевое значение, выполняется первый оператор, а иначе выполняется второй (если он указан). Таким образом, в качестве условия допускается любое выражение типа целое или указатель. Пусть a целое, тогда

```
if (a) // ...
```

эквивалентно

```
if (a != 0) ...
```

Логические операции

```
&& || !
```

обычно используются в условиях. В операциях && и || второй операнд

не вычисляется, если результат определяется значением первого операнда. Например, в выражении

```
if (p && l<p->count)  // ...
```

сначала проверяется значение `p`, и только если оно не равно нулю, то проверяется отношение `l<p->count`.

Некоторые простые операторы `if` удобно заменять выражениями условия. Например, вместо оператора

```
if (a <= b)
   max = b;
else
   max = a;
```

лучше использовать выражение

```
max = (a<=b) ? b : a;
```

Условие в выражении условия не обязательно окружать скобками, но если их использовать, то выражение становится понятнее.

Простой переключатель (switch) можно записать с помощью серии операторов `if`. Например,

```
switch (val) {
 case 1:
    f();
    break;
 case 2:
    g();
    break;
 default:
    h();
    break;
 }
```

можно эквивалентно задать так:

```
if (val == 1)
    f();
else if (val == 2)
    g();
else
    h();
```

Смысл обеих конструкций совпадает, но все же первая предпочтительнее, поскольку в ней нагляднее показана суть операции: проверка на совпадение значения `val` со значением из множества констант. Поэтому в нетривиальных случаях запись, использующая переключатель, понятнее.

Нужно позаботиться о каком-то завершении оператора, указанного в варианте переключателя, если только вы не хотите, чтобы стали выполняться операторы из следующего варианта. Например, переключатель

```
switch (val) { // возможна ошибка
case 1:
    cout << "case 1\n";
case 2:
    cout << "case 2\n";
default:
    cout << "default: case not found\n";
}
```

при val==1 напечатает к большому удивлению непосвященных:

```
case 1
case 2
default: case not found
```

Имеет смысл отметить в комментариях те редкие случаи, когда стандартный переход на следующий вариант оставлен намеренно. Тогда этот переход во всех остальных случаях можно смело считать ошибкой. Для завершения оператора в варианте чаще всего используется `break`, но иногда используются `return` и даже `goto`. Приведем пример:

```
switch (val)  { // возможна ошибка
```

```
case 0:
    cout << "case 0\n";
case1:
case 1:
    cout << "case 1\n";
    return;
case 2:
    cout << "case 2\n";
    goto case1;
default:
    cout << "default: case not found\n";
    return;
}
```

Здесь при значении val равном 2 мы получим:

```
case 2
case 1
```

Отметим, что метку варианта нельзя использовать в операторе goto:

```
goto case 2;  // синтаксическая ошибка
```

3.3.2 Оператор goto

Презираемый оператор `goto` все-таки есть в C++:

```
goto идентификатор;

идентификатор: оператор
```

Вообще говоря, он мало используется в языках высокого уровня, но может быть очень полезен, если текст на C++ создается не человеком, а автоматически, т.е. с помощью программы. Например, операторы `goto` используются при создании анализатора по заданной грамматике языка с помощью программных средств. Кроме того, операторы `goto` могут пригодиться в тех случаях, когда на первый план выходит скорость работы программы. Один из них - когда в реальном времени происходят

какие-то вычисления во внутреннем цикле программы.

Есть немногие ситуации и в обычных программах, когда применение `goto` оправдано. Одна из них - выход из вложенного цикла или переключателя. Дело в том, что оператор `break` во вложенных циклах или переключателях позволяет перейти только на один уровень выше. Приведем пример:

```
void f()
{
  int i;
  int j;

  for ( i = 0; i < n; i++)
for (j = 0; j<m; j++)
  if (nm[i][j] == a) goto found;
// здесь a не найдено
// ...
  found:
//  nm[i][j] == a
}
```

Есть еще оператор `continue`, который позволяет перейти на конец цикла.

3.4 Комментарии и расположение текста

Программу гораздо легче читать, и она становится намного понятнее, если разумно использовать комментарии и систематически выделять текст программы пробелами. Есть несколько способов расположения текста программы, но нет причин считать, что один из них - наилучший. Хотя у каждого свой вкус. То же можно сказать и о комментариях.

Однако можно заполнить программу такими комментариями, что читать и понимать ее будет только труднее. Транслятор не в силах понять комментарий, поэтому он не может убедиться в том, что комментарий:

1. осмысленный,

2. действительно описывает программу,
3. не устарел.

Во многих программах попадаются непостижимые, двусмысленные и просто неверные комментарии. Лучше вообще обходиться без них, чем давать такие комментарии.

Если некий факт можно прямо выразить в языке, то так и следует делать, и не надо считать, что достаточно упомянуть его в комментарии. Последнее замечание относится к комментариям, подобным приведенным ниже:

```
// переменную "v" необходимо инициализировать.

// переменная "v" может использоваться только в функции "f()".

// до вызова любой функции из этого файла
// необходимо вызвать функцию "init()".

// в конце своей программы вызовите функцию "cleanup()".

// не используйте функцию "weird()".

// функция "f()" имеет два параметра.
```

При правильном программировании на C++ такие комментарии обычно оказываются излишними. Чтобы именно эти комментарии стали ненужными, можно воспользоваться правилами связывания и областей видимости, а также правилами инициализации и уничтожения объектов класса.

Если некоторое утверждение выражается самой программой, не нужно повторять его в комментарии. Например:

```
a = b + c;// a  принимает значение b+c
count++;// увеличим счетчик count
```

Такие комментарии хуже, чем избыточные. Они раздувают объем текста, затуманивают программу и могут быть даже ложными. В то же время комментарии именно такого рода используют для примеров в

учебниках по языкам программирования, подобных этой книге. Это одна из многих причин, по которой учебная программа отличается от настоящей.

Можно рекомендовать такой стиль введения комментариев в программу:

1. начинать с комментария каждый файл программы: указать в общих чертах, что в ней определяется, дать ссылки на справочные руководства, общие идеи по сопровождению программы и т.д.;
2. снабжать комментарием каждое определение класса или шаблона типа;
3. комментировать каждую нетривиальную функцию, указав: ее назначение, используемый алгоритм (если только он неочевиден) и, возможно, предположения об окружении, в котором работает функция;
4. комментировать определение каждой глобальной переменной;
5. давать некоторое число комментариев в тех местах, где алгоритм неочевиден или непереносим;
6. больше практически ничего.

Приведем пример:

```
// tbl.c: Реализация таблицы имен.

 /*
    Использован метод Гаусса
см. Ральстон "Начальный курс по ..." стр. 411.
 */

// в swap() предполагается, что стек AT&T начинается с 3B20.

/************************************

Авторские права (c) 1991 AT&T, Inc
Все права сохранены

************************************/
```

Правильно подобранные и хорошо составленные комментарии играют в программе важную роль. Написать хорошие комментарии не менее трудно, чем саму программу, и это - искусство, в котором стоит совершенствоваться.

Заметим, что если в функции используются только комментарии вида //, то любую ее часть можно сделать комментарием с помощью /* */, и наоборот.

Функции

Все нетривиальные программы состоят из нескольких раздельно транслируемых единиц, по традиции называемых файлами. В этой лекции описано, как раздельно транслируемые функции могут вызывать друг друга, каким образом они могут иметь общие данные, и как добиться непротиворечивости типов, используемых в разных файлах программы. Подробно обсуждаются функции, в том числе: передача параметров, перегрузка имени функции, стандартные значения параметров, указатели на функции и, естественно, описания и определения функций. В конце лекции обсуждаются макровозможности языка.

4.1 Введение

Роль файла в языке C++ сводится к тому, что он определяет файловую область видимости. Это область видимости глобальных функций (как статических, так и подстановок), а также глобальных переменных (как статических, так и со спецификацией const). Кроме того, файл является традиционной единицей хранения в системе, а также единицей трансляции. Обычно системы хранят, транслируют и представляют пользователю программу на C++ как множество файлов, хотя существуют системы, устроенные иначе. В этой лекции будет обсуждаться в основном традиционное использование файлов.

Всю программу поместить в один файл, как правило, невозможно, поскольку программы стандартных функций и программы операционной системы нельзя включить в текстовом виде в программу пользователя. Вообще, помещать всю программу пользователя в один файл обычно неудобно и непрактично. Разбиения программы на файлы может облегчить понимание общей структуры программы и дает транслятору возможность поддерживать эту структуру. Если единицей трансляции является файл, то даже при небольшом изменении в нем следует его перетранслировать. Даже для программ не слишком большого размера время на перетрансляцию можно значительно сократить, если ее разбить на файлы подходящего размера.

Вернемся к примеру с калькулятором. Решение было дано в виде одного

файла. Когда вы попытаетесь его транслировать, неизбежно возникнут некоторые проблемы с порядком описаний. По крайней мере одно "ненастоящее" описание придется добавить к тексту, чтобы транслятор мог разобраться в использующих друг друга функциях `expr()`, `term()` и `prim()`. По тексту программы видно, что она состоит из четырех частей: лексический анализатор (сканер), собственно анализатор, таблица имен и драйвер. Однако, этот факт никак не отражен в самой программе. На самом деле калькулятор не был запрограммирован именно так. Так не следует писать программу. Даже если не учитывать все рекомендации по программированию, сопровождению и оптимизации для такой "зряшной" программы, все равно ее следует создавать из нескольких файлов хотя бы для удобства.

Чтобы раздельная трансляция стала возможной, программист должен предусмотреть описания, из которых транслятор получит достаточно сведений о типах для трансляции файла, составляющего только часть программы. Требование непротиворечивости использования всех имен и типов для программы, состоящей из нескольких раздельно транслируемых частей, так же справедливо, как и для программы, состоящей из одного файла. Это возможно только в том случае, когда описания, находящиеся в разных единицах трансляции, будут согласованы. В вашей системе программирования имеются средства, которые способны установить, выполняется ли это. В частности, многие противоречия обнаруживает редактор связей. Редактор связей - это программа, которая связывает по именам раздельно транслируемые части программы. Иногда его по ошибке называют загрузчиком.

4.2 Связывание

Если явно не определено иначе, то имя, не являющееся локальным для некоторой функции или класса, должно обозначать один и тот же тип, значение, функцию или объект во всех единицах трансляции данной программы. Иными словами, в программе может быть только один нелокальный тип, значение, функция или объект с данным именем. Рассмотрим для примера два файла:

```
// file1.c
 int a = 1;
```

```
int f() { /* какие-то операторы */ }

// file2.c
extern int a;
int f();
void g() { a = f(); }
```

В функции `g()` используются те самые `a` и `f()`, которые определены в файле file1.c. Служебное слово `extern` показывает, что описание a в файле file2.c является только описанием, но не определением. Если бы присутствовала инициализация a, то `extern` просто проигнорировалось бы, поскольку описание с инициализацией всегда считается определением. Любой объект в программе может определяться только один раз. Описываться же он может неоднократно, но все описания должны быть согласованы по типу. Например:

```
// file1.c:
    int a = 1;
    int b = 1;
    extern int c;

 // file2.c:
    int a;
    extern double b;
    extern int c;
```

Здесь содержится три ошибки: переменная a определена дважды ("int a;" - это определение, означающее "int a=0;"); `b` описано дважды, причем с разными типами; c описано дважды, но неопределено. Такие ошибки (ошибки связывания) транслятор, который обрабатывает файлы по отдельности, обнаружить не может, но большая их часть обнаруживается редактором связей.

Следующая программа допустима в C, но не в C++:

```
// file1.c:
    int a;
    int f() { return a; }
```

```
// file2.c:
   int a;
   int g() { return f(); }
```

Во-первых, ошибкой является вызов `f()` в file2.c, поскольку в этом файле `f()` не описана. Во-вторых, файлы программы не могут быть правильно связаны, поскольку a определено дважды.

Если имя описано как `static`, оно становится локальным в этом файле. Например:

```
// file1.c:
   static int a = 6;
   static int f() { /* ... */ }

// file2.c:
   static int a = 7;
   static int f() { /* ... */ }
```

Приведенная программа правильна, поскольку `a` и `f` определены как статические. В каждом файле своя переменная a и функция `f()`.

Если переменные и функции в данной части программы описаны как `static`, то в этой части программы проще разобраться, поскольку не нужно заглядывать в другие части. Описывать функции как статические полезно еще и по той причине, что транслятору предоставляется возможность создать более простой вариант операции вызова функции. Если имя объекта или функции локально в данном файле, то говорят, что объект подлежит внутреннему связыванию. Обратно, если имя объекта или функции нелокально в данном файле, то он подлежит внешнему связыванию.

Обычно говорят, что имена типов, т.е. классов и перечислений, не подлежат связыванию. Имена глобальных классов и перечислений должны быть уникальными во всей программе и иметь единственное определение. Поэтому, если есть два даже идентичных определения одного класса, это - все равно ошибка:

```
// file1.c:
   struct S { int a; char b; };
```

```
    extern void f(S*);

// file2.c:
    struct S { int a; char b; };
    void f(S* p) { /* ... */ }
```

Но будьте осторожны: опознать идентичность двух описаний класса не в состоянии большинство систем программирования C++. Такое дублирование может вызвать довольно тонкие ошибки (ведь классы в разных файлах будут считаться различными).

Глобальные функции-подстановки подлежат внутреннему связыванию, и то же по умолчанию справедливо для констант. Синонимы типов, т.е. имена `typedef`, локальны в своем файле, поэтому описания в двух данных ниже файлах не противоречат друг другу:

```
// file1.c:
    typedef int T;
    const int a = 7;
    inline T f(int i) { return i+a; }

// file2.c:
    typedef void T;
    const int a = 8;
    inline T f(double d) { cout<<d; }
```

Константа может получить внешнее связывание только с помощью явного описания:

```
// file3.c:
    extern const int a;
    const int a = 77;

// file4.c:
    extern const int a;
    void g() { cout<<a; }
```

В этом примере `g()` напечатает 77.

4.3 Заголовочные файлы

Типы одного объекта или функции должны быть согласованы во всех их описаниях. Должен быть согласован по типам и входной текст, обрабатываемый транслятором, и связываемые части программы. Есть простой, хотя и несовершенный, способ добиться согласованности описаний в различных файлах. Это: включить во входные файлы, содержащие операторы и определения данных, заголовочные файлы, которые содержат интерфейсную информацию.

Средством включения текстов служит макрокоманда #include, которая позволяет собрать в один файл (единицу трансляции) несколько исходных файлов программы. Команда

```
#include "включаемый-файл"
```

заменяет строку, в которой она была задана, на содержимое файла включаемый-файл. Естественно, это содержимое должно быть текстом на C++, поскольку его будет читать транслятор. Как правило, операция включения реализуется отдельной программой, называемой препроцессором C++. Она вызывается системой программирования перед собственно трансляцией для обработки таких команд во входном тексте. Возможно и другое решение: часть транслятора, непосредственно работающая с входным текстом, обрабатывает команды включения файлов по мере их появления в тексте. В той системе программирования, в которой работает автор, чтобы увидеть результат команд включения файлов, нужно задать команду:

```
CC -E file.c
```

Эта команда для обработки файла file.c запускает препроцессор (и только!), подобно тому, как команда CC без флага `-E` запускает сам транслятор.

Для включения файлов из стандартных каталогов (обычно каталоги с именем INCLUDE) надо вместо кавычек использовать угловые скобки < и >. Например:

```
#include <stream.h>    // включение из стандартного каталога
```

```
#include "myheader.h" // включение из текущего каталога
```

Включение из стандартных каталогов имеет то преимущество, что имена этих каталогов никак не связаны с конкретной программой (обычно вначале включаемые файлы ищутся в каталоге /usr/include/CC, а затем в /usr/include). К сожалению, в этой команде пробелы существенны:

```
#include < stream.h>  // <stream.h> не будет найден
```

Было бы нелепо, если бы каждый раз перед включением файла требовалась его перетрансляция. Обычно включаемые файлы содержат только описания, а не операторы и определения, требующие существенной трансляторной обработки. Кроме того, система программирования может предварительно оттранслировать заголовочные файлы, если, конечно, она настолько развита, что способна сделать это, не изменяя семантики программы.

Укажем, что может содержать заголовочный файл:

```
Определения типов struct point { int x, y; };
   Шаблоны типов   template<class T>
class V { /* ... */ }
   Описания функций extern int strlen(const char*);
   Определения     inline char get() { return *p++; }
   функций-подстановок
   Описания данных extern int a;
   Определения констант const float pi = 3.141593;
   Перечисления    enum bool { false, true };
   Описания имен   class Matrix;
   Команды включения файлов    #include <signal.h>
   Макроопределения #define Case break;case
   Комментарии     /* проверка на конец файла */
```

Перечисление того, что стоит помещать в заголовочный файл, не является требованием языка, это просто совет по разумному использованию включения файлов. С другой стороны, в заголовочном файле никогда не должно быть:

```
Определений обычных функций  char get() { return *p++; }
```

Определений данных int a;
Определений составных const tb[i] = { /* ... */ };
констант

По традиции заголовочные файлы имеют расширение .h, а файлы, содержащие определения функций или данных, расширение .c. Иногда их называют "h-файлы" или "c-файлы" соответственно. Используют и другие расширения для этих файлов: .C, cxx, .cpp и .cc. Макросредства в C++ используются не столь широко, как в C, поскольку C++ имеет определенные возможности в самом языке: определения констант (const), функций-подстановок (inline), дающие возможность более простой операции вызова, и шаблонов типа, позволяющие порождать семейство типов и функций.

Совет помещать в заголовочный файл определения только простых, но не составных, констант объясняется вполне прагматической причиной. Просто большинство трансляторов не настолько разумно, чтобы предотвратить создание ненужных копий составной константы. Вообще говоря, более простой вариант всегда является более общим, а значит транслятор должен учитывать его в первую очередь, чтобы создать хорошую программу.

4.3.1 Единственный заголовочный файл

Проще всего разбить программу на несколько файлов следующим образом: поместить определения всех функций и данных в некоторое число входных файлов, а все типы, необходимые для связи между ними, описать в единственном заголовочном файле. Все входные файлы будут включать заголовочный файл. Программу калькулятора можно разбить на четыре входных файла .c: lex.c, syn.c, table.c и main.c. Заголовочный файл dc.h будет содержать описания каждого имени, которое используется более чем в одном .c файле:

```
// dc.h: общее описание для калькулятора

#include <iostream.h>

enum token_value {
```

```
    NAME,     NUMBER,    END,
    PLUS='+', MINUS='-', MUL='*', DIV='/',
    PRINT=';', ASSIGN='=', LP='(', RP=')'
  };

  extern int no_of_errors;
  extern double error(const char* s);
  extern token_value get_token();
  extern token_value curr_tok;
  extern double number_value;
  extern char name_string[256];
  extern double expr();
  extern double term();
  extern double prim();

  struct name {
    char* string;
    name* next;
    double value;
  };

  extern name* look(const char* p, int ins = 0);
  inline name* insert(const char* s) { return look(s,1); }
```

Если не приводить сами операторы, lex.c должен иметь такой вид:

```
  // lex.c: ввод и лексический анализ

  #include "dc.h"
  #include <ctype.h>

  token_value curr_tok;
  double number_value;
  char name_string[256];

  token_value get_token() { /* ... */ }
```

Используя составленный заголовочный файл, мы добьемся, что описание каждого объекта, введенного пользователем, обязательно

окажется в том файле, где этот объект определяется. Действительно, при обработке файла lex.c транслятор столкнется с описаниями

```
extern token_value get_token();
// ...
token_value get_token() { /* ... */ }
```

Это позволит транслятору обнаружить любое расхождение в типах, указанных при описании данного имени. Например, если бы функция `get_token()` была описана с типом `token_value`, но определена с типом int, трансляция файла lex.c выявила бы ошибку: несоответствие типа.

Файл syn.c может иметь такой вид:

```
// syn.c: синтаксический анализ и вычисления

#include "dc.h"

double prim() { /* ... */ }
double term() { /* ... */ }
double expr() { /* ... */ }
```

Файл table.c может иметь такой вид:

```
// table.c: таблица имен и функция поиска

#include "dc.h"

extern char* strcmp(const char*, const char*);
extern char* strcpy(char*, const char*);
extern int strlen(const char*);

const int TBLSZ = 23;
name* table[TBLSZ];

name* look(char* p, int ins) { /* ... */ }
```

Отметим, что раз строковые функции описаны в самом файле table.c, транслятор не может проверить согласованность этих описаний по

типам. Всегда лучше включить соответствующий заголовочный файл, чем описывать в файле .c некоторое имя как `extern`. Это может привести к включению "слишком многого", но такое включение не страшно, поскольку не влияет на скорость выполнения программы и ее размер, а программисту позволяет сэкономить время. Допустим, функция `strlen()` снова описывается в приведенном ниже файле main.c. Это только лишний ввод символов и потенциальный источник ошибок, т.к. транслятор не сможет обнаружить расхождения в двух описаниях `strlen()` (впрочем, это может сделать редактор связей). Такой проблемы не возникло бы, если бы в файле dc.h содержались все описания `extern`, как первоначально и предполагалось. Подобная небрежность присутствует в нашем примере, поскольку она типична для программ на C. Она очень естественна для программиста, но часто приводит к ошибкам и таким программам, которые трудно сопровождать. Итак, предупреждение сделано!

Наконец, приведем файл main.c:

```
// main.c: инициализация, основной цикл, обработка ошибок

#include "dc.h"

double error(char* s) { /* ... */ }

extern int strlen(const char*);

int main(int argc, char* argv[]) { /* ... */ }
```

В одном важном случае заголовочные файлы вызывают большое неудобство. С помощью серии заголовочных файлов и стандартной библиотеки расширяют возможности языка, вводя множество типов (как общих, так и рассчитанных на конкретные приложения; см. лекции 5-9). В таком случае текст каждой единицы трансляции может начинаться тысячами строк заголовочных файлов. Содержимое заголовочных файлов библиотеки, как правило, стабильно и меняется редко. Здесь очень пригодился бы претранслятор, который обрабатывает его. По сути, нужен язык специального назначения со своим транслятором. Но устоявшихся методов построения такого претранслятора пока нет.

4.3.2 Множественные заголовочные файлы

Разбиение программы в расчете на один заголовочный файл больше подходит для небольших программ, отдельные части которых не имеют самостоятельного назначения. Для таких программ допустимо, что по заголовочному файлу нельзя определить, чьи описания там находятся и по какой причине. Здесь могут помочь только комментарии. Возможно альтернативное решение: пусть каждая часть программы имеет свой заголовочный файл, в котором определяются средства, предоставляемые другим частям. Теперь для каждого файла .c будет свой файл .h, определяющий, что может предоставить первый. Каждый файл .c будет включать как свой файл .h, так и некоторые другие файлы .h, исходя из своих потребностей.

Попробуем использовать такую организацию программы для калькулятора. Заметим, что функция `error()` нужна практически во всех функциях программы, а сама использует только <iostream.h>. Такая ситуация типична для функций, обрабатывающих ошибки. Следует отделить ее от файла main.c:

```
// error.h: обработка ошибок

extern int no_of_errors;

extern double error(const char* s);

// error.c

#include <iostream.h>
#include "error.h"

int no_of_errors;

double error(const char* s) { /* ... */ }
```

При таком подходе к разбиению программы каждую пару файлов .c и .h можно рассматривать как модуль, в котором файл .h задает его интерфейс, а файл .c определяет его реализацию. Таблица имен не

зависит ни от каких частей калькулятора, кроме части обработки ошибок. Теперь этот факт можно выразить явно:

```
// table.c: описание таблицы имен

struct name {
 char* string;
 name* next;
 double value;
};

extern name* look(const char* p, int ins = 0);
inline name* insert(const char* s) { return look(s,1); }

// table.c: определение таблицы имен

#include "error.h"
#include <string.h>
#include "table.c"

const int TBLSZ = 23;
name* table[TBLSZ];

name* look(const char* p, int ins) { /* ... */ }
```

Заметьте, что теперь описания строковых функций берутся из включаемого файла <string.h>. Тем самым удален еще один источник ошибок.

```
// lex.h: описания для ввода и лексического анализа

 enum token_value {
NAME,       NUMBER,END,
PLUS='+',   MINUS='-',    MUL='*',
PRINT=';',  ASSIGN='=',   LP='(',   RP= ')'
};

extern token_value curr_tok;
extern double number_value;
```

```
extern char name_string[256];

extern token_value get_token();
```

Интерфейс с лексическим анализатором достаточно запутанный. Поскольку недостаточно соответствующих типов для лексем, пользователю функции `get_token()` предоставляются те же буферы `number_value` и `name_string`, с которыми работает сам лексический анализатор.

```
// lex.c: определения для ввода и лексического анализа

#include <iostream.h>
#include <ctype.h>
#include "error.h"
#include "lex.h"

token_value curr_tok;
double number_value;
char name_string[256];

token_value get_token() { /* ... */ }
```

Интерфейс с синтаксическим анализатором определен четко:

```
// syn.h: описания для синтаксического анализа и вычислений

extern double expr();
extern double term();
extern double prim();

// syn.c: определения для синтаксического анализа и вычислений

#include "error.h"
#include "lex.h"
#include "syn.h"

double prim() { /* ... */ }
double term() { /* ... */ }
```

```
double expr() { /* ... */ }
```

Как обычно, определение основной программы тривиально:

```
// main.c: основная программа

#include <iostream.h>
#include "error.h"
#include "lex.h"
#include "syn.h"
#include "table.c"

int main(int argc, char* argv[]) { /* ... */ }
```

Какое число заголовочных файлов следует использовать для данной программы, зависит от многих факторов. Большинство их определяется способом обработки файлов именно в вашей системе, а не собственно в C++. Например, если ваш редактор не может работать одновременно с несколькими файлами, диалоговая обработка нескольких заголовочных файлов затрудняется. Другой пример: может оказаться, что открытие и чтение 10 файлов по 50 строк каждый занимает существенно больше времени, чем открытие и чтение одного файла из 500 строк. В результате придется хорошенько подумать, прежде чем разбивать небольшую программу, используя множественные заголовочные файлы. Предостережение: обычно можно управиться с множеством, состоящим примерно из 10 заголовочных файлов (плюс стандартные заголовочные файлы). Если же вы будете разбивать программу на минимальные логические единицы с заголовочными файлами (например, создавая для каждой структуры свой заголовочный файл), то можете очень легко получить неуправляемое множество из сотен заголовочных файлов.

4.4 Связывание с программами на других языках

Программы на C++ часто содержат части, написанные на других языках, и наоборот, часто фрагмент на C++ используется в программах, написанных на других языках. Собрать в одну программу фрагменты, написанные на разных языках, или, написанные на одном языке, но в системах программирования с разными соглашениями о связывании, достаточно трудно. Например, разные языки или разные реализации

одного языка могут различаться использованием регистров при передаче параметров, порядком размещения параметров в стеке, упаковкой таких встроенных типов, как целые или строки, форматом имен функций, которые транслятор передает редактору связей, объемом контроля типов, который требуется от редактора связей. Чтобы упростить задачу, можно в описании внешних функций указать условие связывания. Например, следующее описание объявляет strcpy внешней функцией и указывает, что она должна связываться согласно порядку связывания в C:

```
extern "C" char* strcpy(char*, const char*);
```

Результат этого описания отличается от результата обычного описания

```
extern char* strcpy(char*, const char*);
```

только порядком связывания для вызывающих `strcpy()` функций. Сама семантика вызова и, в частности, контроль фактических параметров будут одинаковы в обоих случаях. Описание extern "C" имеет смысл использовать еще и потому, что языки C и C++, как и их реализации, близки друг другу. Отметим, что в описании extern "C" упоминание C относится к порядку связывания, а не к языку, и часто такое описание используют для связи с Фортраном или ассемблером. Эти языки в определенной степени подчиняются порядку связывания для C.

Утомительно добавлять "C" ко многим описаниям extern, и есть возможность указать такую спецификацию сразу для группы описаний. Например:

```
extern "C" {
    char* strcpy(char*, const char);
    int strcmp(const char*, const char*)
    int strlen(const char*)
    // ...
}
```

В такую конструкцию можно включить весь заголовочный файл C, чтобы указать, что он подчиняется связыванию для C++, например:

```
extern "C" {
    #include <string.h>
}
```

Обычно с помощью такого приема из стандартного заголовочного файла для C получают такой файл для C++. Возможно иное решение с помощью условной трансляции:

```
#ifdef __cplusplus
extern "C" {
#endif

    char* strcpy(char*, const char*);
    int strcmp(const char*, const char*);
    int strlen(const char*);
    // ...

#ifdef __cplusplus
}
#endif
```

Предопределенное макроопределение `__cplusplus` нужно, чтобы обойти конструкцию `extern "C" { ... }`, если заголовочный файл используется для C.

Поскольку конструкция `extern "C" { ... }` влияет только на порядок связывания, в ней может содержаться любое описание, например:

```
extern "C" {
    // произвольные описания

    // например:

    static int st;
    int glob;
}
```

Никак не меняется класс памяти и область видимости описываемых объектов, поэтому по-прежнему `st` подчиняется внутреннему

связыванию, а `glob` остается глобальной переменной.

Укажем еще раз, что описание `extern "C"` влияет только на порядок связывания и не влияет на порядок вызова функции. В частности, функция, описанная как `extern "C"`, все равно подчиняется правилам контроля типов и преобразования фактических параметров, которые в C++ строже, чем в C. Например:

```
extern "C" int f();

int g()
{
  return f(1);  // ошибка: параметров быть не должно
}
```

4.5 Как создать библиотеку

Распространены такие обороты (и в этой книге тоже): "поместить в библиотеку", "поискать в такой-то библиотеке". Что они означают для программ на C++? К сожалению, ответ зависит от используемой системы. В этом разделе говорится о том, как создать и использовать библиотеку для десятой версии системы ЮНИКС. Другие системы должны предоставлять похожие возможности. Библиотека состоит из файлов .o, которые получаются в результате трансляции файлов .c. Обычно существует один или несколько файлов .h, в которых содержатся необходимые для вызова файлов .o описания. Рассмотрим в качестве примера, как для четко не оговоренного множества пользователей можно достаточно удобно определить некоторое множество стандартных математических функций. Заголовочный файл может иметь такой вид:

```
extern "C" { // стандартные математические функции
   // как правило написаны на C

     double sqrt(double); // подмножество <math.h>
     double sin(double);
     double cos(double);
     double exp(double);
     double log(double);
```

```
    // ...

}
```

Определения этих функций будут находиться в файлах sqrt.c, sin.c, cos.c, exp.c и log.c, соответственно.

Библиотеку с именем math.a можно создать с помощью таких команд:

```
$ CC -c sqrt.c sin.c cos.c exp.c log.c
$ ar cr math.a sqrt.o sin.o cos.o exp.o log.o
$ ranlib math.a
```

Здесь символ $ является приглашением системы.

Вначале транслируются исходные тексты, и получаются модули с теми же именами. Команда ar (архиватор) создает архив под именем math.a. Наконец, для быстрого доступа к функциям архив индексируется. Если в вашей системе нет команды `ranlib` (возможно она и не нужна), то, по крайней мере, можно найти в справочном руководстве ссылку на имя ar. Чтобы использовать библиотеку в своей программе, надо задать режим трансляции следующим образом:

```
$ CC myprog.c math.a
```

Встает вопрос: что дает нам библиотека math.a? Ведь можно было бы непосредственно использовать файлы .o, например так:

```
$ CC myprog.c sqrt.o sin.o cos.o exp.o log.o
```

Дело в том, что во многих случаях трудно правильно указать, какие файлы .o действительно нужны. В приведенной выше команде использовались все из них. Если же в myprog вызываются только sqrt() и cos(), тогда, видимо, достаточно задать такую команду:

```
$ CC myprog.c sqrt.o cos.o
```

Но это будет неверно, т.к. функция cos() вызывает sin().

Редактор связей, который вызывается командой CC для обработки файлов .a (в нашем случае для файла math.a), умеет из множества

файлов, образующих библиотеку, извлекать только нужные файлы .o. Иными словами, связывание с библиотекой позволяет включать в программы много определений одного имени (в том числе определения функций и переменных, используемых только внутренними функциями, о которых пользователь никогда не узнает). В то же время в результирующую программу войдет только минимально необходимое число определений.

4.6 Функции

Самый распространенный способ задания в C++ каких-то действий - это вызов функции, которая выполняет такие действия. Определение функции есть описание того, как их выполнить. Неописанные функции вызывать нельзя.

4.6.1 Описания функций

Описание функции содержит ее имя, тип возвращаемого значения (если оно есть) и число и типы параметров, которые должны задаваться при вызове функции. Например:

```
extern double sqrt(double);
extern elem* next_elem();
extern char* strcpy(char* to, const char* from);
extern void exit(int);
```

Семантика передачи параметров тождественна семантике инициализации: проверяются типы фактических параметров и, если нужно, происходят неявные преобразования типов. Так, если учесть приведенные описания, то в следующем определении:

```
double sr2 = sqrt(2);
```

содержится правильный вызов функции `sqrt()` со значением с плавающей точкой 2.0. Контроль и преобразование типа фактического параметра имеет в C++ огромное значение.

В описании функции можно указывать имена параметров. Это

облегчает чтение программы, но транслятор эти имена просто игнорирует.

4.6.2 Определения функций

Каждая вызываемая в программе функция должна быть где-то в ней определена, причем только один раз. Определение функции - это ее описание, в котором содержится тело функции. Например:

```
extern void swap(int*, int*);  // описание

void swap(int* p, int* q)      // определение
{
  int t = *p;
  *p = *q;
  *q = *t;
}
```

Не так редки случаи, когда в определении функции не используются некоторые параметры:

```
void search(table* t, const char* key, const char*)
{
  // третий параметр не используется

  // ...
}
```

Как видно из этого примера, параметр не используется, если не задано его имя. Подобные функции появляются при упрощении программы или если рассчитывают на ее дальнейшее расширение. В обоих случаях резервирование места в определении функции для неиспользуемого параметра гарантирует, что другие функции, содержащие вызов данной, не придется менять.

Уже говорилось, что функцию можно определить как подстановку (`inline`). Например:

```
inline fac(int i) { return i<2 ? 1 : n*fac(n-1); }
```

Спецификация `inline` служит подсказкой транслятору, что вызов функции fac можно реализовать подстановкой ее тела, а не с помощью обычного механизма вызова функций. Хороший оптимизирующий транслятор вместо генерации вызова `fac(6)` может просто использовать константу 720. Из-за наличия взаиморекурсивных вызовов функций-подстановок, а также функций-подстановок, рекурсивность которых зависит от входных данных, нельзя утверждать, что каждый вызов функции-подстановки действительно реализуется подстановкой ее тела. Степень оптимизации, проводимой транслятором, нельзя формализовать, поэтому одни трансляторы создадут команды `6*5*4*3*2*1`, другие - `6*fac(5)`, а некоторые ограничатся неоптимизированным вызовом `fac(6)`.

Чтобы реализация вызова подстановкой стала возможна даже для не слишком развитых систем программирования, нужно, чтобы не только определение, но и описание функции-подстановки находилось в текущей области видимости. В остальном спецификация inline не влияет на семантику вызова.

4.6.3 Передача параметров

При вызове функции выделяется память для ее формальных параметров, и каждый формальный параметр инициализируется значением соответствующего фактического параметра. Семантика передачи параметров тождественна семантике инициализации. В частности, сверяются типы формального и соответствующего ему фактического параметра, и выполняются все стандартные и пользовательские преобразования типа. Существуют специальные правила передачи массивов. Есть возможность передать параметр, минуя контроль типа, и возможность задать стандартное значение параметра. Рассмотрим функцию:

```
void f(int val, int& ref)
{
  val++;
  ref++;
```

```
}
```

При вызове `f()` в выражении `val++` увеличивается локальная копия первого фактического параметра, тогда как в `ref++` - сам второй фактический параметр увеличивается сам. Поэтому в функции

```
void g()
{
 int i = 1;
 int j = 1;
 f(i,j);
}
```

увеличится значение `j`, но не `i`. Первый параметр `i` передается по значению, а второй параметр j передается по ссылке. Мы говорили, что функции, которые изменяют свой передаваемый по ссылке параметр, труднее понять, и что поэтому лучше их избегать. Но большие объекты, очевидно, гораздо эффективнее передавать по ссылке, чем по значению. Правда можно описать параметр со спецификацией `const`, чтобы гарантировать, что передача по ссылке используется только для эффективности, и вызываемая функция не может изменить значение объекта:

```
void f(const large& arg)
{
  // значение "arg" нельзя изменить без явных
  // операций преобразования типа
}
```

Если в описании параметра ссылки `const` не указано, то это рассматривается как намерение изменять передаваемый объект:

```
void g(large& arg); // считается, что в g() arg будет меняться
```

Отсюда мораль: используйте `const` всюду, где возможно.

Точно так же, описание параметра, являющегося указателем, со спецификацией `const` говорит о том, что указуемый объект не будет изменяться в вызываемой функции. Например:

```
extern int strlen(const char*);  // из <string.h>
extern char* strcpy(char* to, const char* from);
extern int strcmp(const char*, const char*);
```

Значение такого приема растет вместе с ростом программы.

Отметим, что семантика передачи параметров отличается от семантики присваивания. Это различие существенно для параметров, являющихся `const` или ссылкой, а также для параметров с типом, определенным пользователем.

Литерал, константу и параметр, требующий преобразования, можно передавать как параметр типа `const&`, но без спецификации const передавать нельзя. Допуская преобразования для параметра типа const `T&`, мы гарантируем, что он может принимать значения из того же множества, что и параметр типа `T`, значение которого передается при необходимости с помощью временной переменной.

```
float fsqrt(const float&);  // функция sqrt в стиле Фортрана

 void g(double d)
 {
float r;

r = fsqrt(2.0f);  // передача ссылки на временную
    // переменную, содержащую 2.0f
r = fsqrt(r);     // передача ссылки на r
r = fsqrt(d);     // передача ссылки на временную
    // переменную, содержащую float(d)
}
```

Запрет на преобразования типа для параметров-ссылок без спецификации `const` введен для того, чтобы избежать нелепых ошибок, связанных с использованием при передаче параметров временных переменных:

```
void update(float& i);

void g(double d)
```

```
{
 float r;

 update(2.0f);    // ошибка: параметр-константа
 update(r);       // нормально: передается ссылка на r
 update(d);       // ошибка: здесь нужно преобразовывать тип

}
```

4.6.4 Возвращаемое значение

Если функция не описана как `void`, она должна возвращать значение. Например:

```
int f() { }    // ошибка
void g() { }   // нормально
```

Возвращаемое значение указывается в операторе `return` в теле функции. Например:

```
int fac(int n) { return (n>1) ? n*fac(n-1) : 1; }
```

В теле функции может быть несколько операторов `return`:

```
int fac(int n)
   {
     if (n > 1)
 return n*fac(n-1);
     else
 return 1;
   }
```

Подобно передаче параметров, операция возвращения значения функции эквивалентна инициализации. Считается, что оператор `return` инициализирует переменную, имеющую тип возвращаемого значения. Тип выражения в операторе `return` сверяется с типом функции, и производятся все стандартные и пользовательские преобразования типа. Например:

```
double f()
        {
// ...
return 1;   // неявно преобразуется в double(1)
        }
```

При каждом вызове функции создается новая копия ее формальных параметров и автоматических переменных. Занятая ими память после выхода из функции будет снова использоваться, поэтому неразумно возвращать указатель на локальную переменную. Содержимое памяти, на которую настроен такой указатель, может измениться непредсказуемым образом:

```
int* f()
{
int local = 1;
// ...
return &local;    // ошибка
}
```

Эта ошибка не столь типична, как сходная ошибка, когда тип функции - ссылка:

```
int& f()
{
int local = 1;
// ...
return local;   // ошибка
        }
```

К счастью, транслятор предупреждает о том, что возвращается ссылка на локальную переменную. Вот другой пример:

```
int& f() { return 1; }  // ошибка
```

4.6.5 Параметр-массив

Если в качестве параметра функции указан массив, то передается указатель на его первый элемент. Например:

```
int strlen(const char*);

void f()
{
char v[] = "массив";
strlen(v);
strlen("Николай");
}
```

Это означает, что фактический параметр типа `T[]` преобразуется к типу `T*`, и затем передается. Поэтому присваивание элементу формального параметра-массива изменяет этот элемент. Иными словами, массивы отличаются от других типов тем, что они не передаются и не могут передаваться по значению.

В вызываемой функции размер передаваемого массива неизвестен. Это неприятно, но есть несколько способов обойти данную трудность. Прежде всего, все строки оканчиваются нулевым символом, и значит их размер легко вычислить. Можно передавать еще один параметр, задающий размер массива. Другой способ: определить структуру, содержащую указатель на массив и размер массива, и передавать ее как параметр. Например:

```
void compute1(int* vec_ptr, int vec_size);  // 1-ый способ

struct vec {   // 2-ой способ
int* ptr;
int size;
};

void compute2(vec v);
```

Сложнее с многомерными массивами, но часто вместо них можно использовать массив указателей, сведя эти случаи к одномерным массивам. Например:

```
char* day[] = {
"mon", "tue", "wed", "thu", "fri", "sat", "sun"
};
```

Теперь рассмотрим функцию, работающую с двумерным массивом - матрицей. Если размеры обоих индексов известны на этапе трансляции, то проблем нет:

```
void print_m34(int m[3][4])
{
for (int i = 0; i<3; i++) {
   for (int j = 0; j<4; J++)
      cout << ' ' << m[i][j];
   cout << '\n';
}
}
```

Конечно, матрица по-прежнему передается как указатель, а размерности приведены просто для полноты описания. Первая размерность для вычисления адреса элемента неважна, поэтому ее можно передавать как параметр:

```
void print_mi4(int m[][4], int dim1)
      {
for ( int i = 0; i<dim1; i++) {
  for ( int j = 0; j<4; j++)
     cout << ' ' << m[i][j];
  cout << '\n';
}
      }
```

Самый сложный случай - когда надо передавать обе размерности. Здесь "очевидное" решение просто непригодно:

```
void print_mij(int m[][], int dim1, int dim2)   // ошибка
     {
for ( int i = 0; i<dim1; i++) {
for ( int j = 0; j<dim2; j++)
    cout << ' ' << m[i][j];
cout << '\n';
}
     }
```

Во-первых, описание параметра `m[][]` недопустимо, поскольку для

вычисления адреса элемента многомерного массива нужно знать вторую размерность. Во-вторых, выражение `m[i][j]` вычисляется как `*(*(m+i)+j)`, а это, по всей видимости, не то, что имел в виду программист. Приведем правильное решение:

```
void print_mij(int** m, int dim1, int dim2)
{
for (int i = 0; i< dim1; i++) {
  for (int j = 0; j<dim2; j++)
    cout << ' ' << ((int*)m)[i*dim2+j];  // запутано
  cout << '\n';
}
}
```

Выражение, используемое для выбора элемента матрицы, эквивалентно тому, которое создает для этой же цели транслятор, когда известна последняя размерность. Можно ввести дополнительную переменную, чтобы это выражение стало понятнее:

```
int* v = (int*)m;
// ...
v[i*dim2+j]
```

Лучше такие достаточно запутанные места в программе упрятывать. Можно определить тип многомерного массива с соответствующей операцией индексирования. Тогда пользователь может и не знать, как размещаются данные в массиве.

4.6.6 Перегрузка имени функции

Обычно имеет смысл давать разным функциям разные имена. Если же несколько функций выполняет одно и то же действие над объектами разных типов, то удобнее дать одинаковые имена всем этим функциям. Перегрузкой имени называется его использование для обозначения разных операций над разными типами. Собственно уже для основных операций C++ применяется перегрузка. Действительно: для операций сложения есть только одно имя +, но оно используется для сложения и целых чисел, и чисел с плавающей точкой, и указателей. Такой подход

легко можно распространить на операции, определенные пользователем, т.е. на функции. Например:

```
void print(int);// печать целого
void print(const char*)   // печать строки символов
```

Для транслятора в таких перегруженных функциях общее только одно - имя. Очевидно, по смыслу такие функции сходны, но язык не способствует и не препятствует выделению перегруженных функций. Таким образом, определение перегруженных функций служит, прежде всего, для удобства записи. Но для функций с такими традиционными именами, как sqrt, print или open, нельзя этим удобством пренебрегать. Если само имя играет важную семантическую роль, например, в таких операциях, как + , * и <<, или для конструктора класса, то такое удобство становится существенным фактором. При вызове функции с именем f транслятор должен разобраться, какую именно функцию f следует вызывать. Для этого сравниваются типы фактических параметров, указанные в вызове, с типами формальных параметров всех описаний функций с именем f. В результате вызывается та функция, у которой формальные параметры наилучшим образом сопоставились с параметрами вызова, или выдается ошибка если такой функции не нашлось. Например:

```
void print(double);
 void print(long);

 void f()
 {
print(1L);    // print(long)
print(1.0);   // print(double)
print(1);     // ошибка, неоднозначность: что вызывать
 // print(long(1)) или print(double(1)) ?
 }
```

Правила сопоставления параметров применяются в следующем порядке по убыванию их приоритета:

1. Точное сопоставление: сопоставление произошло без всяких преобразований типа или только с неизбежными

преобразованиями (например, имени массива в указатель, имени функции в указатель на функцию и типа `T` в `const T`).

2. Сопоставление с использованием стандартных целочисленных преобразований (т.е. `char` в `int`, `short` в `int` и их беззнаковых двойников в `int`), а также преобразований `float` в `double`.
3. Сопоставление с использованием стандартных преобразований, (например, `int` в `double`, `derived*` в `base*`, `unsigned` в `int`).
4. Сопоставление с использованием пользовательских преобразований.
5. Сопоставление с использованием эллипсиса ... в описании функции.

Если найдены два сопоставления по самому приоритетному правилу, то вызов считается неоднозначным, а значит ошибочным. Эти правила сопоставления параметров работают с учетом правил преобразований числовых типов для C и C++. Пусть имеются такие описания функции print:

```
void print(int);
void print(const char*);
void print(double);
void print(long);
void print(char);
```

Тогда результаты следующих вызовов `print()` будут такими:

```
void h(char c, int i, short s, float f)
    {
    print(c);    // точное сопоставление: вызывается print(char)
    print(i);    // точное сопоставление: вызывается print(int)
    print(s);    // стандартное целочисленное преобразование:
// вызывается print(int)
    print(f);    // стандартное преобразование:
// вызывается print(double)

    print('a');  // точное сопоставление: вызывается print(char)
    print(49);   // точное сопоставление: вызывается print(int)
    print(0);    // точное сопоставление: вызывается print(int)
```

```
        print("a");  // точное сопоставление:
  // вызывается print(const char*)
        }
```

Обращение `print(0)` приводит к вызову `print(int)`, ведь 0 имеет тип `int`. Обращение `print('a')` приводит к вызову `print(char)`, т.к. 'a' - типа `char`.

Отметим, что на разрешение неопределенности при перегрузке не влияет порядок описаний рассматриваемых функций, а типы возвращаемых функциями значений вообще не учитываются.

Исходя из этих правил можно гарантировать, что если эффективность или точность вычислений значительно различаются для рассматриваемых типов, то вызывается функция, реализующая самый простой алгоритм. Например:

```
int pow(int, int);
double pow(double, double);     // из <math.h>
complex pow(double, complex);   // из <complex.h>
complex pow(complex, int);
complex pow(complex, double);
complex pow(complex, complex);

void k(complex z)
{
int i = pow(2,2);       // вызывается pow(int,int)
double d = pow(2.0,2);  // вызывается pow(double,double)
complex z2 = pow(2,z);  // вызывается pow(double,complex)
complex z3 = pow(z,2);  // вызывается pow(complex,int)
complex z4 = pow(z,z);  // вызывается pow(complex,complex)
 }
```

4.6.7 Стандартные значения параметров

В общем случае у функции может быть больше параметров, чем в самых простых и наиболее часто используемых случаях. В частности, это свойственно функциям, строящим объекты (например, конструкторам).

Для более гибкого использования этих функций иногда применяются необязательные параметры. Рассмотрим в качестве примера функцию печати целого числа. Вполне разумно применить в качестве необязательного параметра основание счисления печатаемого числа, хотя в большинстве случаев числа будут печататься как десятичные целые значения. Следующая функция

```
void print (int value, int base =10);

void F()
{
 print(31);
 print(31,10);
 print(31,16);
 print(31,2);
}
```

напечатает такие числа:

```
31 31 1f 11111
```

Вместо стандартного значения параметра можно было бы использовать перегрузку функции `print`:

```
void print(int value, int base);
 inline void print(int value) { print(value,10); }
```

Однако в последнем варианте текст программы не столь явно демонстрирует желание иметь одну функцию `print`, но при этом обеспечить удобную и краткую форму записи.

Тип стандартного параметра сверяется с типом указанного значения при трансляции описания функции, а значение этого параметра вычисляется в момент вызова функции. Задавать стандартное значение можно только для завершающих подряд идущих параметров:

```
int f(int, int =0, char* =0);   // нормально
int g(int =0, int =0, char*);   // ошибка
int h(int =0, int, char* =0);   // ошибка
```

Отметим, что в данном контексте наличие пробела между символами * и = весьма существенно, поскольку *= является операцией присваивания:

```
int nasty(char*=0);      // синтаксическая ошибка
```

4.6.8 Неопределенное число параметров

Существуют функции, в описании которых невозможно указать число и типы всех допустимых параметров. Тогда список формальных параметров завершается эллипсисом (...), что означает: "и, возможно, еще несколько аргументов". Например:

```
int printf(const char* ...);
```

При вызове `printf` обязательно должен быть указан параметр типа `char*`, однако могут быть (а могут и не быть) еще другие параметры. Например:

```
printf("Hello, world\n");
printf("My name is %s %s\n", first_name, second_name);
printf("%d + %d = %d\n", 2,3,5);
```

Такие функции пользуются для распознавания своих фактических параметров недоступной транслятору информацией. В случае функции `printf` первый параметр является строкой, специфицирующей формат вывода. Она может содержать специальные символы, которые позволяют правильно воспринять последующие параметры. Например, `%s` означает -"будет фактический параметр типа `char*`", %d означает - "будет фактический параметр типа `int`". Но транслятор этого не знает, и поэтому он не может убедиться, что объявленные параметры действительно присутствуют в вызове и имеют соответствующие типы. Например, следующий вызов

```
printf("My name is %s %s\n",2);
```

нормально транслируется, но приведет (в лучшем случае) к неожиданной выдаче. Можете проверить сами.

Очевидно, что раз параметр не описан, то транслятор не имеет сведений для контроля и стандартных преобразований типа этого параметра. Поэтому `char` или `short` передаются как `int`, а `float` как `double`, хотя пользователь, возможно, имел в виду другое.

В хорошо продуманной программе может потребоваться, в виде исключения, лишь несколько функций, в которых указаны не все типы параметров. Чтобы обойти контроль типов параметров, лучше использовать перегрузку функций или стандартные значения параметров, чем параметры, типы которых не были описаны. Эллипсис становится необходимым только тогда, когда могут меняться не только типы, но и число параметров. Чаще всего эллипсис используется для определения интерфейса с библиотекой стандартных функций на C, если этим функциям нет замены:

```
extern "C" int fprintf(FILE*, const char* ...);
 extern "C" int execl(const char* ...);
```

Есть стандартный набор макроопределений, находящийся в <stdarg.h>, для выбора незаданных параметров этих функций. Рассмотрим функцию реакции на ошибку, первый параметр которой показывает степень тяжести ошибки. За ним может следовать произвольное число строк. Нужно составить сообщение об ошибке с учетом, что каждое слово из него передается как отдельная строка:

```
extern void error(int ...)
extern char* itoa(int);

main(int argc, char* argv[])
{
 switch (argc) {
 case 1:
error(0,argv[0],(char*)0);
break;
 case 2:
error(0,argv[0],argv[1],(char*)0);
break;
 default:
error(1,argv[0],
```

```
        "With",itoa(argc-1),"arguments",(char*)0);
    }
    // ...
}
```

Функция `itoa` возвращает строку символов, представляющую ее целый параметр. Функцию реакции на ошибку можно определить так:

```
#include <stdarg.h>

void error(int severity ...)
        /*
    за "severity" (степень тяжести ошибки) следует
    список строк, завершающийся нулем
        */
        {
    va_list ap;
    va_start(ap,severity);   // начало параметров

    for (;;) {
    char* p = va_arg(ap,char*);
    if (p == 0) break;
    cerr << p << ' ';
    }

    va_end(ap);      // очистка параметров

    cerr << '\n';
    if (severity) exit(severity);
    }
```

Вначале при вызове `va_start()` определяется и инициализируется `va_list`. Параметрами макроопределения `va_start` являются имя типа `va_list` и последний формальный параметр. Для выборки по порядку неописанных параметров используется макроопределение `va_arg()`. В каждом обращении к `va_arg` нужно задавать тип ожидаемого фактического параметра. В `va_arg()` предполагается, что параметр такого типа присутствует в вызове, но обычно нет возможности проверить это. Перед выходом из функции, в которой

было обращение к `va_start`, необходимо вызвать `va_end`. Причина в том, что в `va_start()` могут быть такие операции со стеком, из-за которых корректный возврат из функции становится невозможным. В `va_end()` устраняются все нежелательные изменения стека.

Приведение 0 к `(char*)0` необходимо потому, что `sizeof(int)` не обязано совпадать с `sizeof(char*)`. Этот пример демонстрирует все те сложности, с которыми приходится сталкиваться программисту, если он решил обойти контроль типов, используя эллипсис.

4.6.9 Указатель на функцию

Возможны только две операции с функциями: вызов и взятие адреса. Указатель, полученный с помощью последней операции, можно впоследствии использовать для вызова функции. Например:

```
void error(char* p) { /* ... */ }

 void (*efct)(char*);   // указатель на функцию

 void f()
 {
efct = &error;       // efct настроен на функцию error
(*efct)("error");    // вызов error через указатель efct
 }
```

Для вызова функции с помощью указателя (`efct` в нашем примере) надо вначале применить операцию косвенности к указателю - `*efct`. Поскольку приоритет операции вызова `()` выше, чем приоритет косвенности `*`, нельзя писать просто `*efct("error")`. Это будет означать `*(efct("error"))`, что является ошибкой. По той же причине скобки нужны и при описании указателя на функцию. Однако, писать просто `efct("error")` можно, т.к. транслятор понимает, что `efct` является указателем на функцию, и создает команды, делающие вызов нужной функции.

Отметим, что формальные параметры в указателях на функцию описываются так же, как и в обычных функциях. При присваивании

указателю на функцию требуется точное соответствие типа функции и типа присваиваемого значения. Например:

```
void (*pf)(char*);// указатель на void(char*)
void f1(char*); // void(char*);
int f2(char*);  // int(char*);
void f3(int*);  // void(int*);

void f()
{
pf = &f1;     // нормально
pf = &f2;     // ошибка: не тот тип возвращаемого
  // значения
pf = &f3;     // ошибка: не тот тип параметра

(*pf)("asdf");// нормально
(*pf)(1);     // ошибка: не тот тип параметра

int i = (*pf)("qwer");    // ошибка: void присваивается int
}
```

Правила передачи параметров одинаковы и для обычного вызова, и для вызова с помощью указателя.

Часто бывает удобнее обозначить тип указателя на функцию именем, чем все время использовать достаточно сложную запись. Например:

```
typedef int (*SIG_TYP)(int);    // из <signal.h>
  typedef void (SIG_ARG_TYP)(int);
  SIG_TYP signal(int, SIG_ARG_TYP);
```

Также часто бывает полезен массив указателей на функции. Например, можно реализовать систему меню для редактора с вводом, управляемым мышью, используя массив указателей на функции, реализующие команды. Здесь нет возможности подробно описать такой редактор, но дадим самый общий его набросок:

```
typedef void (*PF)();

 PF edit_ops[] = { // команды редактора
```

```
&cut, &paste, &snarf, &search
};

PF file_ops[] = { // управление файлом
&open, &reshape, &close, &write

};
```

Далее надо определить и инициализировать указатели, с помощью которых будут запускаться функции, реализующие выбранные из меню команды. Выбор происходит нажатием клавиши мыши:

```
PF* button2 = edit_ops;
PF* button3 = file_ops;
```

Для настоящей программы редактора надо определить большее число объектов, чтобы описать каждую позицию в меню. Например, необходимо где-то хранить строку, задающую текст, который будет выдаваться для каждой позиции. При работе с системой меню назначение клавиш мыши будет постоянно меняться. Частично эти изменения можно представить как изменения значений указателя, связанного с данной клавишей. Если пользователь выбрал позицию меню, которая определяется, например, как позиция 3 для клавиши 2, то соответствующая команда реализуется вызовом:

```
(*button2[3])();
```

Чтобы полностью оценить мощность конструкции указатель на функцию, стоит попытаться написать программу без нее. Меню можно изменять в динамике, если добавлять новые функции в таблицу команд. Довольно просто создавать в динамике и новые меню.

Указатели на функции помогают реализовать полиморфические подпрограммы, т.е. такие подпрограммы, которые можно применять к объектам различных типов:

```
typedef int (*CFT)(void*,void*);

void sort(void* base, unsigned n, unsigned int sz, CFT cmp)
/*
```

```
  Сортировка вектора "base" из n элементов
  в возрастающем порядке;
  используется функция сравнения, на которую указывает cmp.
  Размер элементов равен "sz".

  Алгоритм очень неэффективный: сортировка пузырьковым методом
*/
{
 for (int i=0; i<n-1; i++)
     for (int j=n-1; i<j; j--) {
 char* pj = (char*)base+j*sz;  // b[j]
 char* pj1 = pj - sz;// b[j-1]
 if ((*cmp)(pj,pj1) < 0) {
 // поменять местами b[j] и b[j-1]
for (int k = 0; k<sz; k++) {
    char temp = pj[k];
    pj[k] = pj1[k];
    pj1[k] = temp;
}
}
}
 }
```

В подпрограмме `sort` неизвестен тип сортируемых объектов; известно только их число (размер массива), размер каждого элемента и функция, которая может сравнивать объекты. Мы выбрали для функции `sort()` такой же заголовок, как у `qsort()` - стандартной функции сортировки из библиотеки C. Эту функцию используют настоящие программы. Покажем, как с помощью `sort()` можно отсортировать таблицу с такой структурой:

```
struct user {
   char* name;     // имя
   char* id;       // пароль
   int dept;       // отдел
};

typedef user* Puser;
```

```
user heads[] = {
    "Ritchie D.M.",     "dmr",   11271,
    "Sethi R.","ravi",  11272,
    "SZYmanski T.G.",   "tgs",   11273,
    "Schryer N.L.",     "nls",   11274,
    "Schryer N.L.",     "nls",   11275
    "Kernighan B.W.",   "bwk",   11276
};

void print_id(Puser v, int n)
{
  for (int i=0; i<n; i++)
      cout << v[i].name << '\t'
<< v[i].id << '\t'
<< v[i].dept << '\n';
}
```

Чтобы иметь возможность сортировать, нужно вначале определить подходящие функции сравнения. Функция сравнения должна возвращать отрицательное число, если ее первый параметр меньше второго, нуль, если они равны, и положительное число в противном случае:

```
int cmp1(const void* p, const void* q)
// сравнение строк, содержащих имена
{
  return strcmp(Puser(p)->name, Puser(q)->name);
}

int cmp2(const void* p, const void* q)
// сравнение номеров разделов
{
  return Puser(p)->dept - Puser(q)->dept;
}
```

Следующая программа сортирует и печатает результат:

```
int main()
{
```

```
sort(heads,6,sizeof(user),&cmp1);
print_id(heads,6);    // в алфавитном порядке
cout << "\n";
sort(heads,6,sizeof(user),&cmp2);
print_id(heads,6);    // по номерам отделов
}
```

Допустима операция взятия адреса и для функции-подстановки, и для перегруженной функции.

Отметим, что неявное преобразование указателя на что-то в указатель типа void* не выполняется для параметра функции, вызываемой через указатель на нее. Поэтому функцию

```
int cmp3(const mytype*, const mytype*);
```

нельзя использовать в качестве параметра для `sort()`. Поступив иначе, мы нарушаем заданное в описании условие, что `cmp3()` должна вызываться с параметрами типа `mytype*`. Если вы специально хотите нарушить это условие, то должны использовать явное преобразование типа.

4.7 Макросредства

В C++ макросредства играют гораздо меньшую роль, чем в C. Можно даже дать такой совет: используйте макроопределения только тогда, когда не можете без них обойтись. Вообще говоря, считается, что практически каждое появление макроимени является свидетельством некоторых недостатков языка, программы или программиста. Макросредства создают определенные трудности для работы служебных системных программ, поскольку они перерабатывают программный текст еще до трансляции. Поэтому, если ваша программа использует макросредства, то сервис, предоставляемый такими программами, как отладчик, профилировщик, программа перекрестных ссылок, будет для нее неполным. Если все-таки вы решите использовать макрокоманды, то вначале тщательно изучите описание препроцессора C++ в вашем справочном руководстве и не старайтесь быть слишком умным.

Простое макроопределение имеет вид:

```
#define имя  остаток-строки
```

В тексте программы лексема имя заменяется на остаток-строки. Например,

```
объект = имя
```

будет заменено на

```
объект = остаток-строки
```

Макроопределение может иметь параметры. Например:

```
#define mac(a,b)  argument1: a argument2: b
```

В макровызове `mac` должны быть заданы две строки, представляющие параметры. При подстановке они заменят `a` и `b` в макроопределении `mac()`. Поэтому строка

```
expanded = mac(foo bar, yuk yuk)
```

при подстановке преобразуется в

```
expanded = argument1: foo bar argument2: yuk yuk
```

Макроимена нельзя перегружать. Рекурсивные макровызовы ставят перед препроцессором слишком сложную задачу:

```
// ошибка:
#define print(a,b) cout<<(a)<<(b)
#define print(a,b,c)  cout<<(a)<<(b)<<(c)

// слишком сложно:
#define fac(n) (n>1) ?n*fac(n-1) :1
```

Препроцессор работает со строками и практически ничего не знает о синтаксисе C++, типах языка и областях видимости. Транслятор имеет дело только с уже раскрытым макроопределением, поэтому ошибка в нем может диагностироваться уже после подстановки, а не при

определении макроимени. В результате появляются довольно путанные сообщения об ошибках.

Допустимы такие макроопределения:

```
#define Case break;case
#define forever for(;;)
```

А вот совершенно излишние макроопределения:

```
#define PI 3.141593
#define BEGIN {
#define END }
```

Следующие макроопределения могут привести к ошибкам:

```
#define SQUARE(a) a*a
#define INCR_xx (xx)++
#define DISP = 4
```

Чтобы убедиться в этом, достаточно попробовать сделать подстановку в таком примере:

```
int xx = 0;// глобальный счетчик

void f() {
int xx = 0;// локальная переменная
xx = SQUARE(xx+2); // xx = xx +2*xx+2;
INCR_xx;// увеличивается локальная переменная xx
if (a-DISP==b) {   // a-=4==b
// ...
}
}
```

При ссылке на глобальные имена в макроопределении используйте операцию разрешения области видимости, и всюду, где это возможно, заключайте имя параметра макроопределения в скобки. Например:

```
#define MIN(a,b) (((a)<(b))?(a):(b))
```

Если макроопределение достаточно сложное, и требуется комментарий

к нему, то разумнее написать комментарий вида /* */, поскольку в реализации C++ может использоваться препроцессор C, который не распознает комментарии вида //. Например:

```
#define m2(a) something(a)   /* глубокомысленный комментарий */
```

С помощью макросредств можно создать свой собственный язык, правда, скорее всего, он будет непонятен другим. Кроме того, препроцессор C предоставляет довольно слабые макросредства. Если ваша задача нетривиальна, вы, скорее всего, обнаружите, что решить ее с помощью этих средств либо невозможно, либо чрезвычайно трудно. В качестве альтернативы традиционному использованию макросредств в язык введены конструкции `const`, `inline` и шаблоны типов. Например:

```
const int answer = 42;
template<class T>
inline T min(T a, T b) { return (a<b)?a:b; }
```

Классы

В этой лекции описываются возможности определения новых типов, для которых доступ к данным ограничен заданным множеством функций, осуществляющих его. Объясняется, как можно использовать члены структуры данных, как ее защищать, инициализировать и, наконец, уничтожать. В примерах приведены простые классы для управления таблицей имен, работы со стеком, множеством и реализации дискриминирующего (т.е. надежного) объединения. Следующие три лекции завершают описание возможностей C++ для построения новых типов, и в них содержится больше интересных примеров.

5.1 Введение и краткий обзор

Понятие класса, которому посвящена эта и три следующих лекций, служит в C++ для того, чтобы дать программисту инструмент построения новых типов. Ими пользоваться не менее удобно, чем встроенными. В идеале использование определенного пользователем типа не должно отличаться от использования встроенных типов. Различия возможны только в способе построения.

Тип есть вполне конкретное представление некоторого понятия. Например, в C++ тип float с операциями +, -, * и т.д. является хотя и ограниченным, но конкретным представлением математического понятия вещественного числа. Новый тип создается для того, чтобы стать специальным и конкретным представлением понятия, которое не находит прямого и естественного отражения среди встроенных типов. Например, в программе из области телефонной связи можно ввести тип `trunk_module` (линия-связи), в видеоигре - тип `explosion` (взрыв), а в программе, обрабатывающей текст, - тип `list_of_paragraphs` (список-параграфов). Обычно проще понимать и изменять программу, в которой типы хорошо представляют используемые в задаче понятия. Удачно подобранное множество пользовательских типов делает программу более ясной. Оно позволяет транслятору обнаруживать недопустимое использование объектов, которое в противном случае останется невыявленным до отладки программы.

Главное в определении нового типа - это отделить несущественные детали реализации (например, расположение данных в объекте нового типа) от тех его характеристик, которые существенны для правильного его использования (например, полный список функций, имеющих доступ к данным). Такое разделение обеспечивается тем, что вся работа со структурой данных и внутрение, служебные операции над нею доступны только через специальный интерфейс (через "одно горло").

Лекция состоит из четырех частей:

§ 5.2 Классы и члены. Здесь вводится основное понятие пользовательского типа, называемого классом. Доступ к объектам класса может ограничиваться множеством функций, описания которых входят в описание класса. Эти функции называются функциями-членами и друзьями. Для создания объектов класса используются специальные функции-члены, называемые конструкторами. Можно описать специальную функцию-член для удаления объектов класса при его уничтожении. Такая функция называется деструктором.

§ 5.3 Интерфейсы и реализации. Здесь приводятся два примера разработки, реализации и использования классов.

§ 5.4 Дополнительные свойства классов. Здесь приводится много дополнительных подробностей о классах. Показано, как функции, не являющейся членом класса, предоставить доступ к его частной части. Такую функцию называют другом класса. Вводятся понятия статических членов класса и указателей на члены класса. Здесь же показано, как определить дискриминирующее объединение.

§ 5.5 Конструкторы и деструкторы. Объект может создаваться как автоматический, статический или как объект в свободной памяти. Кроме того, объект может быть членом некоторого агрегата (массива или другого класса), который тоже можно размещать одним из этих трех способов. Подробно объясняется использование конструкторов и деструкторов, описывается применение определяемых пользователем функций размещения в свободной памяти и функций освобождения памяти.

5.2 Классы и члены

Класс - это пользовательский тип. Этот раздел знакомит с основными средствами определения класса, создания его объектов, работы с такими объектами и, наконец, удаления этих объектов после использования.

5.2.1 Функции-члены

Посмотрим, как можно представить в языке понятие даты, используя для этого тип структуры и набор функций, работающих с переменными этого типа:

```
struct date { int month, day, year; };
date today;
void set_date(date*, int, int, int);
void next_date(date*);
void print_date(const date*);
// ...
```

Никакой явной связи между функциями и структурой `date` нет. Ее можно установить, если описать функции как члены структуры:

```
struct date {
int month, day, year;

void set(int, int, int);
void get(int*, int* int*);
void next();
void print();
};
```

Описанные таким образом функции называются функциями-членами. Их можно вызывать только через переменные соответствующего типа, используя стандартную запись обращения к члену структуры:

```
date today;
date my_birthday;
```

```
void f()
{
        my_birthday.set(30,12,1950);
        today.set(1,18,1991);

        my_birthday.print();
        today.next();
}
```

Поскольку разные структуры могут иметь функции-члены с одинаковыми именами, при определении функции-члена нужно указывать имя структуры:

```
void date::next()
{
  if (++day > 28 ) {
    // здесь сложный вариант
  }
}
```

В теле функции-члена имена членов можно использовать без указания имени объекта. В таком случае имя относится к члену того объекта, для которого была вызвана функция.

5.2.2 Классы

Мы определили несколько функций для работы со структурой date, но из ее описания не следует, что это единственные функции, которые предоставляют доступ к объектам типа `date`. Можно установить такое ограничение, описав класс вместо структуры:

```
class date {
  int month, day, year;
public:
  void set(int, int, int);
  void get(int*, int*, int*);
  void next();
```

```
    void print()
};
```

Служебное слово public (общий) разбивает описание класса на две части. Имена, описанные в первой частной (private) части класса, могут использоваться только в функциях-членах. Вторая - общая часть - представляет собой интерфейс с объектами класса. Поэтому структура - это такой класс, в котором по определению все члены являются общими. Функции-члены класса определяются и используются точно так же, как было показано в предыдущем разделе:

```
void date::print()     // печать даты  в принятом в США виде
{
  cout << month << '/' << day << '/' << year ;
}
```

Однако от функций не членов частные члены класса date уже ограждены:

```
void backdate()
{
  today.day--;    // ошибка
}
```

Есть ряд преимуществ в том, что доступ к структуре данных ограничен явно указанным списком функций. Любая ошибка в дате (например, December, 36, 1985) могла быть внесена только функцией-членом, поэтому первая стадия отладки - локализация ошибки - происходит даже до первого пуска программы. Это только частный случай общего правила: любое изменение в поведении типа `date` может и должно вызываться изменениями в его членах. Другое преимущество в том, что потенциальному пользователю класса для работы с ним достаточно знать только определения функций-членов.

Защита частных данных основывается только на ограничении использования имен членов класса. Поэтому ее можно обойти с помощью манипуляций с адресами или явных преобразований типа, но это уже можно считать мошенничеством.

5.2.3 Ссылка на себя

В функции-члене можно непосредственно использовать имена членов того объекта, для которого она была вызвана:

```
class X {
  int m;
public:
  int readm() { return m; }
};

void f(X aa, X bb)
{
  int a = aa.readm();
  int b = bb.readm();
  // ...
}
```

При первом вызове `readm()` `m` обозначает `aa.m`, а при втором - `bb.m`.

У функции-члена есть дополнительный скрытый параметр, являющийся указателем на объект, для которого вызывалась функция. Можно явно использовать этот скрытый параметр под именем `this`. Считается, что в каждой функции-члене класса `X` указатель `this` описан неявно как

```
X *const this;
```

и инициализируется, чтобы указывать на объект, для которого функция-член вызывалась. Этот указатель нельзя изменять, поскольку он постоянный (`*const`). Явно описать его тоже нельзя, т.к. `this` - это служебное слово. Можно дать эквивалентное описание класса `X`:

```
class X {
   int m;
public:
   int readm() { return this->m; }
};
```

Для обращения к членам использовать `this` излишне. В основном `this` используется в функциях-членах, непосредственно работающих с указателями. Типичный пример - функция, которая вставляет элемент в список с двойной связью:

```
class dlink {
 dlink* pre;  // указатель на предыдущий элемент
 dlink* suc;  // указатель на следующий элемент
        public:
 void append(dlink*);
 // ...
        };

        void dlink::append(dlink* p)
        {
p->suc = suc;    // т.е. p->suc = this->suc
p->pre = this;   // явное использование "this"
suc->pre = p;    // т.е. this->suc->pre = p
suc = p;         // т.е. this->suc = p
        }

        dlink* list_head;

        void f(dlink* a, dlink* b)
        {
// ...
list_head->append(a);
list_head->append(b);
        }
```

Списки с такой общей структурой служат фундаментом списочных классов, описываемых в лекции 8. Чтобы присоединить звено к списку, нужно изменить объекты, на которые настроены указатели `this`, `pre` и `suc`. Все они имеют тип `dlink`, поэтому функция-член `dlink::append()` имеет к ним доступ. Защищаемой единицей в C ++ является класс, а не отдельный объект класса.

Можно описать функцию-член таким образом, что объект, для которого она вызывается, будет доступен ей только по чтению. Тот факт, что

функция не будет изменять объект, для которого она вызывается (т.е. `this` *), обозначается служебным словом `const` в конце списка параметров:

```
class X {
int m;
 public:
readme() const { return m; }
writeme(int i) { m = i; }
 };
```

Функцию-член со спецификацией `const` можно вызывать для постоянных объектов, а функцию-член без такой спецификации - нельзя:

```
void f(X& mutable, const X& constant)
 {
   mutable.readme();    // нормально
   mutable.writeme(7);  // нормально
   constant.readme();   // нормально
   constant.writeme(7); // ошибка
 }
```

В этом примере разумный транслятор смог бы обнаружить, что функция `X::writeme()` пытается изменить постоянный объект. Однако, это непростая задача для транслятора. Из-за раздельной трансляции он в общем случае не может гарантировать "постоянство" объекта, если нет соответствующего описания со спецификацией `const`. Например, определения `readme()` и `writeme()` могли быть в другом файле:

```
class X {
int m;
 public:
readme() const;
writeme(int i);
 };
```

В таком случае описание `readme()` со спецификацией `const` существенно.

Тип указателя `this` в постоянной функции-члене класса `X` есть `const X *const`. Это значит, что без явного приведения с помощью `this` нельзя изменить значение объекта:

```
class X {
int m;
 public:
// ...
void implicit_cheat() const { m++; }  // ошибка
void explicit_cheat() const { ((X*)this)->m++; }
    // нормально
};
```

Отбросить спецификацию `const` можно потому, что понятие "постоянства" объекта имеет два значения. Первое, называемое "физическим постоянством" состоит в том, что объект хранится в защищенной от записи памяти. Второе, называемое "логическим постоянством" заключается в том, что объект выступает как постоянный (неизменяемый) по отношению к пользователям. Операция над логически постоянным объектом может изменить часть данных объекта, если при этом не нарушается его постоянство с точки зрения пользователя. Операциями, не нарушающими логическое постоянство объекта, могут быть буферизация значений, ведение статистики, изменение переменных-счетчиков в постоянных функциях-членах.

Логического постоянства можно достигнуть приведением, удаляющим спецификацию `const`:

```
class calculator1 {
   int cache_val;
   int cache_arg;
   // ...
public:
   int compute(int i) const;
   // ...
};

int calculator1::compute(int i) const
{
```

```
  if (i == cache_arg) return cache_val;
  // нелучший способ
  ((calculator1*)this)->cache_arg = i;
  ((calculator1*)this)->cache_val = val;
  return val;
 }
```

Этого же результата можно достичь, используя указатель на данные без `const`:

```
struct cache {
int val;
int arg;
};

class calculator2 {
cache* p;
// ...
public:
int compute(int i) const;
// ...
};

int calculator2::compute(int i) const
{
 if (i == p->arg) return p->val;
 // нелучший способ
 p->arg = i;
 p->val = val;
 return val;
}
```

Отметим, что `const` нужно указывать как в описании, так и в определении постоянной функции-члена. Физическое постоянство обеспечивается помещением объекта в защищенную по записи память, только если в классе нет конструктора.

5.2.4 Инициализация

Инициализация объектов класса с помощью таких функций как `set_date()` - неэлегантное и чреватое ошибками решение. Поскольку явно не было указано, что объект требует инициализации, программист может либо забыть это сделать, либо сделать дважды, что может привести к столь же катастрофическим последствиям. Лучше дать программисту возможность описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значение данного типа, она называется конструктором. Эту функцию легко распознать - она имеет то же имя, что и ее класс:

```
class date {
// ...
date(int, int, int);
 };
```

Если в классе есть конструктор, все объекты этого класса будут проинициализированы. Если конструктору требуются параметры, их надо указывать:

```
date today = date(23,6,1983);
 date xmas(25,12,0);  // краткая форма
 date my_birthday;    // неправильно, нужен инициализатор
```

Часто бывает удобно указать несколько способов инициализации объекта. Для этого нужно описать несколько конструкторов:

```
class date {
int month, day, year;
 public:
// ...
date(int, int, int);  // день, месяц, год
date(int, int);       // день, месяц и текущий год
date(int); // день и текущие год и месяц
date();// стандартное значение: текущая дата
date(const char*);    // дата в строковом представлении
 };
```

Параметры конструкторов подчиняются тем же правилам о типах параметров, что и все остальные функции (§ 4.6.6). Пока конструкторы

достаточно различаются по типам своих параметров, транслятор способен правильно выбрать конструктор:

```
date today(4);
date july4("July 4, 1983");
date guy("5 Nov");
date now;       // инициализация стандартным значением
```

Размножение конструкторов в примере с `date` типично. При разработке класса всегда есть соблазн добавить еще одну возможность, - а вдруг она кому-нибудь пригодится. Чтобы определить действительно нужные возможности, надо поразмышлять, но зато в результате, как правило, получается более компактная и понятная программа. Сократить число сходных функций можно с помощью стандартного значения параметра. В примере с `date` для каждого параметра можно задать стандартное значение, что означает: "взять значение из текущей даты".

```
class date {
  int month, day, year;
public:
  // ...
  date(int d =0, int m =0, int y=0);
  // ...
};

date::date(int d, int m, int y)
{
  day = d ? d : today.day;
  month = m ? m : today.month;
  year = y ? y : today.year;
  // проверка правильности даты
  // ...
}
```

Когда используется стандартное значение параметра, оно должно отличаться от всех допустимых значений параметра. В случае месяца и дня очевидно, что при значении нуль - это так, но неочевидно, что нуль подходит для значения года. К счастью, в европейском календаре нет

нулевого года, т.к. сразу после 1 г. до р.х. (`year==-1`) идет `1 г. р.х.` (`year==1`). Однако для обычной программы это, возможно, слишком тонкий момент.

Объект класса без конструктора может инициализироваться присваиванием ему другого объекта этого же класса. Это не запрещено и в том случае, когда конструкторы описаны:

```
date d = today;  // инициализация присваиванием
```

На самом деле, имеется стандартный конструктор копирования, определенный как поэлементное копирование объектов одного класса. Если такой конструктор для класса `X` не нужен, можно переопределить его как конструктор копирования `X::X(const X&)`.

5.2.5 Удаление

Пользовательские типы чаще имеют, чем не имеют, конструкторы, которые проводят надлежащую инициализацию. Для многих типов требуется и обратная операция - деструктор, гарантирующая правильное удаление объектов этого типа. Деструктор класса `X` обозначается `~X` ("дополнение конструктора"). В частности, для многих классов используется свободная память (см. § 3.2.6), выделяемая конструктором и освобождаемая деструктором. Вот, например, традиционное определение типа стек, из которого для краткости полностью выброшена обработка ошибок:

```
class char_stack {
int size;
char* top;
char* s;
 public:
char_stack(int sz) { top=s=new char[size=sz]; }
~char_stack()  { delete[] s; }  // деструктор
void push(char c) { *top++ = c; }
char pop()   { return *--top; }
};
```

Когда объект типа `char_stack` выходит из текущей области видимости, вызывается деструктор:

```
void f()
{
  char_stack s1(100);
  char_stack s2(200);
  s1.push('a');
  s2.push(s1.pop());
  char ch = s2.pop();
  cout << ch << '\n';
}
```

Когда начинает выполняться `f()`, вызывается конструктор `char_stack`, который размещает массив из 100 символов `s1` и массив из 200 символов `s2`. При возврате из `f()` память, которая была занята обоими массивами, будет освобождена.

5.2.6 Подстановка

Программирование с классами предполагает, что в программе появится множество маленьких функций. По сути, всюду, где в программе с традиционной организацией стояло бы обычное обращение к структуре данных, используется функция. То, что было соглашением, стало стандартом, проверяемым транслятором. В результате программа может стать крайне неэффективной. Хотя вызов функции в C++ и не столь дорогостоящая операция по сравнению с другими языками, все-таки цена ее много выше, чем у пары обращений к памяти, составляющих тело тривиальной функции.

Преодолеть эту трудность помогают функции-подстановки (inline). Если в описании класса функция-член определена, а не только описана, то она считается подстановкой. Это значит, например, что при трансляции функций, использующих `char_stack` из предыдущего примера, не будет использоваться никаких операций вызова функций, кроме реализации операций вывода! Другими словами, при разработке такого класса не нужно принимать во внимание затраты на вызов функций. Любое, даже самое маленькое действие, можно смело определять как

функцию без потери эффективности. Это замечание снимает наиболее часто приводимый довод в пользу общих членов данных.

Функцию-член можно описать со спецификацией inline и вне описания класса:

```
class char_stack {
  int size;
  char* top;
  char* s;
public:
  char pop();
  // ...
};

inline char char_stack::pop()
{
 return *--top;
}
```

Отметим, что недопустимо описывать разные определения функции-члена, являющейся подстановкой, в различных исходных файлах. Это нарушило бы понятие о классе как о цельном типе.

5.3 Интерфейсы и реализации

Что представляет собой хороший класс? Это нечто, обладающее хорошо определенным множеством операций. Нечто, рассматриваемое как "черный ящик", управлять которым можно только посредством этих операций. Нечто, чье фактическое представление можно изменить любым мыслимым способом, но не изменяя при этом способа использования операций. Нечто, что может потребоваться в нескольких экземплярах.

Очевидные примеры хороших классов дают контейнеры разных видов: таблицы, множества, списки, вектора, словари и т.д. Такой класс имеет операцию занесения в контейнер. Обычно имеется и операция проверки: был ли данный член занесен в контейнер? Могут быть операции упорядочивания всех членов и просмотра их в определенном

порядке. Наконец, может быть операция удаления члена. Обычно контейнерные классы имеют конструкторы и деструкторы.

5.3.1 Альтернативные реализации

Пока описание общей части класса и функций-членов остается неизменным, можно, не влияя на пользователей класса, менять его реализацию. В подтверждение этого рассмотрим таблицу имен из программы калькулятора, приведенной в лекции 3. Структура ее такова:

```
struct name {
   char* string;
   name* next;
   double value;
 };
```

А вот вариант класса `table` (таблица имен):

```
// файл table.h
 class table {
name* tbl;
 public:
table() { tbl = 0; }

name* look(char*, int = 0);
name* insert(char* s) { return look(s,1); }
 };
```

Эта таблица отличается от определенной в лекции 3 тем, что это настоящий тип. Можно описать несколько таблиц, завести указатель на таблицу и т.д. Например:

```
#include "table.h"

 table globals;
 table keywords;
 table* locals;
```

```
main()
{
  locals = new table;
  // ...
}
```

Приведем реализацию функции `table::look()`, в которой используется линейный поиск в списке имен таблицы:

```
#include <string.h>

name* table::look(char* p, int ins)
{
  for (name* n = tbl; n; n=n->next)
  if (strcmp(p,n->string) == 0) return n;
  if (ins == 0) error("имя не найдено");

  name* nn = new name;
  nn->string = new char[strlen(p)+1];
  strcpy(nn->string,p);
  nn->value = 1;
  nn->next = tbl;
  tbl = nn;
  return nn;
}
```

Теперь усовершенствуем класс `table` так, чтобы поиск имени шел по ключу (хэш-функции от имени), как это и было сделано в примере с калькулятором. Сделать это труднее, если соблюдать ограничение, требующее, чтобы не все программы, использующие приведенную версию класса `table`, надо было изменять:

```
class table {
 name** tbl;
 int size;
  public:
 table(int sz = 15);
 ~table();
```

```
 name* look(char*, int = 0);
 name* insert(char* s) { return look(s,1); }
  };
```

Изменения в структуре данных и конструкторе произошли потому, что для хэширования таблица должна иметь определенный размер. Задание конструктора со стандартным значением параметра гарантирует, что старые программы, в которых не использовался размер таблицы, останутся верными. Стандартные значения параметров полезны в таких случаях, когда нужно изменить класс, не влияя на программы пользователей класса. Теперь конструктор и деструктор создают и уничтожают хэшированные таблицы:

```
table::table(int sz)
  {
if (sz < 0) error("размер таблицы отрицателен");
tbl = new name*[size = sz];
for ( int i = 0; i<sz; i++) tbl[i] = 0;
  }

  table::~table()
  {
for (int i = 0; i<size; i++) {
    name* nx;
    for (name* n = tbl[i]; n; n=nx) {
        nx = n->next;
        delete n->string;
        delete n;
    }
}
delete tbl;
}
```

Описав деструктор для класса `name`, можно получить более ясный и простой вариант `table::~table()`. Функция поиска практически совпадает с приведенной в примере калькулятора:

```
name* table::look(const char* p, int ins)
  {
```

```
int ii = 0;
char* pp = p;
while (*pp) ii = ii<<1 ^ *pp++;
if (ii < 0) ii = -ii;
ii %= size;

for (name* n=tbl[ii]; n; n=n->next)
    if (strcmp(p,n->string) == 0) return n;

name* nn = new name;
nn->string = new char[strlen(p)+1];
strcpy(nn->string,p);
nn->value = 1;
nn->next = tbl[ii];
tbl[ii] = nn;
return nn;
  }
```

Очевидно, что функции-члены класса должны перетранслироваться всякий раз, когда в описание класса вносится какое-либо изменение. В идеале такое изменение никак не должно отражаться на пользователях класса. К сожалению, обычно бывает не так. Для размещения переменной, имеющей тип класса, транслятор должен знать размер объекта класса. Если размер объекта изменится, нужно перетранслировать файлы, в которых использовался класс. Можно написать системную программу (и она даже уже написана), которая будет определять минимальное множество файлов, подлежащих перетрансляции после изменения класса. Но такая программа еще не получила широкого распространения.

Возможен вопрос: почему C++ был спроектирован таким образом, что после изменения частной части класса требуется перетрансляция программ пользователя? Почему вообще частная часть класса присутствует в описании класса? Иными словами, почему описания частных членов присутствуют в заголовочных файлах, доступных пользователю, если все равно недоступны для него в программе? Ответ один - эффективность. Во многих системах программирования процесс трансляции и последовательность команд, производящая вызов функции, будет проще, если размер автоматических (т.е. размещаемых в

стеке) объектов известен на стадии трансляции.

Можно не знать определения всего класса, если представлять каждый объект как указатель на "настоящий" объект. Это позволяет решить задачу, поскольку все указатели будут иметь одинаковый размер, а размещение настоящих объектов будет проводиться только в одном файле, в котором доступны частные части классов. Однако, такое решение приводит к дополнительному расходу памяти на каждый объект и дополнительному обращению к памяти при каждом использовании члена. Еще хуже, что каждый вызов функции с автоматическим объектом класса требует вызовов функций выделения и освобождения памяти. К тому же становится невозможной реализация подстановкой функций-членов, работающих с частными членами класса. Наконец, такое изменение сделает невозможным связывание программ на C++ и на C, поскольку транслятор C будет по другому обрабатывать структуры (struct). Поэтому такое решение было сочтено неприемлемым для C++.

С другой стороны, C++ предоставляет средство для создания абстрактных типов, в которых связь между интерфейсом пользователя и реализацией довольно слабая. В лекции 6 вводятся производные классы и описываются абстрактные базовые классы. Цель этого - дать возможность определять пользовательские типы столь же эффективные и конкретные, как и стандартные, и дать основные средства определения более гибких вариантов типов, которые могут оказаться и не столь эффективными.

5.3.2 Законченный пример класса

Программирование без упрятывания данных (в расчете на структуры) требует меньшего предварительного обдумывания задачи, чем программирование с упрятыванием данных (в расчете на классы). Структуру можно определить не очень задумываясь о том, как ее будут использовать. Когда определяется класс, внимание концентрируется на том, чтобы обеспечить для нового типа полный набор операций. Это важное смещение акцента в проектировании программ. Обычно время, затраченное на разработку нового типа, многократно окупается в процессе отладки и развития программы.

Вот пример законченного определения типа intset, представляющего понятие "множество целых":

```
class intset {
  int cursize, maxsize;
  int  *x;
public:
  intset(int m, int n);    // не более m целых из 1..n
  ~intset();

  int member(int t) const; // является ли t членом?
  void insert(int t);      // добавить к множеству t

  void start(int& i) const { i = 0; }
  int ok(int& i) const    { return i<cursize; }
  int next(int& i) const  { return x[i++]; }
        };
```

Для проверки этого класса вначале создадим, а затем распечатаем множество случайных целых чисел. Это простое множество целых можно использовать для проверки, есть ли повторения в их последовательности. Но для большинства задач нужен, конечно, более развитый тип множества. Как всегда возможны ошибки, поэтому нужна функция:

```
#include <iostream.h>

void error(const char *s)
   {
        cerr << "set: " << s << '\n';
        exit(1);
    }
```

Класс `intset` используется в функции `main()`, для которой должно быть задано два параметра: первый определяет число создаваемых случайных чисел, а второй - диапазон их значений:

```
int main(int argc, char* argv[])
        {
 if (argc != 3) error("нужно задавать два параметра");
```

```
int count = 0;
int m = atoi(argv[1]);    // число элементов множества
int n = atoi(argv[2]);    // из диапазона 1..n
intset s(m,n);

while (count<m) {
  int t = randint(n);
  if (s.member(t)==0) {
  s.insert(t);
  count++;
  }
}

print_in_order(&s);
        }
```

Значение счетчика параметров программы `argc` равно 3, хотя программа имеет только два параметра. Дело в том, что в `argv[0]` всегда передается дополнительный параметр, содержащий имя программы.

Функция

```
extern "C" int atoi(const char*)
```

является стандартной библиотечной функцией, преобразующей целое из строкового представления во внутреннюю двоичную форму. Как обычно, если вы не хотите иметь такое описание в своей программе, то вам надо включить в нее соответствующий заголовочный файл, содержащий описания стандартных библиотечных функций. Случайные числа генерируются с помощью стандартной функции `rand`:

```
extern "C" int rand();  // будьте осторожны:
      // числа не совсем случайные
int randint(int u)      // диапазон 1..u
{
  int r = rand();
  if (r < 0) r = -r;
  return 1 + r%u;
```

```
}
```

Подробности реализации класса мало интересны для пользователя, но в любом случае будут использоваться функции-члены. Конструктор размещает массив целых с размером, равным заданному максимальному размеру множества, а деструктор удаляет этот массив:

```
intset::intset(int m, int n)  // не более m целых в 1..n
        {
 if (m<1 || n<m) error("недопустимый размер intset");
 cursize = 0;
 maxsize = m;
 x = new int[maxsize];
        }

        intset::~intset()
        {
 delete x;
        }
```

Целые добавляются таким образом, что они хранятся во множестве в возрастающем порядке:

```
void intset::insert(int t)
        {
 if (++cursize > maxsize) error("слишком много элементов");
 int i = cursize-1;
 x[i] = t;

 while (i>0 && x[i-1]>x[i]) {
int t = x[i];     // поменять местами x[i] и x[i-1]
x[i] = x[i-1];
x[i-1] = t;
i--;
 }
        }
```

Чтобы найти элемент, используется простой двоичный поиск:

```
int intset::member(int t) const  // двоичный поиск
```

```
{
 int l = 0;
 int u = cursize-1;

 while (l <= u) {
  int m = (l+u)/2;
  if (t < x[m])
 u = m-1;
  else if (t > x[m])
 l = m+1;
  else
 return 1;   // найден
 }
 return 0;        // не найден
}
```

Наконец, нужно предоставить пользователю набор операций, с помощью которых он мог бы организовать итерацию по множеству в некотором порядке (ведь порядок, используемый в представлении `intset`, от него скрыт). Множество по своей сути не является внутренне упорядоченным, и нельзя позволить просто выбирать элементы массива (а вдруг завтра `intset` будет реализовано в виде связанного списка?).

Пользователь получает три функции: `start()` - для инициализации итерации, `ok()` - для проверки, есть ли следующий элемент, и `next()` - для получения следующего элемента:

```
class intset {
  // ...
  void start(int& i) const   { i = 0; }
  int ok(int& i) const       { return i<cursize; }
  int next(int& i) const     { return x[i++]; }
};
```

Чтобы обеспечить совместную работу этих трех операций, надо запоминать тот элемент, на котором остановилась итерация. Для этого пользователь должен задавать целый параметр. Поскольку наше представление множества упорядоченное, реализация этих операций

тривиальна. Теперь можно определить функцию `print_in_order`:

```
void print_in_order(intset* set)
{
  int var;
  set->start(var);
  while (set->ok(var)) cout << set->next(var) << '\n';
}
```

5.4 Еще о классах

В этом разделе описаны дополнительные свойства класса. Описан способ обеспечить доступ к частным членам в функциях, не являющихся членами (§ 5.4.1). Описано, как разрешить коллизии имен членов (§ 5.4.2) и как сделать описания классов вложенными (§ 5.4.3), но при этом избежать нежелательной вложенности (§ 5.4.4). Вводится понятие статических членов (static), которые используются для представления операций и данных, относящихся к самому классу, а не к отдельным его объектам (§ 5.4.5). Раздел завершается примером, показывающим, как можно построить дискриминирующее (надежное) объединение (§ 5.4.6).

5.4.1 Друзья

Пусть определены два класса: `vector` (вектор) и `matrix` (матрица). Каждый из них скрывает свое представление, но дает полный набор операций для работы с объектами его типа. Допустим, надо определить функцию, умножающую матрицу на вектор. Для простоты предположим, что вектор имеет четыре элемента с индексами от 0 до 3, а в матрице четыре вектора тоже с индексами от 0 до 3. Доступ к элементам вектора обеспечивается функцией `elem()`, и аналогичная функция есть для матрицы. Можно определить глобальную функцию `multiply` (умножить) следующим образом:

```
vector multiply(const matrix& m, const vector& v);
{
```

```
    vector r;
    for (int i = 0; i<3; i++) { // r[i] = m[i] * v;
    r.elem(i) = 0;
    for (int j = 0; j<3; j++)
        r.elem(i) +=m.elem(i,j) * v.elem(j);
    }
    return r;
}
```

Это вполне естественное решение, но оно может оказаться очень неэффективным. При каждом вызове `multiply()` функция `elem()` будет вызываться `4*(1+4*3)` раз. Если в `elem()` проводится настоящий контроль границ массива, то на такой контроль будет потрачено значительно больше времени, чем на выполнение самой функции, и в результате она окажется непригодной для пользователей. С другой стороны, если `elem()` есть некий специальный вариант доступа без контроля, то тем самым мы засоряем интерфейс с вектором и матрицей особой функцией доступа, которая нужна только для обхода контроля.

Если можно было бы сделать `multiply` членом обоих классов `vector` и `matrix`, мы могли бы обойтись без контроля индекса при обращении к элементу матрицы, но в то же время не вводить специальной функции `elem()`. Однако, функция не может быть членом двух классов. Надо иметь в языке возможность предоставлять функции, не являющейся членом, право доступа к частным членам класса. Функция - не член класса, - имеющая доступ к его закрытой части, называется другом этого класса. Функция может стать другом класса, если в его описании она описана как `friend` (друг). Например:

```
class matrix;

class vector {
  float v[4];
  // ...
  friend vector multiply(const matrix&, const vector&);
};

class matrix {
```

```
    vector v[4];
    // ...
    friend vector multiply(const matrix&, const vector&);
};
```

Функция-друг не имеет никаких особенностей, за исключением права доступа к закрытой части класса. В частности, в такой функции нельзя использовать указатель `this`, если только она действительно не является членом класса. Описание `friend` является настоящим описанием. Оно вводит имя функции в область видимости класса, в котором она была описана, и при этом происходят обычные проверки на наличие других описаний такого же имени в этой области видимости. Описание `friend` может находится как в общей, так и в частной частях класса, это не имеет значения.

Теперь можно написать функцию multiply, используя элементы вектора и матрицы непосредственно:

```
vector multiply(const matrix& m, const vector& v)
{
  vector r;
  for (int i = 0; i<3; i++) {  // r[i] = m[i] * v;
  r.v[i] = 0;
  for ( int j = 0; j<3; j++)
     r.v[i] +=m.v[i][j] * v.v[j];
  }
  return r;
}
```

Отметим, что подобно функции-члену дружественная функция явно описывается в описании класса, с которым дружит. Поэтому она является неотъемлемой частью интерфейса класса наравне с функцией-членом.

Функция-член одного класса может быть другом другого класса:

```
class x {
  // ...
  void f();
};
```

```
class y {
  // ...
  friend void x::f();
};
```

Вполне возможно, что все функции одного класса являются друзьями другого класса. Для этого есть краткая форма записи:

```
class x {
  friend class y;
  // ...
};
```

В результате такого описания все функции-члены y становятся друзьями класса x.

5.4.2 Уточнение имени члена

Иногда полезно делать явное различие между именами членов классов и прочими именами. Для этого используется операция `::` (разрешения области видимости):

```
class X {
 int m;
  public:
 int readm() const { return m; }
 void setm(int m)  { X::m = m; }
 };
```

В функции `X::setm()` параметр `m` скрывает член `m`, поэтому к члену можно обращаться, только используя уточненное имя `X::m`. Правый операнд операции `::` должен быть именем класса.

Начинающееся с `::` имя должно быть глобальным именем. Это особенно полезно при использовании таких распространенных имен как `read`, `put`, `open`, которыми можно обозначать функции-члены, не теряя возможности обозначать ими же функции, не являющиеся членами.

Например:

```
class my_file {
// ...
 public:
int open(const char*, const char*);
 };

 int my_file::open(const char* name, const char* spec)
 {
  // ...
  if (::open(name,flag)) {  // используется open() из UNIX(2)
 // ...
  }
  // ...
}
```

5.4.3 Вложенные классы

Описание класса может быть вложенным. Например:

```
class set {
   struct setmem {
  int mem;
  setmem* next;
  setmem(int m, setmem* n) { mem=m; next=n; }
   };
   setmem* first;
public:
   set() { first=0; }
   insert(int m) { first = new setmem(m,first); }
   // ...
};
```

Доступность вложенного класса ограничивается областью видимости лексически объемлющего класса:

```
setmem m1(1,0);  // ошибка: setmem не находится
```

```
  // в глобальной области видимости
```

Если только описание вложенного класса не является совсем простым, то лучше описывать этот класс отдельно, поскольку вложенные описания могут стать очень запутанными:

```
class setmem {
friend class set;  // доступно только для членов set
  int mem;
  setmem* next;
  setmem(int m, setmem* n) { mem=m; next=n; }

  // много других полезных членов
};

class set {
  setmem* first;
public:
  set() { first=0; }
  insert(int m) { first = new setmem(m,first); }
  // ...
};
```

Полезное свойство вложенности - это сокращение числа глобальных имен, а недостаток его в том, что оно нарушает свободу использования вложенных типов.

Имя класса-члена (вложенного класса) можно использовать вне описания объемлющего его класса так же, как имя любого другого члена:

```
class X {
 struct M1 { int m; };
public:
 struct M2 { int m; };

 M1 f(M2);
};

void f()
{
```

```
 M1 a;     // ошибка: имя 'M1' вне области видимости
 M2 b;     // ошибка: имя 'M2' вне области видимости
 X::M1 c;  // ошибка: X::M1 частный член
 X::M2 d;  // нормально
}
```

Отметим, что контроль доступа происходит и для имен вложенных классов.

В функции-члене область видимости класса начинается после уточнения `X::` и простирается до конца описания функции. Например:

```
M1 X::f(M2 a)   // ошибка: имя `M1' вне области видимости
  { /* ... */ }

    X::M1 X::f(M2 a)  // нормально
  { /* ... */ }

    X::M1 X::f(X::M2 a) // нормально, но третье уточнение X:: излишне
  { /* ... */ }
```

5.4.4 Статические члены

Класс - это тип, а не некоторое данное, и для каждого объекта класса создается своя копия членов, представляющих данные. Однако, наиболее удачная реализация некоторых типов требует, чтобы все объекты этого типа имели некоторые общие данные. Лучше, если эти данные можно описать как часть класса. Например, в операционных системах или при моделировании управления задачами часто нужен список задач:

```
class task {
// ...
static task* chain;
// ...
      };
```

Описав член `chain` как статический, мы получаем гарантию, что он

будет создан в единственном числе, т.е. не будет создаваться для каждого объекта `task`. Но он находится в области видимости класса `task`, и может быть доступен вне этой области, если только описан в общей части. В этом случае имя члена должно уточняться именем класса:

```
if (task::chain == 0)  // какие-то операторы
```

В функции-члене его можно обозначать просто `chain`. Использование статических членов класса может заметно сократить потребность в глобальных переменных.

Описывая член как статический, мы ограничиваем его область видимости и делаем его независимым от отдельных объектов его класса. Это свойство полезно как для функций-членов, так и для членов, представляющих данные:

```
class task {
 // ...
 static task* task_chain;
 static void shedule(int);
 // ...
};
```

Но описание статического члена - это только описание, и где-то в программе должно быть единственное определение для описываемого объекта или функции, например, такое:

```
task* task::task_chain = 0;
void task::shedule(int p) { /* ... */ }
```

Естественно, что и частные члены могут определяться подобным образом.

Отметим, что служебное слово `static` не нужно и даже нельзя использовать в определении статического члена класса. Если бы оно присутствовало, возникла бы неоднозначность: указывает ли оно на то, что член класса является статическим, или используется для описания глобального объекта или функции?

Слово static одно из самых перегруженных служебных слов в C и C++. К статическому члену, представляющему данные, относятся оба основных его значения: "статически размещаемый" , т.е. противоположный объектам, размещаемым в стеке или свободной памяти, и "статический" в смысле с ограниченной областью видимости, т.е. противоположный объектам, подлежащим внешнему связыванию. К функциям-членам относится только второе значение static.

5.4.5 Указатели на члены

Можно брать адрес члена класса. Операция взятия адреса функции-члена часто оказывается полезной, поскольку цели и способы применения указателей на функции, в равной степени относятся и к таким функциям. Указатель на член можно получить, применив операцию взятия адреса & к полностью уточненному имени члена класса, например, `&class_name::member_name`. Чтобы описать переменную типа "указатель на член класса X", надо использовать описатель вида `X::*`. Например:

```
#include <iostream.h>

struct cl
{
  char* val;
  void print(int x) { cout << val << x << '\n'; }
  cl(char* v) { val = v; }
};
```

Указатель на член можно описать и использовать так:

```
typedef void (cl::*PMFI)(int);

int main()
        {
cl z1("z1 ");
cl z2("z2 ");
cl* p = &z2;
PMFI pf = &cl::print;
```

```
    z1.print(1);
    (z1.*pf)(2);
    z2.print(3);
    (p->*pf)(4);
        }
```

Использование `typedef` для замены трудно воспринимаемого описателя в C достаточно типичный случай. Операции `.*` и `->*` настраивают указатель на конкретный объект, выдавая в результате функцию, которую можно вызывать. Приоритет операции `()` выше, чем у операций `.*` и `->*`, поэтому нужны скобки.

Во многих случаях виртуальные функции успешно заменяют указатели на функции.

5.4.6 Структуры и объединения

По определению структура - это класс, все члены которого общие, т.е. описание

```
struct s { ...
```

это просто краткая форма описания

```
class s { public: ...
```

Поименованное объединение определяется как структура, все члены которой имеют один и тот же адрес. Если известно, что в каждый момент времени используется значение только одного члена структуры, то объявив ее объединением, можно сэкономить память. Например, можно использовать объединение для хранения лексем транслятора C:

```
union tok_val {
  char* p;      // строка
  char v[8];    // идентификатор (не более 8 символов)
  long i;       // значения целых
  double d;     // значения чисел с плавающей точкой
};
```

Проблема с объединениями в том, что транслятор в общем случае не знает, какой член используется в данный момент, и поэтому контроль типа невозможен. Например:

```
void strange(int i)
{
  tok_val x;
  if (i)
 x.p = "2";
  else
 x.d = 2;
  sqrt(x.d);     // ошибка, если i != 0
}
```

Кроме того, определенное таким образом объединение нельзя инициализировать таким кажущимся вполне естественным способом:

```
tok_val val1 = 12;   // ошибка: int присваивается  tok_val
tok_val val2 = "12"; // ошибка: char* присваивается tok_val
```

Для правильной инициализации надо использовать конструкторы:

```
union tok_val {
  char* p;    // строка
  char v[8];  // идентификатор (не более 8 символов)
  long i;     // значения целых
  double d;   // значения чисел с плавающей точкой

  tok_val(const char*);  // нужно выбирать между p и v
  tok_val(int ii)    { i = ii; }
  tok_val(double dd) { d = dd; }
};
```

Эти описания позволяют разрешить с помощью типа членов неоднозначность при перегрузке имени функции. Например:

```
void f()
{
  tok_val a = 10;    // a.i = 10
  tok_val b = 10.0;  // b.d = 10.0
```

```
}
```

Если это невозможно (например, для типов `char*` и `char[8]` или `int` и `char` и т.д.), то определить, какой член инициализируется, можно, изучив инициализатор при выполнении программы, или введя дополнительный параметр. Например:

```
tok_val::tok_val(const char* pp)
{
  if (strlen(pp) <= 8)
 strncpy(v,pp,8);    // короткая строка
  else
 p = pp; // длинная строка
}
```

Но лучше подобной неоднозначности избегать.

Стандартная функция `strncpy()` подобно `strcpy()` копирует строки, но у нее есть дополнительный параметр, задающий максимальное число копируемых символов.

То, что для инициализации объединения используются конструкторы, еще не гарантирует от случайных ошибок при работе с объединением, когда присваивается значение одного типа, а выбирается значение другого типа. Такую гарантию можно получить, если заключить объединение в класс, в котором будет отслеживаться тип заносимого значения :

```
class tok_val {
public:
  enum Tag { I, D, S, N };

        private:
  union {
const char* p;
char v[8];
long i;
double d;
  };
```

```
    Tag tag;

    void check(Tag t) { if (tag != t) error(); }
  public:
    Tag get_tag() { return tag; }

    tok_val(const char* pp);
    tok_val(long ii)   { i = ii; tag = I; }
    tok_val(double dd) { d = dd; tag = D; }

    long& ival()        { check(I); return i; }
    double& fval()      { check(D); return d; }
    const char*& sval() { check(S); return p; }
    char* id()          { check(N); return v; }
          };

          tok_val::tok_val(const char* pp)
          {
   if (strlen(pp) <= 8)  { // короткая строка
  tag = N;
  strncpy(v,pp,8);
   }
   else {   // длинная строка
  tag = S;
  p = pp;   // записывается только указатель
   }
  }
```

Использовать класс `tok_val` можно так:

```
  void f()
  {
   tok_val t1("короткая");       // присваивается v
   tok_val t2("длинная строка"); // присваивается p
   char s[8];
   strncpy(s,t1.id(),8);         // нормально
   strncpy(s,t2.id(),8);         // check() выдаст ошибку
  }
```

Описав тип `Tag` и функцию `get_tag()` в общей части, мы гарантируем, что тип `tok_val` можно использовать как тип параметра. Таким образом, появляется надежная в смысле типов альтернатива описанию параметров с эллипсисом. Вот, например, описание функции обработки ошибок, которая может иметь один, два, или три параметра с типами `char*`, `int` или `double`:

```
extern tok_val no_arg;

void error(
const char* format,
tok_val a1 = no_arg,
tok_val a2 = no_arg,
tok_val a3 = no_arg);
```

5.5 Конструкторы и деструкторы

Если у класса есть конструктор, он вызывается всякий раз при создании объекта этого класса. Если у класса есть деструктор, он вызывается всякий раз, когда уничтожается объект этого класса. Объект может создаваться как:

1. автоматический, который создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается при выходе из блока, в котором он описан;
2. статический, который создается один раз при запуске программы и уничтожается при ее завершении;
3. объект в свободной памяти, который создается операцией `new` и уничтожается операцией `delete` ;
4. объект-член, который создается в процессе создания другого класса или при создании массива, элементом которого он является.

Кроме этого объект может создаваться, если в выражении явно используется его конструктор или как временный объект. В обоих случаях такой объект не имеет имени. В следующих подразделах предполагается, что объекты относятся к классу с конструктором и деструктором.

5.5.1 Локальные переменные

Конструктор локальной переменной вызывается каждый раз, когда при выполнении программы встречается ее описание. Деструктор локальной переменной вызывается всякий раз при выходе из блока, где она была описана. Деструкторы для локальных переменных вызываются в порядке, обратном вызову конструкторов при их создании:

```
void f(int i)
{
 table aa;
 table bb;
 if (i>0) {
 table cc;
 // ...
 }
 // ...
}
```

Здесь `aa` и `bb` создаются (именно в таком порядке) при каждом вызове `f()`, а уничтожаются они при возврате из `f()` в обратном порядке - `bb`, затем `aa`. Если в текущем вызове `f()` `i` больше нуля, то `cc` создается после `bb` и уничтожается прежде него.

Поскольку aa и `bb` - объекты класса `table`, присваивание `aa=bb` означает копирование по членам `bb` в `aa`. Такая интерпретация присваивания может привести к неожиданному (и обычно нежелательному) результату, если присваиваются объекты класса, в котором определен конструктор:

```
void h()
{
  table t1(100);
  table t2 = t1;   // неприятность
  table t3(200);

  t3 = t2;         // неприятность
```

```
}
```

В этом примере конструктор `table` вызывается дважды: для `t1` и `t3`. Он не вызывается для `t2`, поскольку этот объект инициализируется присваиванием. Тем не менее, деструктор для `table` вызывается три раза: для `t1`, `t2` и `t3`! Далее, стандартная интерпретация присваивания - это копирование по членам, поэтому перед выходом из `h()` `t1`, `t2` и `t3` будут содержать указатель на массив имен, память для которого была выделена в свободной памяти при создании `t1`. Указатель на память, выделенную для массива имен при создании `t3`, будет потерян. Этих неприятностей можно избежать.

5.5.2 Статическая память

Рассмотрим такой пример:

```
table tbl(100);

void f(int i)
{
    static table tbl2(i);
}

int main()
{
f(200);
 // ...
}
```

Здесь конструктор, определенный в § 5.3.1, будет вызываться дважды: один раз для `tbl` и один раз для `tbl2`. Деструктор `table::~table()` также будет вызван дважды: для уничтожения `tbl` и `tbl2` при выходе из `main()`. Конструкторы глобальных статических объектов в файле вызываются в том же порядке, в каком встречаются в файле описания объектов, а деструкторы для них вызываются в обратном порядке. Конструктор локального статического объекта вызывается, когда при выполнении программы первый раз

встречается определение объекта.

Традиционно выполнение `main()` рассматривалось как выполнение всей программы. На самом деле, это не так даже для C. Уже размещение статического объекта класса с конструктором и (или) деструктором позволяет программисту задать действия, которые будут выполняться до вызова `main()` и (или) при выходе из `main()`.

Вызов конструкторов и деструкторов для статических объектов играет в C++ чрезвычайно важную роль. С их помощью можно обеспечить соответствующую инициализацию и удаление структур данных, используемых в библиотеках. Рассмотрим <iostream.h>. Откуда берутся `cin`, `cout` и `cerr`? Когда они инициализируются? Более существенный вопрос: поскольку для выходных потоков используются внутренние буфера символов, то происходит выталкивание этих буферов, но когда? Есть простой и очевидный ответ: все действия выполняются соответствующими конструкторами и деструкторами до запуска `main()` и при выходе из нее. Существуют альтернативы использованию конструкторов и деструкторов для инициализации и уничтожения библиотечных структур данных, но все они или очень специализированы, или неуклюжи, или и то и другое вместе.

Если программа завершается обращением к функции `exit()`, то вызываются деструкторы для всех построенных статических объектов. Однако, если программа завершается обращением к `abort()`, этого не происходит. Заметим, что `exit()` не завершает программу немедленно. Вызов `exit()` в деструкторе может привести к бесконечной рекурсии. Если нужна гарантия, что будут уничтожены как статические, так и автоматические объекты, можно воспользоваться особыми ситуациями.

Иногда при разработке библиотеки бывает необходимо или просто удобно создать тип с конструктором и деструктором только для одной цели: инициализации и уничтожения объектов. Такой тип используется только один раз для размещения статического объекта, чтобы вызвать конструкторы и деструкторы.

5.5.3 Освобождение памяти

Рассмотрим пример:

```
main()
{
  table* p = new table(100);
  table* q = new table(200);
  delete p;
  delete p;  // вероятно, вызовет ошибку при выполнении
}
```

Конструктор `table::table()` будет вызываться дважды, как и деструктор `table::~table()`. Но это ничего не значит, т.к. в C++ не гарантируется, что деструктор будет вызываться только для объекта, созданного операцией `new`. В этом примере `q` не уничтожается вообще, зато `p` уничтожается дважды! В зависимости от типа `p` и `q` программист может считать или не считать это ошибкой. То, что объект не удаляется, обычно бывает не ошибкой, а просто потерей памяти. В то же время повторное удаление `p` - серьезная ошибка. Повторное применение `delete` к тому же самому указателю может привести к бесконечному циклу в подпрограмме, управляющей свободной памятью. Но в языке результат повторного удаления не определен, и он зависит от реализации.

Пользователь может определить свою реализацию операций `new` и `delete`. Кроме того, можно установить взаимодействие конструктора или деструктора с операциями `new` и `delete`.

5.5.4 Объекты класса как члены

Рассмотрим пример:

```
class classdef {
  table members;
  int no_of_members;
  // ...
  classdef(int size);
  ~classdef();
};
```

Цель этого определения, очевидно, в том, чтобы `classdef` содержал член, являющийся таблицей размером `size`, но есть сложность: надо обеспечить вызов конструктора `table::table()` с параметром size. Это можно сделать, например, так:

```
classdef::classdef(int size)
 :members(size)
        {
 no_of_members = size;
 // ...
}
```

Параметр для конструктора члена (т.е. для `table::table()`) указывается в определении (но не в описании) конструктора класса, содержащего член (т.е. в определении `classdef::classdef()`). Конструктор для члена будет вызываться до выполнения тела того конструктора, который задает для него список параметров.

Аналогично можно задать параметры для конструкторов других членов (если есть еще другие члены):

```
class classdef {
  table members;
  table friends;
  int no_of_members;
  // ...
  classdef(int size);
  ~classdef();
};
```

Списки параметров для членов отделяются друг от друга запятыми (а не двоеточиями), а список инициализаторов для членов можно задавать в произвольном порядке:

```
classdef::classdef(int size)
: friends(size), members(size), no_of_members(size)
{
  // ...
}
```

Конструкторы вызываются в том порядке, в котором они заданы в описании класса.

Подобные описания конструкторов существенны для типов, инициализация и присваивание которых отличны друг от друга, иными словами, для объектов, являющихся членами класса с конструктором, для постоянных членов или для членов типа ссылки. Однако, как показывает член `no_of_members` из приведенного примера, такие описания конструкторов можно использовать для членов любого типа.

Если конструктору члена не требуется параметров, то и не нужно задавать никаких списков параметров. Так, поскольку конструктор `table::table()` был определен со стандартным значением параметра, равным 15, достаточно такого определения:

```
classdef::classdef(int size)
  : members(size), no_of_members(size)
      {
  // ...
}
```

Тогда размер таблицы `friends` будет равен 15.

Если уничтожается объект класса, который сам содержит объекты класса (например, `classdef`), то вначале выполняется тело деструктора объемлющего класса, а затем деструкторы членов в порядке, обратном их описанию.

Рассмотрим вместо вхождения объектов класса в качестве членов традиционное альтернативное ему решение: иметь в классе указатели на члены и инициализировать члены в конструкторе:

```
class classdef {
  table* members;
  table* friends;
  int no_of_members;
  // ...
};

      classdef::classdef(int size)
```

```
        {
  members = new table(size);
  friends = new table;  // используется стандартный
    // размер table
  no_of_members = size;
  // ...
}
```

Поскольку таблицы создавались с помощью операции new, они должны уничтожаться операцией `delete`:

```
classdef::~classdef()
{
 // ...
 delete members;
 delete friends;
}
```

Такие отдельно создаваемые объекты могут оказаться полезными, но учтите, что `members` и `friends` указывают на независимые от них объекты, каждый из которых надо явно размещать и удалять. Кроме того, указатель и объект в свободной памяти суммарно занимают больше места, чем объект-член.

5.5.5 Массивы объектов класса

Чтобы можно было описать массив объектов класса с конструктором, этот класс должен иметь стандартный конструктор, т.е. конструктор, вызываемый без параметров. Например, в соответствии с определением

```
table tbl[10];
```

будет создан массив из 10 таблиц, каждая из которых инициализируется вызовом `table::table(15)`, поскольку вызов `table::table()` будет происходить с фактическим параметром 15.

В описании массива объектов не предусмотрено возможности указать

параметры для конструктора. Если члены массива обязательно надо инициализировать разными значениями, то начинаются трюки с глобальными или статическими членами.

Когда уничтожается массив, деструктор должен вызываться для каждого элемента массива. Для массивов, которые размещаются не с помощью `new`, это делается неявно. Однако для размещенных в свободной памяти массивов неявно вызывать деструктор нельзя, поскольку транслятор не отличит указатель на отдельный объект массива от указателя на начало массива, например:

```
void f()
{
 table* t1 = new table;
 table* t2 = new table[10];
 delete t1;  // удаляется одна таблица
 delete t2;  // неприятность:
        // на самом деле удаляется 10 таблиц
}
```

В данном случае программист должен указать, что `t2` - указатель на массив:

```
void g(int sz)
{
  table* t1 = new table;
  table* t2 = new table[sz];
  delete t1;
  delete[] t2;
}
```

Функция размещения хранит число элементов для каждого размещаемого массива. Требование использовать для удаления массивов только операцию `delete[]` освобождает функцию размещения от обязанности хранить счетчики числа элементов для каждого массива. Исполнение такой обязанности в реализациях C++ вызывало бы существенные потери времени и памяти и нарушило совместимость с C.

5.5.6 Небольшие объекты

Если в вашей программе много небольших объектов, размещаемых в свободной памяти, то может оказаться, что много времени тратится на размещение и удаление таких объектов. Для выхода из этой ситуации можно определить более оптимальный распределитель памяти общего назначения, а можно передать обязанность распределения свободной памяти создателю класса, который должен будет определить соответствующие функции размещения и удаления.

Вернемся к классу `name`, который использовался в примерах с `table`. Он мог бы определяться так:

```
struct name {
  char* string;
  name* next;
  double value;

  name(char*, double, name*);
  ~name();

  void* operator new(size_t);
  void operator delete(void*, size_t);
private:
  enum { NALL = 128 };
  static name* nfree;
};
```

Функции `name::operator new()` и `name::operator delete()` будут использоваться (неявно) вместо глобальных функций `operator new()` и `operator delete()`. Программист может для конкретного типа написать более эффективные по времени и памяти функции размещения и удаления, чем универсальные функции `operator new()` и `operator delete()`. Можно, например, разместить заранее "куски" памяти, достаточной для объектов типа `name`, и связать их в список; тогда операции размещения и удаления сводятся к простым операциям со списком. Переменная `nfree` используется как начало списка неиспользованных кусков памяти:

```
void* name::operator new(size_t)
 {
  register name* p = nfree;  // сначала выделить

  if (p)
  nfree = p->next;
  else { // выделить и связать в список
  name* q = (name*) new char[NALL*sizeof(name) ];
  for (p=nfree=&q[NALL-1]; q<p; p--) p->next = p-1;
  (p+1)->next = 0;
  }

  return p;
}
```

Распределитель памяти, вызываемый `new`, хранит вместе с объектом его размер, чтобы операция `delete` выполнялась правильно. Этого дополнительного расхода памяти можно легко избежать, если использовать распределитель, рассчитанный на конкретный тип. Так, на машине автора функция `name::operator new()` для хранения объекта name использует 16 байтов, тогда как стандартная глобальная функция `operator new()` использует 20 байтов.

Отметим, что в самой функции `name::operator new()` память нельзя выделять таким простым способом:

```
name* q= new name[NALL];
```

Это вызовет бесконечную рекурсию, т.к. `new` будет вызывать `name::name()`.

Освобождение памяти обычно тривиально:

```
void name::operator delete(void* p, size_t)
{
 ((name*)p)->next = nfree;
 nfree = (name*) p;
}
```

Приведение параметра типа `void*` к типу `name*` необходимо, поскольку функция освобождения вызывается после уничтожения объекта, так что больше нет реального объекта типа `name`, а есть только кусок памяти размером `sizeof(name)`. Параметры типа `size_t` в приведенных функциях `name::operator new()` и `name::operator delete()` не использовались. Отметим, что наши функции размещения и удаления используются только для объектов типа `name`, но не для массивов `names`.

Производные классы

Эта лекция посвящена понятию производного класса. Производные классы - это простое, гибкое и эффективное средство определения класса. Новые возможности добавляются к уже существующему классу, не требуя его перепрограммирования или перетрансляции. С помощью производных классов можно организовать общий интерфейс с несколькими различными классами так, что в других частях программы можно будет единообразно работать с объектами этих классов. Вводится понятие виртуальной функции, которое позволяет использовать объекты надлежащим образом даже в тех случаях, когда их тип на стадии трансляции неизвестен. Основное назначение производных классов - упростить программисту задачу выражения общности классов.

6.1 Введение и краткий обзор

Любое понятие не существует изолированно, оно существует во взаимосвязи с другими понятиями, и мощность данного понятия во многом определяется наличием таких связей. Раз класс служит для представления понятий, встает вопрос, как представить взаимосвязь понятий. Понятие производного класса и поддерживающие его языковые средства служат для представления иерархических связей, иными словами, для выражения общности между классами. Например, понятия окружности и треугольника связаны между собой, так как оба они представляют еще понятие фигуры, т.е. содержат более общее понятие. Чтобы представлять в программе окружности и треугольники и при этом не упускать из вида, что они являются фигурами, надо явно определять классы окружность и треугольник так, чтобы было видно, что у них есть общий класс - фигура. В лекции исследуется, что вытекает из этой простой идеи, которая по сути является основой того, что обычно называется объектно-ориентированным программированием. Лекция состоит из шести разделов:

§ 6.2с помощью серии небольших примеров вводится понятие производного класса, иерархии классов и виртуальных функций.

§ 6.3 вводится понятие чисто виртуальных функций и абстрактных классов, даны небольшие примеры их использования.

§ 6.4 производные классы показаны на законченном примере

§ 6.5 вводится понятие множественного наследования как возможность иметь для класса более одного прямого базового класса, описываются способы разрешения коллизий имен, возникающих при множественном наследовании.

§ 6.6 обсуждается механизм контроля доступа.

§ 6.7 приводятся некоторые приемы управления свободной памятью для производных классов.

В последующих лекциях также будут приводиться примеры, использующие эти возможности языка.

6.2 Производные классы

Обсудим, как написать программу учета служащих некоторой фирмы. В ней может использоваться, например, такая структура данных:

```
struct employee {// служащий
    char*    name;        // имя
    short    age;         // возраст
    short    department;  // отдел
    int      salary;      // оклад
    employee* next;
    // ...
};
```

Поле `next` нужно для связывания в список записей о служащих одного отдела (employee). Теперь попробуем определить структуру данных для управляющего (manager):

```
struct manager {
    employee emp;    // запись employee для управляющего
    employee* group; // подчиненный коллектив
    short   level;
    // ...
```

```
};
```

Управляющий также является служащим, поэтому запись `employee` хранится в члене `emp` объекта `manager`. Для человека эта общность очевидна, но для транслятора член emp ничем не отличается от других членов класса. Указатель на структуру `manager` (manager*) не является указателем на `employee` (employee*), поэтому нельзя свободно использовать один вместо другого. В частности, без специальных действий нельзя объект `manager` включить в список объектов типа `employee`. Придется либо использовать явное приведение типа `manager*`, либо в список записей `employee` включить адрес члена `emp`. Оба решения некрасивы и могут быть достаточно запутанными. Правильное решение состоит в том, чтобы тип `manager` был типом `employee` с некоторой дополнительной информацией:

```
struct manager : employee {
  employee* group;
  short level;
  // ...
};
```

Класс `manager` является производным от `employee`, и, наоборот, `employee` является базовым классом для `manager`. Помимо члена `group` в классе `manager` есть члены класса `employee` (name, age и т.д.).

Графически отношение наследования обычно изображается в виде стрелки от производных классов к базовому:

```
employee
   ^
   |
 manager
```

Обычно говорят, что производный класс наследует базовый класс, поэтому и отношение между ними называется наследованием. Иногда базовый класс называют суперклассом, а производный - подчиненным классом. Но эти термины могут вызывать недоумение, поскольку объект производного класса содержит объект своего базового класса. Вообще

производный класс больше своего базового в том смысле, что в нем содержится больше данных и определено больше функций.

Имея определения `employee` и `manager`, можно создать список служащих, часть из которых является и управляющими:

```
void f()
{
  manager m1, m2;
  employee e1, e2;
  employee* elist;
  elist = &m1;          // поместить m1 в elist
  m1.next = &e1;        // поместить e1 в elist
  e1.next = &m2;        // поместить m2 в elist
  m2.next = &e2;        // поместить e2 в elist
  e2.next = 0;          // конец списка
}
```

Поскольку управляющий является и служащим, указатель `manager*` можно использовать как `employee *`. В тоже время служащий не обязательно является управляющим, и поэтому `employee*` нельзя использовать как `manager*`.

В общем случае, если класс `derived` имеет общий базовый класс `base`, то указатель на `derived` можно без явных преобразований типа присваивать переменной, имеющей тип указателя на `base`. Обратное преобразование от указателя на base к указателю на `derived` может быть только явным:

```
void g()
{
  manager mm;
  employee* pe = &mm;  // нормально

  employee ee;
  manager* pm = &ee;  // ошибка:
    // не всякий служащий является управляющим

  pm->level = 2;      // катастрофа: при размещении ee
```

```
    // память для члена `level' не выделялась

  pm = (manager*) pe;  // нормально: на самом деле pe
    // не настроено на объект mm типа manager

  pm->level = 2;       // отлично: pm указывает на объект mm
    // типа manager, а в нем при размещении
    // выделена память для члена `level'
      }
```

Иными словами, если работа с объектом производного класса идет через указатель, то его можно рассматривать как объект базового класса. Обратное неверно. Отметим, что в обычной реализации C++ не предполагается динамического контроля над тем, чтобы после преобразования типа, подобного тому, которое использовалось в присваивании `pe` в `pm`, получившийся в результате указатель действительно был настроен на объект требуемого типа.

6.2.1 Функции-члены

Простые структуры данных вроде `employee` и `manager` сами по себе не слишком интересны, а часто и не особенно полезны. Поэтому добавим к ним функции:

```
class employee {
  char* name;
  // ...
public:
  employee* next;  // находится в общей части, чтобы
     // можно было работать со списком
  void print() const;
  // ...
     };

     class manager : public employee {
  // ...
     public:
  void print() const;
```

```
    // ...
    };
```

Надо ответить на некоторые вопросы. Каким образом функция-член производного класса `manager` может использовать члены базового класса `employee`? Какие члены базового класса `employee` могут использовать функции-члены производного класса `manager`? Какие члены базового класса `employee` может использовать функция, не являющаяся членом объекта типа `manager`? Какие ответы на эти вопросы должна давать реализация языка, чтобы они максимально соответствовали задаче программиста?

Рассмотрим пример:

```
void manager::print() const
{
  cout << " имя "   << name << '\n';
}
```

Член производного класса может использовать имя из общей части своего базового класса наравне со всеми другими членами, т.е. без указания имени объекта. Предполагается, что есть объект, на который настроен this, поэтому корректным обращением к name будет this->name. Однако, при трансляции функции `manager::print()` будет зафиксирована ошибка: члену производного класса не предоставлено право доступа к частным членам его базового класса, значит name недоступно в этой функции.

Возможно многим это покажется странным, но давайте рассмотрим альтернативное решение: функция-член производного класса имеет доступ к частным членам своего базового класса. Тогда само понятие частного (закрытого) члена теряет всякий смысл, поскольку для доступа к нему достаточно просто определить производный класс. Теперь уже будет недостаточно для выяснения, кто использует частные члены класса, просмотреть все функции-члены и друзей этого класса. Придется просмотреть все исходные файлы программы, найти производные классы, затем исследовать каждую функцию этих классов. Далее надо снова искать производные классы от уже найденных и т.д. Это, по крайней мере, утомительно, а скорее всего нереально. Нужно

всюду, где это возможно, использовать вместо частных членов защищенные.

Как правило, самое надежное решение для производного класса - использовать только общие члены своего базового класса:

```
void manager::print() const
{
  employee::print();   // печать данных о служащих

  // печать данных об управляющих
}
```

Отметим, что операция :: необходима, поскольку функция print() переопределена в классе `manager`. Такое повторное использование имен типично для C++. Неосторожный программист написал бы:

```
void manager::print() const
{
   print();   // печать данных о служащих

   // печать данных об управляющих
}
```

В результате он получил бы рекурсивную последовательность вызовов `manager::print()`.

6.2.2 Конструкторы и деструкторы

Для некоторых производных классов нужны конструкторы. Если конструктор есть в базовом классе, то именно он и должен вызываться с указанием параметров, если таковые у него есть:

```
class employee {
  // ...
public:
  // ...
  employee(char* n, int d);
```

```
};

class manager : public employee {
// ...
public:
// ...
manager(char* n, int i, int d);
};
```

Параметры для конструктора базового класса задаются в определении конструктора производного класса. В этом смысле базовый класс выступает как класс, являющийся членом производного класса:

```
manager::manager(char* n, int l, int d)
  : employee(n,d), level(l), group(0)
{
}
```

Конструктор базового класса `employee::employee()` может иметь такое определение:

```
employee::employee(char* n, int d)
   : name(n), department(d)
{
 next = list;
 list = this;
}
```

Здесь `list` должен быть описан как статический член `employee`. Объекты классов создаются снизу вверх: вначале базовые, затем члены и, наконец, сами производные классы. Уничтожаются они в обратном порядке: сначала сами производные классы, затем члены, а затем базовые. Члены и базовые создаются в порядке описания их в классе, а уничтожаются они в обратном порядке.

6.2.3 Иерархия классов

Производный класс сам в свою очередь может быть базовым классом:

```
class employee { /* ... */ };
class manager : public employee { /* ... */ };
class director : public manager { /* ... */ };
```

Такое множество связанных между собой классов обычно называют иерархией классов. Обычно она представляется деревом, но бывают иерархии с более общей структурой в виде графа:

```
class temporary { /* ... */ };
class secretary : public employee { /* ... */ };

    class tsec
: public temporary, public secretary { /* ... */ };

    class consultant
: public temporary, public manager { /* ... */ };
```

Видим, что классы в C++ могут образовывать направленный ацикличный граф.

6.2.4 Поля типа

Чтобы производные классы были не просто удобной формой краткого описания, в реализации языка должен быть решен вопрос: к какому из производных классов относится объект, на который смотрит указатель base*? Существует три основных способа ответа:

1. Обеспечить, чтобы указатель мог ссылаться на объекты только одного типа;
2. Поместить в базовый класс поле типа, которое смогут проверять функции;
3. использовать виртуальные функции.

Указатели на базовые классы обыкновенно используются при проектировании контейнерных классов (множество, вектор, список и т.д.). Тогда в случае [1] мы получим однородные списки, т.е. списки объектов одного типа. Способы [2] и [3] позволяют создавать разнородные списки, т.е. списки объектов нескольких различных типов

(на самом деле, списки указателей на эти объекты). Способ [3] - это специальный надежный в смысле типа вариант способа [2]. Особенно интересные и мощные варианты дают комбинации способов [1] и [3].

Вначале обсудим простой способ с полем типа, т.е. способ [2]. Пример с классами `manager/employee` можно переопределить так:

```
struct employee {
  enum empl_type { M, E };
  empl_type type;
  employee* next;
  char*     name;
  short     department;
  // ...
      };

      struct manager : employee {
  employee* group;
  short     level;
  // ...
      };
```

Имея эти определения, можно написать функцию, печатающую данные о произвольном служащем:

```
void print_employee(const employee* e)
{
  switch (e->type) {
  case E:
    cout << e->name << '\t' << e->department << '\n';
    // ...
    break;
  case M:
    cout << e->name << '\t' << e->department << '\n';
    // ...
    manager* p = (manager*) e;
    cout << "level" << p->level << '\n';
    // ...
    break;
```

```
    }
  }
```

Напечатать список служащих можно так:

```
void f(const employee* elist)
        {
          for (; elist; elist=elist->next) print_employee(elist);
        }
```

Это вполне хорошее решение, особенно для небольших программ, написанных одним человеком, но оно имеет существенный недостаток: транслятор не может проверить, насколько правильно программист обращается с типами. В больших программах это приводит к ошибкам двух видов. Первый - когда программист забывает проверить поле типа. Второй - когда в переключателе указываются не все возможные значения поля типа. Этих ошибок достаточно легко избежать в процессе написания программы, но совсем нелегко избежать их при внесении изменений в нетривиальную программу, а особенно, если это большая программа, написанная кем-то другим. Еще труднее избежать таких ошибок потому, что функции типа print() часто пишутся так, чтобы можно было воспользоваться общностью классов:

```
void print(const employee* e)
{
  cout << e->name << '\t' << e->department << '\n';
  // ...
  if (e->type == M) {
    manager* p = (manager*) e;
    cout << "level" << p->level << '\n';
    // ...
  }
}
```

Операторы `if`, подобные приведенным в примере, сложно найти в большой функции, работающей со многими производными классами. Но даже когда они найдены, нелегко понять, что происходит на самом деле. Кроме того, при всяком добавлении нового вида служащих требуются изменения во всех важных функциях программы, т.е. функциях, проверяющих поле типа. В результате приходится править

важные части программы, увеличивая тем самым время на отладку этих частей.

Иными словами, использование поля типа чревато ошибками и трудностями при сопровождении программы. Трудности резко возрастают по мере роста программы, ведь использование поля типа противоречит принципам модульности и упрятывания данных. Каждая функция, работающая с полем типа, должна знать представление и специфику реализации всякого класса, являющегося производным для класса, содержащего поле типа.

6.2.5 Виртуальные функции

С помощью виртуальных функций можно преодолеть трудности, возникающие при использовании поля типа. В базовом классе описываются функции, которые могут переопределяться в любом производном классе. Транслятор и загрузчик обеспечат правильное соответствие между объектами и применяемыми к ним функциями:

```
class employee {
  char* name;
  short department;
  // ...
  employee* next;
  static employee* list;
public:
  employee(char* n, int d);
  // ...
  static void print_list();
  virtual void print() const;

      };
```

Служебное слово `virtual` (виртуальная) показывает, что функция `print()` может иметь разные версии в разных производных классах, а выбор нужной версии при вызове `print()` - это задача транслятора. Тип функции указывается в базовом классе и не может быть переопределен в производном классе. Определение виртуальной

функции должно даваться для того класса, в котором она была впервые описана (если только она не является чисто виртуальной функцией). Например:

```
void employee::print() const
{
  cout << name << '\t' << department << '\n';
  // ...
}
```

Мы видим, что виртуальную функцию можно использовать, даже если нет производных классов от ее класса. В производном же классе не обязательно переопределять виртуальную функцию, если она там не нужна. При построении производного класса надо определять только те функции, которые в нем действительно нужны:

```
class manager : public employee {
    employee* group;
    short     level;
    // ...
 public:
    manager(char* n, int d);
    // ...
    void print() const;
 };
```

Место функции `print_employee()` заняли функции-члены `print()`, и она стала не нужна. Список служащих строит конструктор `employee`. Напечатать его можно так:

```
void employee::print_list()
 {
   for ( employee* p = list; p; p=p->next) p->print();
 }
```

Данные о каждом служащем будут печататься в соответствии с типом записи о нем. Поэтому программа

```
int main()
{
```

```
    employee e("J.Brown",1234);
    manager m("J.Smith",2,1234);
    employee::print_list();
}
```

напечатает

```
J.Smith 1234
      level 2
J.Brown 1234
```

Обратите внимание, что функция печати будет работать даже в том случае, если функция `employee_list()` была написана и оттранслирована еще до того, как был задуман конкретный производный класс `manager`! Очевидно, что для правильной работы виртуальной функции нужно в каждом объекте класса `employee` хранить некоторую служебную информацию о типе. Как правило, реализации в качестве такой информации используют просто указатель. Этот указатель хранится только для объектов класса с виртуальными функциями, и не хранится для объектов всех классов, и даже не для всех объектов производных классов. Дополнительная память отводится только для классов, в которых описаны виртуальные функции. Заметим, что при использовании поля типа, для него все равно нужна дополнительная память.

Если в вызове функции явно указана операция разрешения области видимости `::`, например, в вызове `manager::print()`, то механизм вызова виртуальной функции не действует. Иначе подобный вызов привел бы к бесконечной рекурсии. Уточнение имени функции дает еще один положительный эффект: если виртуальная функция является подстановкой (в этом нет ничего необычного), то в вызове с операцией `::` происходит подстановка тела функции. Это эффективный способ вызова, который можно применять в важных случаях, когда одна виртуальная функция обращается к другой с одним и тем же объектом. Пример такого случая - вызов функции `manager::print()`. Поскольку тип объекта явно задается в самом вызове `manager::print()`, нет нужды определять его в динамике для функции `employee::print()`, которая и будет вызываться.

6.3 Абстрактные классы

Многие классы сходны с классом `employee` тем, что в них можно дать разумное определение виртуальным функциям. Однако, есть и другие классы. Некоторые, например, класс `shape`, представляют абстрактное понятие (фигура), для которого нельзя создать объекты. Класс `shape` приобретает смысл только как базовый класс в некотором производном классе. Причиной является то, что невозможно дать осмысленное определение виртуальных функций класса `shape`:

```
class shape {
  // ...
public:
  virtual void rotate(int) { error("shape::rotate"); }
  virtual void draw()  { error("shape::draw"): }
  // нельзя ни вращать, ни рисовать абстрактную фигуру
  // ...
};
```

Создание объекта типа `shape` (абстрактной фигуры) законная, хотя совершенно бессмысленная операция:

```
shape s;  // бессмыслица: ``фигура вообще"
```

Она бессмысленна потому, что любая операция с объектом s приведет к ошибке.

Лучше виртуальные функции класса `shape` описать как чисто виртуальные. Сделать виртуальную функцию чисто виртуальной можно, добавив инициализатор = 0:

```
class shape {
  // ...
public:
  virtual void rotate(int) = 0; // чисто виртуальная функция
  virtual void draw() = 0;      // чисто виртуальная функция
};
```

Класс, в котором есть чисто виртуальные функции, называется

абстрактным. Объекты такого класса создать нельзя:

```
shape s;   // ошибка: переменная абстрактного класса shape
```

Абстрактный класс можно использовать только в качестве базового для другого класса:

```
class circle : public shape {
  int radius;
public:
  void rotate(int) { } // нормально:
 // переопределение shape::rotate
  void draw();         // нормально:
 // переопределение shape::draw

  circle(point p, int r);
};
```

Если чисто виртуальная функция не определяется в производном классе, то она и остается таковой, а значит производный класс тоже является абстрактным. При таком подходе можно реализовывать классы поэтапно:

```
class X {
public:
  virtual void f() = 0;
  virtual void g() = 0;
};

X b;   // ошибка: описание объекта абстрактного класса X

class Y : public X {
  void f();  // переопределение X::f
};

Y b;   // ошибка: описание объекта абстрактного класса Y

class Z : public Y {
  void g();  // переопределение X::g
};
```

```
Z c;  // нормально
```

Абстрактные классы нужны для задания интерфейса без уточнения каких-либо конкретных деталей реализации. Например, в операционной системе детали реализации драйвера устройства можно скрыть таким абстрактным классом:

```
class character_device {
public:
  virtual int open() = 0;
  virtual int close(const char*) = 0;
  virtual int read(const char*, int) =0;
  virtual int write(const char*, int) = 0;
  virtual int ioctl(int ...) = 0;
  // ...
};
```

Настоящие драйверы будут определяться как производные от класса `character_device`.

После введения абстрактного класса у нас есть все основные средства для того, чтобы написать законченную программу.

6.4 Пример законченной программы

Рассмотрим программу рисования геометрических фигур на экране. Она естественным образом распадается на три части:

1. монитор экрана: набор функций и структур данных низкого уровня для работы с экраном; оперирует только такими понятиями, как точки, линии;
2. библиотека фигур: множество определений фигур общего вида (например, прямоугольник, окружность) и стандартные функции для работы с ними;
3. прикладная программа: конкретные определения фигур, относящихся к задаче, и работающие с ними функции.

Как правило, эти три части программируются разными людьми в

разных организациях и в разное время, причем они обычно создаются в перечисленном порядке. При этом естественно возникают затруднения, поскольку, например, у разработчика монитора нет точного представления о том, для каких задач в конечном счете он будет использоваться. Наш пример будет отражать этот факт. Чтобы пример имел допустимый размер, библиотека фигур весьма ограничена, а прикладная программа тривиальна. Используется совершенно примитивное представление экрана, чтобы даже читатель, на машине которого нет графических средств, сумел поработать с этой программой. Можно легко заменить монитор экрана на более развитую программу, не изменяя при этом библиотеку фигур или прикладную программу.

6.4.1 Монитор экрана

Вначале было желание написать монитор экрана на C, чтобы еще больше подчеркнуть разделение между уровнями реализации. Но это оказалось утомительным, и поэтому выбрано компромиссное решение: стиль программирования, принятый в C (нет функций-членов, виртуальных функций, пользовательских операций и т.д.), но используются конструкторы, параметры функций полностью описываются и проверяются и т.д. Этот монитор очень напоминает программу на C, которую модифицировали, чтобы воспользоваться возможностями C++, но полностью переделывать не стали.

Экран представлен как двумерный массив символов и управляется функциями `put_point()` и `put_line()`. В них для связи с экраном используется структура point:

```
// файл screen.h

const int XMAX=40;
const int YMAX=24;

struct point {
    int x, y;
    point() { }
    point(int a,int b) { x=a; y=b; }
};
```

```
extern void put_point(int a, int b);
inline void put_point(point p) { put_point(p.x,p.y); }

extern void put_line(int, int, int, int);
extern void put_line(point a, point b)
  { put_line(a.x,a.y,b.x,b.y); }

extern void screen_init();
extern void screen_destroy();
extern void screen_refresh();
extern void screen_clear();

#include <iostream.h>
```

До вызова функций, выдающих изображение на экран (`put_...`), необходимо обратиться к функции инициализации экрана `screen_init()`. Изменения в структуре данных, описывающей экран, станут видимы на нем только после вызова функции обновления экрана `screen_refresh()`. Читатель может убедиться, что обновление экрана происходит просто с помощью копирования новых значений в массив, представляющий экран. Приведем функции и определения данных для управления экраном:

```
#include "screen.h"
#include <stream.h>

enum color { black='*', white=' ' };

char screen[XMAX] [YMAX];

void screen_init()
{
  for (int y=0; y<YMAX; y++)
     for (int x=0; x<XMAX; x++)
 screen[x] [y] = white;
}
```

Функция

```
void screen_destroy() { }
```

приведена просто для полноты картины. В реальных системах обычно нужны подобные функции уничтожения объекта.

Точки записываются, только если они попадают на экран:

```
inline int on_screen(int a, int b)  // проверка попадания
{
  return 0<=a && a <XMAX && 0<=b && b<YMAX;
}

void put_point(int a, int b)
{
   if (on_screen(a,b)) screen[a] [b] = black;
}
```

Для рисования прямых линий используется функция `put_line()`:

```
void put_line(int x0, int y0, int x1, int y1)
/*
   Нарисовать отрезок прямой (x0,y0) - (x1,y1).
   Уравнение прямой: b(x-x0) + a(y-y0) = 0.
   Минимизируется величина abs(eps),
   где eps = 2*(b(x-x0)) + a(y-y0).
   См. Newman, Sproull
   ``Principles of interactive Computer Graphics"
   McGraw-Hill, New York, 1979. pp. 33-34.
 */
 {
   register int dx = 1;
   int a = x1 - x0;
   if (a < 0) dx = -1, a = -a;

   register int dy = 1;
   int b = y1 - y0;
   if (b < 0) dy = -1, b = -b;

   int two_a = 2*a;
   int two_b = 2*b;
```

```
    int xcrit = -b + two_a;
    register int eps = 0;

    for (;;) {
      put_point(x0,y0);
      if (x0==x1 && y0==y1) break;
      if (eps <= xcrit) x0 +=dx, eps +=two_b;
      if (eps>=a || a<b) y0 +=dy, eps -=two_a;
    }
  }
```

Имеются функции для очистки и обновления экрана:

```
  void screen_clear() { screen_init(); }

  void screen_refresh()
  {
   for (int y=YMAX-1; 0<=y; y--) { // с верхней строки до нижней
      for (int x=0; x<XMAX; x++)   // от левого столбца до правого
   cout << screen[x] [y];
      cout << '\n';
   }
       }
```

Но нужно понимать, что все эти определения хранятся в некоторой библиотеке как результат работы транслятора, и изменить их нельзя.

6.4.2 Библиотека фигур

Начнем с определения общего понятия фигуры. Определение должно быть таким, чтобы им можно было воспользоваться (как базовым классом `shape`) в разных классах, представляющих все конкретные фигуры (окружности, квадраты и т.д.). Оно также должно позволять работать со всякой фигурой исключительно с помощью интерфейса, определяемого классом shape:

```
  struct shape {
    static shape* list;
```

```
    shape* next;

    shape() { next = list; list = this; }

    virtual point north() const = 0;
    virtual point south() const = 0;
    virtual point east() const = 0;
    virtual point west() const = 0;
    virtual point neast() const = 0;
    virtual point seast() const = 0;
    virtual point nwest() const = 0;
    virtual point swest() const = 0;

    virtual void draw() = 0;
    virtual void move(int, int) = 0;
};
```

Фигуры помещаются на экран функцией `draw()`, а движутся по нему с помощью `move()`. Фигуры можно помещать относительно друг друга, используя понятие точек контакта. Для обозначения точек контакта используются названия сторон света в компасе: north - север, ... , neast - северо-восток, ... , swest - юго-запад. Класс каждой конкретной фигуры сам определяет смысл этих точек и определяет, как рисовать фигуру. Конструктор `shape::shape()` добавляет фигуру к списку фигур `shape::list`. Для построения этого списка используется член `next`, входящий в каждый объект `shape`. Поскольку нет смысла в объектах типа общей фигуры, класс `shape` определен как абстрактный класс.

Для задания отрезка прямой нужно указать две точки или точку и целое. В последнем случае отрезок будет горизонтальным, а целое задает его длину. Знак целого показывает, где должна находиться заданная точка относительно конечной точки, т.е. слева или справа от нее:

```
class line : public shape {
/*
  отрезок прямой ["w", "e" ]
  north() определяет точку - `` выше центра отрезка и
  так далеко на север, как самая его северная точка"
*/
```

```
    point w, e;
        public:
    point north() const
      { return point((w.x+e.x)/2,e.y<w.y?w.y:e.y); }
    point south() const
      { return point((w.x+e.x)/2,e.y<w.y?e.y:w.y); }
    point east() const;
    point west() const;
    point neast() const;
    point seast() const;
    point nwest() const;
    point swest() const;

    void move(int a, int b)
       { w.x +=a; w.y +=b; e.x +=a; e.y +=b; }
    void draw() { put_line(w,e); }

    line(point a, point b) { w = a; e = b; }
    line(point a, int l) { w = point(a.x+l-1,a.y); e = a; }
        };
```

Аналогично определяется прямоугольник:

```
  class rectangle : public shape {
        /*   nw ------ n ----- ne
     |       |
     |       |
     w       c       e
     |       |
     |       |
     sw ------ s ----- se
        */
    point sw, ne;
        public:
    point north() const { return point((sw.x+ne.x)/2,ne.y); }
    point south() const { return point((sw.x+ne.x)/2,sw.y); }
    point east() const;
    point west() const;
    point neast() const { return ne; }
```

```
    point seast() const;
    point nwest() const;
    point swest() const { return sw; }

    void move(int a, int b)
    { sw.x+=a; sw.y+=b; ne.x+=a; ne.y+=b; }
    void draw();

    rectangle(point,point);
};
```

Прямоугольник строится по двум точкам. Конструктор усложняется, так как необходимо выяснять относительное положение этих точек:

```
rectangle::rectangle(point a, point b)
{
  if (a.x <= b.x) {
    if (a.y <= b.y) {
      sw = a;
      ne = b;
    }
    else {
      sw = point(a.x,b.y);
      ne = point(b.x,a.y);
    }
  }
  else {
    if (a.y <= b.y) {
      sw = point(b.x,a.y);
      ne = point(a.x,b.y);
    }
    else {
      sw = b;
      ne = a;
    }
  }
}
```

Чтобы нарисовать прямоугольник, надо нарисовать четыре отрезка:

```
void rectangle::draw()
{
  point nw(sw.x,ne.y);
  point se(ne.x,sw.y);
  put_line(nw,ne);
  put_line(ne,se);
  put_line(se,sw);
  put_line(sw,nw);
}
```

В библиотеке фигур есть определения фигур и функции для работы с ними:

```
void shape_refresh();      // нарисовать все фигуры
void stack(shape* p, const shape* q); // поместить p над q
```

Функция обновления фигур нужна, чтобы работать с нашим примитивным представлением экрана; она просто заново рисует все фигуры. Отметим, что эта функция не имеет понятия, какие фигуры она рисует:

```
void shape_refresh()
{
  screen_clear();
  for (shape* p = shape::list; p; p=p->next) p->draw();
  screen_refresh();
}
```

Наконец, есть одна действительно сервисная функция, которая рисует одну фигуру над другой. Для этого она определяет юг (`south()`) одной фигуры как раз над севером (`north()`) другой:

```
void stack(shape* p, const shape* q) // поместить p над q
{
  point n = q->north();
  point s = p->south();
  p->move(n.x-s.x,n.y-s.y+1);
}
```

Представим теперь, что эта библиотека является собственностью

некоторой фирмы, продающей программы, и, что она продает только заголовочный файл с определениями фигур и оттранслированные определения функций. Все равно вы сможете определить новые фигуры, воспользовавшись для этого купленными вами функциями.

6.4.3 Прикладная программа

Прикладная программа предельно проста. Определяется новая фигура `myshape` (если ее нарисовать, то она напоминает лицо), а затем приводится функция `main()`, в которой она рисуется со шляпой. Вначале дадим описание фигуры `myshape`:

```
#include "shape.h"

class myshape : public rectangle {
  line* l_eye;   // левый глаз
  line* r_eye;   // правый глаз
  line* mouth;   // рот
public:
  myshape(point, point);
  void draw();
  void move(int, int);
};
```

Глаза и рот являются отдельными независимыми объектами которые создает конструктор класса myshape:

```
myshape::myshape(point a, point b) : rectangle(a,b)
{
  int ll = neast().x-swest().x+1;
  int hh = neast().y-swest().y+1;
  l_eye = new line(
     point(swest().x+2,swest().y+hh*3/4),2);
  r_eye = new line(
     point(swest().x+ll-4,swest().y+hh*3/4),2);
  mouth = new line(
     point(swest().x+2,swest().y+hh/4),ll-4);
}
```

Объекты, представляющие глаза и рот, выдаются функцией `shape_refresh()` по отдельности. В принципе с ними можно работать независимо от объекта `my_shape`, к которому они принадлежат. Это один из способов задания черт лица для строящегося иерархически объекта `myshape`. Как это можно сделать иначе, видно из задания носа. Никакой тип "нос" не определяется, он просто дорисовывается в функции `draw()`:

```
void myshape::draw()
 {
  rectangle::draw();
  int a = (swest().x+neast().x)/2;
  int b = (swest().y+neast().y)/2;
  put_point(point(a,b));
 }
```

Движение фигуры `myshape` сводится к движению объекта базового класса `rectangle` и к движению вторичных объектов (`l_eye`, `r_eye` и `mouth`):

```
void myshape::move(int a, int b)
 {
  rectangle::move(a,b);
  l_eye->move(a,b);
  r_eye->move(a,b);
  mouth->move(a,b);
 }
```

Наконец, определим несколько фигур и будем их двигать:

```
int main()
 {
  screen_init();
  shape* p1 = new rectangle(point(0,0),point(10,10));
  shape* p2 = new line(point(0,15),17);
  shape* p3 = new myshape(point(15,10),point(27,18));
  shape_refresh();
  p3->move(-10,-10);
  stack(p2,p3);
```

```
    stack(p1,p2);
    shape_refresh();
    screen_destroy();
    return 0;
}
```

Вновь обратим внимание на то, что функции, подобные `shape_refresh()` и `stack()`, работают с объектами, типы которых были определены заведомо после определения этих функций (и, вероятно, после их трансляции).

Вот получившееся лицо со шляпой:

```
***********
*     *
*     *
*     *
*     *
*     *
*     *
*     *
***********
    ****************
***********
*     *
* **  ** *
*     *
*   *   *
*     *
* ******* *
*     *
***********
```

Для упрощения примера копирование и удаление фигур не обсуждалось.

6.5 Множественное наследование

У класса может быть несколько прямых базовых классов. Это значит, что в описании класса после : может быть указано более одного класса.

Рассмотрим задачу моделирования, в которой параллельные действия представлены стандартной библиотекой классов `task`, а сбор и выдачу информации обеспечивает библиотечный класс `displayed`. Тогда класс моделируемых объектов (назовем его `satellite`) можно определить так:

```
class satellite : public task, public displayed {
  // ...
};
```

Такое определение обычно называется множественным наследованием. Обратно, существование только одного прямого базового класса называется единственным наследованием.

Ко всем определенным в классе `satellite` операциям добавляется объединение операций классов `task` и `displayed`:

```
void f(satellite& s)
        {
 s.draw();    // displayed::draw()
 s.delay(10); // task::delay()
 s.xmit();    // satellite::xmit()
        }
```

С другой стороны, объект типа `satellite` можно передавать функциям с параметром типа `task` или `displayed`:

```
void highlight(displayed*);
void suspend(task*);

void g(satellite* p)
{
 highlight(p);   // highlight((displayed*)p)
 suspend(p);     // suspend((task*)p);
}
```

Очевидно, реализация этой возможности требует некоторого (простого) трюка от транслятора: нужно функциям с параметрами `task` и `displayed` передать разные части объекта типа `satellite`.

Для виртуальных функций, естественно, вызов и так выполнится правильно:

```
class task {
  // ...
  virtual pending() = 0;
};

class displayed {
  // ...
   virtual void draw() = 0;
};

class satellite : public task, public displayed {
   // ...
   void pending();
   void draw();
};
```

Здесь функции `satellite::draw()` и `satellite::pending()` для объекта типа `satellite` будут вызываться также, как если бы он был объектом типа `displayed` или `task`, соответственно.

Отметим, что ориентация только на единственное наследование ограничивает возможности реализации классов `displayed`, `task` и satellite. В таком случае класс `satellite` мог бы быть `task` или `displayed`, но не то и другое вместе (если, конечно, `task` не является производным от `displayed` или наоборот). В любом случае теряется гибкость.

6.5.1 Множественное вхождение базового класса

Возможность иметь более одного базового класса влечет за собой возможность неоднократного вхождения класса как базового. Допустим, классы `task` и `displayed` являются производными класса `link`, тогда в `satellite` он будет входить дважды:

```
class task : public link {
```

```
    // link используется для связывания всех
    // задач в список (список диспетчера)

    // ...
};

class displayed : public link {
    // link используется для связывания всех
    // изображаемых объектов (список изображений)

    // ...
};
```

Но проблем не возникает. Два различных объекта `link` используются для различных списков, и эти списки не конфликтуют друг с другом. Конечно, без риска неоднозначности нельзя обращаться к членам класса `link`, но как это сделать корректно, показано в следующем разделе.

Но можно привести примеры, когда общий базовый класс не должен представляться двумя различными объектами.

6.5.2 Разрешение неоднозначности

Естественно, у двух базовых классов могут быть функции-члены с одинаковыми именами:

```
class task {
    // ...
    virtual debug_info* get_debug();
};

class displayed {
    // ...
    virtual debug_info* get_debug();
};
```

При использовании класса `satellite` подобная неоднозначность функций должна быть разрешена:

```
void f(satellite* sp)
{
 debug_info* dip = sp->get_debug(); //ошибка: неоднозначность
 dip = sp->task::get_debug();       // нормально
 dip = sp->displayed::get_debug();  // нормально
}
```

Однако, явное разрешение неоднозначности хлопотно, поэтому для ее устранения лучше всего определить новую функцию в производном классе:

```
class satellite : public task, public derived {
  // ...
  debug_info* get_debug()
  {
    debug_info* dip1 = task:get_debug();
    debug_info* dip2 = displayed::get_debug();
    return dip1->merge(dip2);
  }
};
```

Тем самым локализуется информация из базовых для `satellite` классов. Поскольку `satellite::get_debug()` является переопределением функций `get_debug()` из обоих базовых классов, гарантируется, что именно она будет вызываться при всяком обращении к `get_debug()` для объекта типа `satellite`.

Транслятор выявляет коллизии имен, возникающие при определении одного и того же имени в более, чем одном базовом классе. Поэтому программисту не надо указывать какое именно имя используется, кроме случая, когда его использование действительно неоднозначно. Как правило использование базовых классов не приводит к коллизии имен. В большинстве случаев, даже если имена совпадают, коллизия не возникает, поскольку имена не используются непосредственно для объектов производного класса.

Если неоднозначности не возникает, излишне указывать имя базового класса при явном обращении к его члену. В частности, если множественное наследование не используется, вполне достаточно

использовать обозначение типа "где-то в базовом классе". Это позволяет программисту не запоминать имя прямого базового класса и спасает его от ошибок (впрочем, редких), возникающих при перестройке иерархии классов. Например:

```
void manager::print()
{
  employee::print();
  // ...
}
```

предполагается, что `employee` - прямой базовый класс для `manager`. Результат этой функции не изменится, если `employee` окажется косвенным базовым классом для `manager`, а в прямом базовом классе функции `print()` нет. Однако, кто-то мог бы следующим образом перестроить классы:

```
class employee {
  // ...
  virtual void print();
};

class foreman : public employee {
  // ...
  void print();
};

class manager : public foreman {
  // ...
  void print();
};
```

Теперь функция `foreman::print()` не будет вызываться, хотя почти наверняка предполагался вызов именно этой функции. С помощью небольшой хитрости можно преодолеть эту трудность:

```
class foreman : public employee {
  typedef employee inherited;
  // ...
```

```
  void print();
};

class manager : public foreman {
  typedef foreman inherited;
  // ...
  void print();
};

void manager::print()
{
  inherited::print();
  // ...
}
```

Правила областей видимости, в частности те, которые относятся к вложенным типам, гарантируют, что возникшие несколько типов `inherited` не будут конфликтовать друг с другом. В общем-то дело вкуса, считать решение с типом `inherited` наглядным или нет.

6.5.3 Виртуальные базовые классы

В предыдущих разделах множественное наследование рассматривалось как существенный фактор, позволяющий за счет слияния классов безболезненно интегрировать независимо создававшиеся программы. Это самое основное применение множественного наследования, и, к счастью (но не случайно), это самый простой и надежный способ его применения.

Иногда применение множественного наследования предполагает достаточно тесную связь между классами, которые рассматриваются как "братские" базовые классы. Такие классы-братья обычно должны проектироваться совместно. В большинстве случаев для этого не требуется особый стиль программирования, существенно отличающийся от того, который мы только что рассматривали. Просто на производный класс возлагается некоторая дополнительная работа. Обычно она сводится к переопределению одной или нескольких виртуальных функций. В некоторых случаях классы-братья должны

иметь общую информацию. Поскольку C++ - язык со строгим контролем типов, общность информации возможна только при явном указании того, что является общим в этих классах. Способом такого указания может служить виртуальный базовый класс.

Виртуальный базовый класс можно использовать для представления "головного" класса, который может конкретизироваться разными способами:

```
class window {
  // головная информация
  virtual void draw();
};
```

Для простоты рассмотрим только один вид общей информации из класса window - функцию `draw()`. Можно определять разные более развитые классы, представляющие окна (window). В каждом определяется своя (более развитая) функция рисования (draw):

```
class window_w_border : public virtual window {
    // класс "окно с рамкой"
    // определения, связанные с рамкой
    void draw();
};

class window_w_menu : public virtual window {
    // класс "окно с меню"
    // определения, связанные с меню
    void draw();
};
```

Теперь хотелось бы определить окно с рамкой и меню:

```
class window_w_border_and_menu
      : public virtual window,
      public window_w_border,
      public window_w_menu {
      // класс "окно с рамкой и меню"
      void draw();
};
```

Каждый производный класс добавляет новые свойства окна. Чтобы воспользоваться комбинацией всех этих свойств, мы должны гарантировать, что один и тот же объект класса window используется для представления вхождений базового класса window в эти производные классы. Именно это обеспечивает описание window во всех производных классах как виртуального базового класса.

В графе наследования каждый базовый класс с данным именем, который был указан как виртуальный, будет представлен единственным объектом этого класса. Напротив, каждый базовый класс, который при описании наследования не был указан как виртуальный, будет представлен своим собственным объектом.

Теперь надо написать все эти функции `draw()`. Это не слишком трудно, но для неосторожного программиста здесь есть ловушка. Сначала пойдем самым простым путем, который как раз к ней и ведет:

```
void window_w_border::draw()
  {
    window::draw();
    // рисуем рамку
  }

  void window_w_menu::draw()
  {
   window::draw();
   // рисуем меню
  }
```

Пока все хорошо. Все это очевидно, и мы следуем образцу определения таких функций при условии единственного наследования, который работал прекрасно. Однако, в производном классе следующего уровня появляется ловушка:

```
void window_w_border_and_menu::draw() // ловушка!
{
  window_w_border::draw();
  window_w_menu::draw();

  // теперь операции, относящиеся только
```

```
    // к окну с рамкой и меню
}
```

На первый взгляд все вполне нормально. Как обычно, сначала выполняются все операции, необходимые для базовых классов, а затем те, которые относятся собственно к производным классам. Но в результате функция `window::draw()` будет вызываться дважды! Для большинства графических программ это не просто излишний вызов, а порча картинки на экране. Обычно вторая выдача на экран затирает первую.

Чтобы избежать ловушки, надо действовать не так поспешно. Мы отделим действия, выполняемые базовым классом, от действий, выполняемых из базового класса. Для этого в каждом классе введем функцию `_draw()`, которая выполняет нужные только для него действия, а функция `draw()` будет выполнять те же действия плюс действия, нужные для каждого производного класса. Для класса `window` изменения сводятся к введению излишней функции:

```
class window {
 // головная информация
 void _draw();
 void draw();
};
```

Для производных классов эффект тот же:

```
class window_w_border : public virtual window {
    // класс "окно с рамкой"
    // определения, связанные с рамкой
    void _draw();
    void draw();
};

void window_w_border::draw()
{
    window::_draw();
    _draw();   // рисует рамку
};
```

Только для производного класса следующего уровня проявляется отличие функции, которое и позволяет обойти ловушку с повторным вызовом `window::draw()`, поскольку теперь вызывается `window::_draw()` и только один раз:

```
class window_w_border_and_menu
    : public virtual window,
    public window_w_border,
    public window_w_menu {

    void _draw();
    void draw();
};

void window_w_border_and_menu::draw()
{
  window::_draw();
  window_w_border::_draw();
  window_w_menu::_draw();

  _draw();  // теперь операции, относящиеся только
  // к окну с рамкой и меню
}
```

Не обязательно иметь обе функции `window::draw()` и `window::_draw()`, но наличие их позволяет избежать различных простых описок.

В этом примере класс window служит хранилищем общей для `window_w_border` и `window_w_menu` информации и определяет интерфейс для общения этих двух классов. Если используется единственное наследование, то общность информации в дереве классов достигается тем, что эта информация передвигается к корню дерева до тех пор, пока она не станет доступна всем заинтересованным в ней узловым классам. В результате легко возникает неприятный эффект: корень дерева или близкие к нему классы используются как пространство глобальных имен для всех классов дерева, а иерархия классов вырождается в множество несвязанных объектов.

Существенно, чтобы в каждом из классов-братьев переопределялись функции, определенные в общем виртуальном базовом классе. Таким образом каждый из братьев может получить свой вариант операций, отличный от других. Пусть в классе window есть общая функция ввода `get_input()`:

```
class window {
  // головная информация
  virtual void draw();
  virtual void get_input();
};
```

В одном из производных классов можно использовать эту функцию, не задумываясь о том, где она определена:

```
class window_w_banner : public virtual window {
   // класс "окно с заголовком"
   void draw();
   void update_banner_text();
};

void window_w_banner::update_banner_text()
{
  // ...
  get_input();
  // изменить текст заголовка
}
```

В другом производном классе функцию `get_input()` можно определять, не задумываясь о том, кто ее будет использовать:

```
class window_w_menu : public virtual window {
   // класс "окно с меню"
   // определения, связанные с меню
   void draw();
   void get_input(); // переопределяет window::get_input()
};
```

Все эти определения собираются вместе в производном классе следующего уровня:

```
class window_w_banner_and_menu
  : public virtual window,
  public window_w_banner,
  public window_w_menu
{
  void draw();
};
```

Контроль неоднозначности позволяет убедиться, что в классах-братьях определены разные функции:

```
class window_w_input : public virtual window {
  // ...
  void draw();
  void get_input();  // переопределяет window::get_input
};

class window_w_input_and_menu
  : public virtual window,
  public window_w_input,
  public window_w_menu
      {   // ошибка: оба класса window_w_input и
  // window_w_menu переопределяют функцию
  // window::get_input
  void draw();
      };
```

Транслятор обнаруживает подобную ошибку, а устранить неоднозначность можно обычным способом: ввести в классы `window_w_input` и `window_w_menu` функцию, переопределяющую "функцию-нарушителя", и каким-то образом устранить неоднозначность:

```
class window_w_input_and_menu
  : public virtual window,
  public window_w_input,
  public window_w_menu
      {
  void draw();
```

```
    void get_input();
        };
```

В этом классе `window_w_input_and_menu::get_input()` будет переопределять все функции `get_input()`.

6.6 Контроль доступа

Член класса может быть частным (private), защищенным (protected) или общим (public):

- Частный член класса `X` могут использовать только функции-члены и друзья класса `X`.
- Защищенный член класса `X` могут использовать только функции-члены и друзья класса `X`, а также функции-члены и друзья всех производных от `X` классов.
- Общий член можно использовать в любой функции.

Эти правила соответствуют делению обращающихся к классу функций на три вида: функции, реализующие класс (его друзья и члены), функции, реализующие производный класс (друзья и члены производного класса) и все остальные функции.

Контроль доступа применяется единообразно ко всем именам. На контроль доступа не влияет, какую именно сущность обозначает имя. Это означает, что частными могут быть функции-члены, константы и т.д. наравне с частными членами, представляющими данные:

```
class X {
 private:
   enum { A, B };
   void f(int);
   int a;
 };

 void X::f(int i)
 {
   if (i<A) f(i+B);
```

```
  a++;
}

void g(X& x)
{
  int i = X::A; // ошибка: X::A частный член
  x.f(2);       // ошибка: X::f частный член
  x.a++;        // ошибка: X::a частный член
}
```

6.6.1 Защищенные члены

Дадим пример защищенных членов, вернувшись к классу window из предыдущего раздела. Здесь функции `_draw()` предназначались только для использования в производных классах, поскольку предоставляли неполный набор возможностей, а поэтому не были достаточны удобны и надежны для общего применения. Они были как бы строительным материалом для более развитых функций. С другой стороны, функции `draw()` предназначались для общего применения. Это различие можно выразить, разбив интерфейсы классов window на две части - защищенный интерфейс и общий интерфейс:

```
class window {
public:
  virtual void draw();
  // ...
protected:
  void _draw();
  // другие функции, служащие строительным материалом
private:
  // представление класса
};
```

Такое разбиение можно проводить и в производных классах, таких, как `window_w_border` или `window_w_menu`.

Префикс _ используется в именах защищенных функций, являющихся частью реализации класса, по общему правилу: имена, начинающиеся с

_ , не должны присутствовать в частях программы, открытых для общего использования. Имен, начинающихся с двойного символа подчеркивания, лучше вообще избегать (даже для членов).

Вот менее практичный, но более подробный пример:

```
class X {
// по умолчанию частная часть класса
   int priv;
protected:
   int prot;
public:
   int publ;
   void m();
};
```

Для члена `X::m` доступ к членам класса неограничен:

```
void X::m()
{
  priv = 1;   // нормально
  prot = 2;   // нормально
  publ = 3;   // нормально
}
```

Член производного класса имеет доступ только к общим и защищенным членам:

```
class Y : public X {
   void mderived();
};

Y::mderived()
{
  priv = 1;    // ошибка: priv частный член
  prot = 2;    // нормально: prot защищенный член, а
   // mderived() член производного класса Y
  publ = 3;    // нормально: publ общий член
}
```

В глобальной функции доступны только общие члены:

```
void f(Y* p)
{
  p->priv = 1;  // ошибка: priv частный член
  p->prot = 2;  // ошибка: prot защищенный член, а f()
    // не друг или член классов X и Y
  p->publ = 3;  // нормально: publ общий член
}
```

6.6.2 Доступ к базовым классам

Подобно члену базовый класс можно описать как частный, защищенный или общий:

```
class X {
public:
  int a;
  // ...
};

class Y1 : public X {  };
class Y2 : protected X { };
class Y3 : private X { };
```

Поскольку `X` - общий базовый класс для `Y1`, в любой функции, если есть необходимость, можно (неявно) преобразовать `Y1*` в `X*`, и притом в ней будут доступны общие члены класса `X`:

```
void f(Y1* py1, Y2* py2, Y3* py3)
{
  X* px = py1;  // нормально: X - общий базовый класс Y1
  py1->a = 7;   // нормально
  px = py2;     // ошибка: X - защищенный базовый класс Y2
  py2->a = 7;   // ошибка
  px = py3;     // ошибка: X - частный базовый класс Y3
  py3->a = 7;   // ошибка
}
```

Теперь пусть описаны

```
class Y2 : protected X { };
class Z2 : public Y2 { void f(); };
```

Поскольку `X` - защищенный базовый класс `Y2`, только друзья и члены `Y2`, а также друзья и члены любых производных от `Y2` классов (в частности `Z2`) могут при необходимости преобразовывать (неявно) `Y2*` в `X*`. Кроме того они могут обращаться к общим и защищенным членам класса `X`:

```
void Z2::f(Y1* py1, Y2* py2, Y3* py3)
{
  X* px = py1; // нормально: X - общий базовый класс Y1
  py1->a = 7; // нормально
  px = py2;   // нормально: X - защищенный базовый класс Y2,
  // а Z2 - производный класс Y2
  py2->a = 7; // нормально
  px = py3;   // ошибка: X - частный базовый класс Y3
  py3->a = 7; // ошибка
}
```

Наконец, рассмотрим:

```
class Y3 : private X { void f(); };
```

Поскольку `X` - частный базовый класс `Y3`, только друзья и члены `Y3` могут при необходимости преобразовывать (неявно) `Y3*` в `X*`. Кроме того они могут обращаться к общим и защищенным членам класса `X`:

```
void Y3::f(Y1* py1, Y2* py2, Y3* py3)
{
  X* px = py1;  // нормально: X - общий базовый класс Y1
  py1->a = 7;   // нормально
  px = py2;     // ошибка: X - защищенный базовый класс Y2
  py2->a = 7;   // ошибка
  px = py3;     // нормально: X - частный базовый класс Y3,
    // а Y3::f член Y3
  py3->a = 7;   // нормально
```

```
}
```

6.7 Свободная память

Если определить функции `operator new()` и `operator delete()`, управление памятью для класса можно взять в свои руки. Это также можно, (а часто и более полезно), сделать для класса, служащего базовым для многих производных классов. Допустим, нам потребовались свои функции размещения и освобождения памяти для класса `employee` (§ 6.2.5) и всех его производных классов:

```
class employee {
  // ...
public:
  void* operator new(size_t);
  void operator delete(void*, size_t);
};

void* employee::operator new(size_t s)
{
  // отвести память в `s' байтов
  // и возвратить указатель на нее
}

void employee::operator delete(void* p, size_t s)
{
  // `p' должно указывать на память в `s' байтов,
  // отведенную функцией employee::operator new();
  // освободить эту память для повторного использования
}
```

Назначение до сей поры загадочного параметра типа `size_t` становится очевидным. Это - размер освобождаемого объекта. При удалении простого служащего этот параметр получает значение sizeof(`employee`), а при удалении управляющего - `sizeof` (`manager`). Поэтому собственные функции классы для размещения могут не хранить размер каждого размещаемого объекта. Конечно, они могут хранить эти размеры (подобно функциям размещения общего

назначения) и игнорировать параметр `size_t` в вызове operator `delete()`, но тогда вряд ли они будут лучше, чем функции размещения и освобождения общего назначения.

Как транслятор определяет нужный размер, который надо передать функции `operator delete()`? Пока тип, указанный в operator `delete()`, соответствует истинному типу объекта, все просто; но рассмотрим такой пример:

```
class manager : public employee {
  int level;
  // ...
};

void f()
{
  employee* p = new manager; // проблема
  delete p;
}
```

В этом случае транслятор не сможет правильно определить размер. Как и в случае удаления массива, нужна помощь программиста. Он должен определить виртуальный деструктор в базовом классе `employee`:

```
class employee {
  // ...
public:
  // ...
  void* operator new(size_t);
  void operator delete(void*, size_t);
  virtual ~employee();
};
```

Даже пустой деструктор решит нашу проблему:

```
employee::~employee() { }
```

Теперь освобождение памяти будет происходить в деструкторе (а в нем размер известен), а любой производный от `employee` класс также будет вынужден определять свой деструктор (тем самым будет

установлен нужный размер), если только пользователь сам не определит его. Теперь следующий пример пройдет правильно:

```
void f()
{
 employee* p = new manager; // теперь без проблем
 delete p;
}
```

Размещение происходит с помощью (созданного транслятором) вызова

```
employee::operator new(sizeof(manager))
```

а освобождение с помощью вызова

```
employee::operator delete(p,sizeof(manager))
```

Иными словами, если нужно иметь корректные функции размещения и освобождения для производных классов, надо либо определить виртуальный деструктор в базовом классе, либо не использовать в функции освобождения параметр `size_t`. Конечно, можно было при проектировании языка предусмотреть средства, освобождающие пользователя от этой проблемы. Но тогда пользователь "освободился" бы и от определенных преимуществ более оптимальной, хотя и менее надежной системы.

В общем случае, всегда есть смысл определять виртуальный деструктор для всех классов, которые действительно используются как базовые, т.е. с объектами производных классов работают и, возможно, удаляют их, через указатель на базовый класс:

```
class X {
 // ...
public:
 // ...
 virtual void f(); // в X есть виртуальная функция, поэтому
      // определяем виртуальный деструктор
 virtual ~X();
};
```

6.7.1 Виртуальные конструкторы

Узнав о виртуальных деструкторах, естественно спросить: "Могут ли конструкторы то же быть виртуальными?" Если ответить коротко - нет. Можно дать более длинный ответ: "Нет, но можно легко получить требуемый эффект".

Конструктор не может быть виртуальным, поскольку для правильного построения объекта он должен знать его истинный тип. Более того, конструктор - не совсем обычная функция. Он может взаимодействовать с функциями управления памятью, что невозможно для обычных функций. От обычных функций-членов он отличается еще тем, что не вызывается для существующих объектов. Следовательно нельзя получить указатель на конструктор.

Но эти ограничения можно обойти, если определить функцию, содержащую вызов конструктора и возвращающую построенный объект. Это удачно, поскольку нередко бывает нужно создать новый объект, не зная его истинного типа. Например, при трансляции иногда возникает необходимость сделать копию дерева, представляющего разбираемое выражение. В дереве могут быть узлы выражений разных видов. Допустим, что узлы, которые содержат повторяющиеся в выражении операции, нужно копировать только один раз. Тогда нам потребуется виртуальная функция размножения для узла выражения.

Как правило "виртуальные конструкторы" являются стандартными конструкторами без параметров или конструкторами копирования, параметром которых служит тип результата:

```
class expr {
  // ...
public:
  expr();  // стандартный конструктор
  virtual expr* new_expr() { return new expr(); }
};
```

Виртуальная функция `new_expr()` просто возвращает стандартно инициализированный объект типа `expr`, размещенный в свободной памяти. В производном классе можно переопределить функцию

`new_expr()` так, чтобы она возвращала объект этого класса:

```
class conditional : public expr {
  // ...
public:
  conditional();  // стандартный конструктор
  expr* new_expr() { return new conditional(); }
};
```

Это означает, что, имея объект класса `expr`, пользователь может создать объект в "точности такого же типа":

```
void user(expr* p1, expr* p2)
{
  expr* p3 = p1->new_expr();
  expr* p4 = p2->new_expr();
  // ...
}
```

Переменным `p3` и `p4` присваиваются указатели неизвестного, но подходящего типа.

Тем же способом можно определить виртуальный конструктор копирования, называемый операцией размножения, но надо подойти более тщательно к специфике операции копирования:

```
class expr {
    // ...
    expr* left;
    expr* right;
public:
    // ...
      // копировать `s' в `this'
    inline void copy(expr* s);
      // создать копию объекта, на который смотрит this
    virtual expr* clone(int deep = 0);
};
```

Параметр `deep` показывает различие между копированием собственно объекта (поверхностное копирование) и копированием всего поддерева,

корнем которого служит объект (глубокое копирование). Стандартное значение 0 означает поверхностное копирование.

Функцию `clone()` можно использовать, например, так:

```
void fct(expr* root)
{
  expr* c1 = root->clone(1);  // глубокое копирование
  expr* c2 = root->clone();   // поверхностное копирование
  // ...
}
```

Являясь виртуальной, функция clone() способна размножать объекты любого производного от expr класса.

Настоящее копирование можно определить так:

```
void expr::copy(expression* s, int deep)
{
  if (deep == 0) { // копируем только члены
    *this = *s;
  }
  else { // пройдемся по указателям:
    left = s->clone(1);
    right = s->clone(1);
    // ...
  }
}
```

Функция `expr::clone()` будет вызываться только для объектов типа `expr` (но не для производных от expr классов), поэтому можно просто разместить в ней и возвратить из нее объект типа expr, являющийся собственной копией:

```
expr* expr::clone(int deep)
{
  expr* r = new expr();  // строим стандартное выражение
  r->copy(this,deep);    // копируем `*this' в `r'
  return r;
}
```

Такую функцию `clone()` можно использовать для производных от expr классов, если в них не появляются члены-данные (а это как раз типичный случай):

```
class arithmetic : public expr {
  // ...
  // новых членов-данных нет =>
  // можно использовать уже определенную функцию clone
};
```

С другой стороны, если добавлены члены-данные, то нужно определять собственную функцию `clone()`:

```
class conditional : public expression {
    expr* cond;
public:
    inline void copy(cond* s, int deep = 0);
    expr* clone(int deep = 0);
    // ...
};
```

Функции `copy()` и `clone()` определяются подобно своим двойникам из `expression`:

```
expr* conditional::clone(int deep)
{
  conditional* r = new conditional();
  r->copy(this,deep);
  return r;
}

void conditional::copy(expr* s, int deep)
{
  if (deep == 0) {
     *this = *s;
  }
  else {
     expr::copy(s,1);  // копируем часть expr
     cond = s->cond->clone(1);
```

```
    }
}
```

Определение последней функции показывает отличие настоящего копирования в `expr::copy()` от полного размножения в `expr::clone()` (т.е. создания нового объекта и копирования в него). Простое копирование оказывается полезным для определения более сложных операций копирования и размножения. Различие между `copy()` и `clone()` эквивалентно различию между операцией присваивания и конструктором копирования и эквивалентно различию между функциями `_draw()` и `draw()`. Отметим, что функция `copy()` не является виртуальной. Ей и не надо быть таковой, поскольку виртуальна вызывающая ее функция `clone()`. Очевидно, что простые операции копирования можно также определять как функции-подстановки.

6.7.2 Указание размещения

По умолчанию операция new создает указанный ей объект в свободной памяти. Как быть, если надо разместить объект в определенном месте? Этого можно добиться переопределением операции размещения. Рассмотрим простой класс:

```
class X {
// ...
public:
X(int);
 // ...
};
```

Объект можно разместить в любом месте, если ввести в функцию размещения дополнительные параметры:

```
// операция размещения в указанном месте:
void* operator new(size_t, void* p) { return p; }
```

и задав эти параметры для операции `new` следующим образом:

```
char buffer[sizeof(X)];

void f(int i)
{
  X* p = new(buffer) X(i); // разместить X в buffer
  // ...
}
```

Функция `operator new()`, используемая операцией `new`, выбирается согласно правилам сопоставления параметров. Все функции `operator new()` должны иметь первым параметром `size_t`. Задаваемый этим параметром размер неявно передается операцией `new`.

Определенная нами функция operator new() с задаваемым размещением является самой простой из функций подобного рода. Можно привести другой пример функции размещения, выделяющей память из некоторой заданной области:

```
class Arena {
  // ...
  virtual void* alloc(size_t) = 0;
  virtual void free(void*) = 0;
};

void operator new(size_t sz, Arena* a)
{
  return a->alloc(sz);
}
```

Теперь можно отводить память для объектов произвольных типов из различных областей (Arena):

```
extern Arena* Persistent; // постоянная память
extern Arena* Shared;     // разделяемая память

void g(int i)
{
  X* p = new(Persistent) X(i); // X в постоянной памяти
```

```
  X* q = new(Shared) X(i);     // X в разделяемой памяти
  // ...
}
```

Если мы помещаем объект в область памяти, которая непосредственно не управляется стандартными функциями распределения свободной памяти, то надо позаботиться о правильном уничтожении объекта. Основным средством здесь является явный вызов деструктора:

```
void h(X* p)
{
  p->~X();     // вызов деструктора
  Persistent->free(p); // освобождение памяти
}
```

Заметим, что явных вызовов деструкторов, как и глобальных функций размещения специального назначения, следует, по возможности, избегать. Бывают случаи, когда обойтись без них трудно, но новичок должен трижды подумать, прежде чем использовать явный вызов деструктора, и должен сначала посоветоваться с более опытным коллегой.

Перегрузка операций

Лекция содержит описание механизма перегрузки операций в C++. Программист может задать интерпретацию операций, когда они применяются к объектам определенного класса. Помимо арифметических, логических и операций отношения можно переопределить вызов функций (), индексацию [], косвенное обращение ->, а также присваивание и инициализацию. Можно определить явные и скрытые преобразования между пользовательскими и основными типами. Показано, как определить класс, объект которого можно копировать и уничтожать только с помощью специальных, определенных пользователем функций.

7.1 Введение

Если я выбираю слово, оно значит только то, что я решу, ни больше и ни меньше.

Обычно в программах используются объекты, являющиеся конкретным представлением абстрактных понятий. Например, в C++ тип данных int вместе с операциями +, -, *, / и т.д. реализует (хотя и ограниченно) математическое понятие целого. Обычно с понятием связывается набор действий, которые реализуются в языке в виде основных операций над объектами, задаваемых в сжатом, удобном и привычном виде.

К сожалению, в языках программирования непосредственно представляется только малое число понятий. Так, понятия комплексных чисел, алгебры матриц, логических сигналов и строк в C++ не имеют непосредственного выражения. Возможность задать представление сложных объектов вместе с набором операций, выполняемых над такими объектами, реализуют в C++ классы. Позволяя программисту определять операции над объектами классов, мы получаем более удобную и традиционную систему обозначений для работы с этими объектами по сравнению с той, в которой все операции задаются как обычные функции. Приведем пример:

```
class complex {
  double re, im;
```

```
public:
  complex(double r, double i) { re=r; im=i; }
  friend complex operator+(complex, complex);
  friend complex operator*(complex, complex);
};
```

Здесь приведена простая реализация понятия комплексного числа, когда оно представлено парой чисел с плавающей точкой двойной точности, с которыми можно оперировать только с помощью операций `+` и `*`.

Интерпретацию этих операций задает программист в определениях функций с именами `operator+` и `operator*`. Так, если `b` и `c` имеют тип `complex`, то `b+c` означает (по определению) `operator+(b,c)`. Теперь можно приблизиться к привычной записи комплексных выражений:

```
void f()
{
  complex a = complex(1,3.1);
  complex b = complex(1.2,2);
  complex c = b;

  a = b+c;
  b = b+c*a;
  c = a*b+complex(1,2);
}
```

Сохраняются обычные приоритеты операций, поэтому второе выражение выполняется как `b=b+(c*a)`, а не как `b=(b+c)*a`.

7.2 Операторные функции

Можно описать функции, определяющие интерпретацию следующих операций:

```
+   -   *   /   %   ^   &   |   ~   !
=   <   >   +=  -=  *=  /=  %=  ^=  &=
|=  <<  >>  >>= <<= ==  !=  <=  >=  &&
||  ++  --  ->* ,   ->  []  ()  new delete
```

Последние пять операций означают: косвенное обращение (§ 7.9), индексацию (§ 7.7), вызов функции (§ 7.8), размещение в свободной памяти и освобождение (§ 3.2.6). Нельзя изменить приоритеты этих операций, равно как и синтаксические правила для выражений. Так, нельзя определить унарную операцию `%`, также как и бинарную операцию !. Нельзя ввести новые лексемы для обозначения операций, но если набор операций вас не устраивает, можно воспользоваться привычным обозначением вызова функции. Поэтому используйте `pow()`, а не `**`.

Эти ограничения можно счесть драконовскими, но более свободные правила легко приводят к неоднозначности. Допустим, мы определим операцию `**` как возведение в степень, что на первый взгляд кажется очевидной и простой задачей. Но если как следует подумать, то возникают вопросы: должны ли операции ** выполняться слева направо (как в Фортране) или справа налево (как в Алголе)? Как интерпретировать выражение `a**p` как `a*(*p)` или как `(a)**(p)`?

Именем операторной функции является служебное слово operator, за которым идет сама операция, например, operator<<. Операторная функция описывается и вызывается как обычная функция. Использование символа операции является просто краткой формой записи вызова операторной функции:

```
void f(complex a, complex b)
{
  complex c = a + b;          // краткая форма
  complex d = operator+(a,b); // явный вызов
}
```

С учетом приведенного описания типа complex инициализаторы в этом примере являются эквивалентными.

7.2.1 Бинарные и унарные операции

Бинарную операцию можно определить как функцию-член с одним параметром, или как глобальную функцию с двумя параметрами. Значит,

для любой бинарной операции @ выражение aa @ bb интерпретируется либо как aa.operator@(bb), либо как operator@(aa,bb). Если определены обе функции, то выбор интерпретации происходит по правилам сопоставления параметров (§ 4.13.2). Префиксная или постфиксная унарная операция может определяться как функция-член без параметров, или как глобальная функция с одними параметром. Для любой префиксной унарной операции @ выражение @aa интерпретируется либо как aa.operator@(), либо как operator@(aa). Если определены обе функции, то выбор интерпретации происходит по правилам сопоставления параметров (§ 4.13.2). Для любой постфиксной унарной операции @ выражение aa@ интерпретируется либо как aa.operator@(int), либо как operator@(aa,int). Подробно это объясняется в § 7.10. Если определены обе функции, то выбор интерпретации происходит по правилам сопоставления параметров (§ 13.2). Операцию можно определить только в соответствии с синтаксическими правилами, имеющимися для нее в грамматике C++.

В частности, нельзя определить % как унарную операцию, a + как тернарную. Проиллюстрируем сказанное примерами:

```
class X {
  // члены (неявно используется указатель `this'):

  X* operator&();        // префиксная унарная операция &
                         // (взятие адреса)
  X operator&(X);        // бинарная операция & (И поразрядное)
  X operator++(int);     // постфиксный инкремент
  X operator&(X,X);      // ошибка: & не может быть тернарной
  X operator/();         // ошибка: / не может быть унарной
};

                  // глобальные функции (обычно друзья)

X operator-(X);          // префиксный унарный минус
X operator-(X,X);        // бинарный минус
X operator--(X&,int);    // постфиксный декремент
X operator-();           // ошибка: нет операнда
```

```
X operator-(X,X,X);     // ошибка: тернарная операция
X operator%(X);         // ошибка: унарная операция %
```

Операция [] описывается в § 7.7, операция () в § 7.8, операция -> в § 7.9, а операции ++ и -- в § 7.10.

7.2.2 Предопределенные свойства операций

Используется только несколько предположений о свойствах пользовательских операций. В частности, `operator=`, `operator[]`, `operator()` и `operator->` должны быть нестатическими функциями-членами. Этим обеспечивается то, что первый операнд этих операций является адресом.

Для некоторых встроенных операций их интерпретация определяется как комбинация других операций, выполняемых над теми же операндами.

Так, если a типа `int`, то `++a` означает `a+=1`, что в свою очередь означает `a=a+1`. Такие соотношения не сохраняются для пользовательских операций, если только пользователь специально не определил их с такой целью. Так, определение operator `+=()` для типа `complex` нельзя вывести из определений `complex::operator+()` и `complex operator=()`.

По исторической случайности оказалось, что операции = (присваивание), &(взятие адреса) и , (операция запятая) обладают предопределенными свойствами для объектов классов. Но можно закрыть от произвольного пользователя эти свойства, если описать эти операции как частные:

```
class X {
  // ...
private:
  void operator=(const X&);
  void operator&();
  void operator,(const X&);
```

```
    // ...
};

void f(X a, X b)
{
    a= b;   // ошибка: операция = частная
    &a;     // ошибка: операция & частная
    a,b     // ошибка: операция , частная
}
```

С другой стороны, можно наоборот придать с помощью соответствующих определений этим операциям иное значение.

7.2.3 Операторные функции и пользовательские типы

Операторная функция должна быть либо членом, либо иметь по крайней мере один параметр, являющийся объектом класса (для функций, переопределяющих операции new и delete, это не обязательно). Это правило гарантирует, что пользователь не сумеет изменить интерпретацию выражений, не содержащих объектов пользовательского типа. В частности, нельзя определить операторную функцию, работающую только с указателями. Этим гарантируется, что в C++ возможны расширения, но не мутации (не считая операций =, &, и , для объектов класса).

Операторная функция, имеющая первым параметр основного типа, не может быть функцией-членом. Так, если мы прибавляем комплексную переменную aa к целому 2, то при подходящем описании функции-члена aa+2 можно интерпретировать как `aa.operator+(2)`, но `2+aa` так интерпретировать нельзя, поскольку не существует класса `int`, для которого `+` определяется как `2.operator+(aa)`. Даже если бы это было возможно, для интерпретации `aa+2` и `2+aa` пришлось иметь дело с двумя разными функциями-членами. Этот пример тривиально записывается с помощью функций, не являющихся членами.

Каждое выражение проверяется для выявления неоднозначностей. Если пользовательские операции задают возможную интерпретацию выражения, оно проверяется в соответствии с правилами § R.13.2.

7.3 Пользовательские операции преобразования типа

Описанная во введении реализация комплексного числа является слишком ограниченной, чтобы удовлетворить кого-нибудь, и ее надо расширить. Делается простым повторением описаний того же вида, что уже были применены:

```
class complex {
  double re, im;
public:
  complex(double r, double i) { re=r; im=i; }

  friend complex operator+(complex, complex);
  friend complex operator+(complex, double);
  friend complex operator+(double, complex);

  friend complex operator-(complex, complex);
  friend complex operator-(complex, double);
  friend complex operator-(double, complex);
  complex operator-();  // унарный -

  friend complex operator*(complex, complex);
  friend complex operator*(complex, double);
  friend complex operator*(double, complex);

  // ...
};
```

Имея такое определение комплексного числа, можно писать:

```
void f()
{
  complex a(1,1), b(2,2), c(3,3), d(4,4), e(5,5);
  a = -b-c;
  b = c*2.0*c;
  c = (d+e)*a;
}
```

Все-таки утомительно, как мы это только что делали для `operator*`

() писать для каждой комбинации complex и double свою функцию. Более того, разумные средства для комплексной арифметики должны предоставлять десятки таких функций (посмотрите, например, как описан тип complex в <complex.h>).

7.3.1 Конструкторы

Вместо того, чтобы описывать несколько функций, можно описать конструктор, который из параметра double создает complex:

```
class complex {
  // ...
  complex(double r) { re=r; im=0; }
};
```

Этим определяется как получить complex, если задан double. Это традиционный способ расширения вещественной прямой до комплексной плоскости.

Конструктор с единственным параметром не обязательно вызывать явно:

```
complex z1 = complex(23);
complex z2 = 23;
```

Обе переменные z1 и z2 будут инициализироваться вызовом complex(23).

Конструктор является алгоритмом создания значения заданного типа. Если требуется значение некоторого типа и существует строящий его конструктор, параметром которого является это значение, то тогда этот конструктор и будет использоваться. Так, класс complex можно было описать следующим образом:

```
class complex {
  double re, im;
public:
  complex(double r, double i =0) { re=r; im=i; }
```

```
    friend complex operator+(complex, complex);
    friend complex operator*(complex, complex);

    complex operator+=(complex);
    complex operator*=(complex);

    // ...
};
```

Все операции над комплексными переменными и целыми константами с учетом этого описания становятся законными. Целая константа будет интерпретироваться как комплексное число с мнимой частью, равной нулю. Так, `a=b*2` означает:

```
a = operator*(b, complex( double(2), double(0) ) )
```

Новые версии операций таких, как `+`, имеет смысл определять только, если практика покажет, что повышение эффективности за счет отказа от преобразований типа стоит того. Например, если выяснится, что операция умножения комплексной переменной на вещественную константу является критичной, то к множеству операций можно добавить `operator*=(double)`:

```
class complex {
  double re, im;
public:
  complex(double r, double i =0) { re=r; im=i; }

  friend complex operator+(complex, complex);
  friend complex operator*(complex, complex);

  complex& operator+=(complex);
  complex& operator*=(complex);
  complex& operator*=(double);

  // ...
};
```

Операции присваивания типа *= и += могут быть очень полезными для работы с пользовательскими типами, поскольку обычно запись с ними короче, чем с их обычными "двойниками" * и +, а кроме того они могут повысить скорость выполнения программы за счет исключения временных переменных:

```
inline complex& complex::operator+=(complex a)
{
  re += a.re;
  im += a.im;
  return *this;
}
```

При использовании этой функции не требуется временной переменной для хранения результата, и она достаточно проста, чтобы транслятор мог "идеально" произвести подстановку тела. Такие простые операции как сложение комплексных тоже легко задать непосредственно:

```
inline complex operator+(complex a, complex b)
{
  return complex(a.re+b.re, a.im+b.im);
}
```

Здесь в операторе `return` используется конструктор, что дает транслятору ценную подсказку на предмет оптимизации. Но для более сложных типов и операций, например таких, как умножение матриц, результат нельзя задать как одно выражение, тогда операции * и + проще реализовать с помощью *= и +=, и они будут легче поддаваться оптимизации:

```
matrix& matrix::operator*=(const matrix& a)
{
  // ...
  return *this;
}

matrix operator*(const matrix& a, const matrix& b)
{
  matrix prod = a;
```

```
    prod *= b;
    return prod;
  }
```

Отметим, что в определенной подобным образом операции не нужно никаких особых прав доступа к классу, к которому она применяется, т.е. эта операция не должна быть другом или членом этого класса.

Пользовательское преобразование типа применяется только в том случае, если оно единственное(§ 7.3.3).

Построенный в результате явного или неявного вызова конструктора, объект является автоматическим, и уничтожается при первой возможности,- как правило сразу после выполнения оператора, в котором он был создан.

7.3.2 Операции преобразования

Конструктор удобно использовать для преобразования типа, но возможны нежелательные последствия:

1. Неявные преобразования от пользовательского типа к основному невозможны (поскольку основные типы не являются классами).
2. Нельзя задать преобразование из нового типа в старый, не изменяя описания старого типа.
3. Нельзя определить конструктор с одним параметром, не определив тем самым и преобразование типа.

Последнее не является большой проблемой, а первые две можно преодолеть, если определить операторную функцию преобразования для исходного типа. Функция-член `X::operator T()`, где `T` - имя типа, определяет преобразование типа `X` в `T`. Например, можно определить тип `tiny` (крошечный), значения которого находятся в диапазоне 0..63, и этот тип может в арифметических операциях практически свободно смешиваться с целыми:

```
class tiny {
  char v;
```

```
  void assign(int i)
  { if (i>63) { error("выход из диапазона"); v=i&~63;  }
    v=i;
  }
public:
  tiny(int i) { assign(i) }
  tiny(const tiny& t) { v = t.v; }
  tiny& operator=(const tiny& t)
    { v = t.v; return *this; }
  tiny& operator=(int i) { assign(i); return *this; }
  operator int() { return v; }
};
```

Попадание в диапазон проверяется как при инициализации объекта `tiny`, так и в присваивании ему `int`. Один объект `tiny` можно присвоить другому без контроля диапазона. Для выполнения обычных операций с целыми для переменных типа `tiny` определяется функция `tiny::operator int()`, производящая неявное преобразование типа из `tiny` в `int`. Там, где требуется `int`, а задана переменная типа `tiny`, используется преобразованное к int значение:

```
void main()
{
  tiny c1 = 2;
  tiny c2 = 62;
  tiny c3 = c2 -c1;  // c3 = 60
  tiny c4 = c3;      // контроля диапазона нет (он не нужен)
  int i = c1 + c2;   // i = 64
  c1 = c2 + 2 * c1;  // выход из диапазона: c1 = 0 (а не 66)
  c2 = c1 - i;       // выход из диапазона: c2 = 0
  c3 = c2;           // контроля диапазона нет (он не нужен)
}
```

Более полезным может оказаться вектор из объектов `tiny`, поскольку он позволяет экономить память. Чтобы такой тип было удобно использовать, можно воспользоваться операцией индексации [].

Пользовательские операции преобразования типа могут пригодиться для работы с типами, реализующими нестандартные представления

чисел (арифметика с основанием 100, арифметика чисел с фиксированной точкой, представление в двоично-десятичной записи и т.д.). При этом обычно приходится переопределять такие операции, как + и *.

Особенно полезными функции преобразования типа оказываются для работы с такими структурами данных, для которых чтение (реализованное как операция преобразования) является тривиальным, а присваивание и инициализация существенно более сложные операции.

Функции преобразования нужны для типов `istream` и `ostream`, чтобы стали возможными, например, такие операторы:

```
while (cin>>x) cout<<x;
```

Операция ввода `cin>>x` возвращает значение `istream&`. Оно неявно преобразуется в значение, показывающее состояние потока cin, которое затем проверяется в операторе `while` (см. § 10.3.2). Но все-таки определять неявное преобразование типа, при котором можно потерять преобразуемое значение, как правило, плохое решение.

Избыток таких операций может вызывать большое число неоднозначностей.

Транслятор обнаруживает эти неоднозначности, но разрешить их может быть совсем непросто. Возможно вначале лучше для преобразований использовать поименованные функции, например, `X::intof()`, и только после того, как такую функцию как следуют опробуют, и явное преобразование типа будет сочтено неэлегантным решением, можно заменить операторной функцией преобразования `X::operator int()`.

7.3.3 Неоднозначности

Присваивание или инициализация объекта класса `X` является законным, если присваиваемое значение имеет тип `X`, или если существует единственное преобразование его в значение типа `X`.

В некоторых случаях значение нужного типа строится с помощью повторных применений конструкторов или операций преобразования.

Это должно задаваться явным образом, допустимо неявное пользовательское преобразование только одного уровня вложенности. В некоторых случаях существует несколько способов построения значения нужного типа, но это является незаконным. Приведем пример:

```
class x { /* ... */ x(int); x(char*); };
class y { /* ... */ y(int); };
class z { /* ... */ z(x); };

x f(x);
y f(y);

z g(z);

void k1()
{
  f(1);      // недопустимо, неоднозначность: f(x(1)) или f(y(1))
  f(x(1));
  f(y(1));
  g("asdf"); // недопустимо, g(z(x("asdf"))) не используется
}
```

Пользовательские преобразования типа рассматриваются только в том случае, когда без них нельзя однозначно выбрать вызываемую функцию:

```
class x { /* ... */ x(int); };

void h(double);
void h(x);

void k2()
{
  h(1);
}
```

Вызов `h(1)` можно интерпретировать либо как `h(double(1))`, либо

как `h(x(1))`, поэтому в силу требования однозначности его можно счесть незаконным. Но поскольку в первой интерпретации используется только стандартное преобразование, то по правилам, указанным в § 4.6.6 и § R.13.2, выбирается оно.

Правила на преобразования типа не слишком просто сформулировать и реализовать, не обладают они и достаточной общностью. Рассмотрим требование единственности законного преобразования. Проще всего разрешить транслятору применять любое преобразование, которое он сумеет найти. Тогда для выяснения корректности выражения не нужно рассматривать все существующие преобразования. К сожалению, в таком случае поведение программы будет зависеть от того, какое именно преобразование найдено. В результате поведение программы будет зависеть от порядка описаний преобразований. Поскольку часто эти описания разбросаны по разным исходным файлам (созданным, возможно, разными программистами), то результат программы будет зависеть в каком порядке эти файлы сливаются в программу. С другой стороны, можно вообще запретить неявные преобразования, и это самое простое решение. Но результатом будет некачественный интерфейс, определяемый пользователем, или взрывной рост перегруженных функций и операций, что мы и видели на примере класса complex из предыдущего раздела.

При самом общем подходе учитываются все сведения о типах и рассматриваются все существующие преобразования.

Например, с учетом приведенных описаний в присваивании `aa=f(1)` можно разобраться с вызовом `f(1)`, поскольку тип `aa` задает единственное преобразование. Если аа имеет тип `x`, то единственным преобразованием будет `f(x(1))`, поскольку только оно дает нужный для левой части тип `x`. Если `aa` имеет тип `y`, будет использоваться `f(y(1))`. При самом общем подходе удается разобраться и с вызовом `g("asdf")`, поскольку `g(z(x("asdf")))` является его единственной интерпретацией. Трудность этого подхода в том, что требуется доскональный разбор всего выражения, чтобы установить интерпретацию каждой операции и вызова функции. В результате трансляция замедляется, вычисление выражения может произойти странным образом и появляются загадочные сообщения об ошибках,

когда транслятор учитывает определенные в библиотеках преобразования и т.д.

В результате транслятору приходится учитывать больше информации, чем известно самому программисту! Выбран подход, при котором проверка является строго восходящим процессом, когда в каждый момент рассматривается только одна операция с операндами, типы которых уже прошли проверку.

Требование строго восходящего разбора выражения предполагает, что тип возвращаемого значения не учитывается при разрешении перегрузки:

```
class quad {
  // ...
public:
  quad(double);
  // ...
};

quad operator+(quad,quad);

void f(double a1, double a2)
{
  quad r1 = a1+a2;       // сложение с двойной точностью
  quad r2 = quad(a1)+a2; // вынуждает использовать
                         // операции с типами quad
}
```

В проектировании языка делался расчет на строго восходящий разбор, поскольку он более понятный, а кроме того, не дело транслятора решать такие вопросы, какую точность для сложения желает программист.

Однако, надо отметить, что если определились типы обеих частей в присваивании и инициализации, то для их разрешения используется они оба:

```
class real {
  // ...
public:
```

```
    operator double();
    operator int();
    // ...
};

void g(real a)
{
  double d = a;  // d = a.double();
  int i = a;     // i = a.int();

  d = a;         // d = a.double();
  i = a;         // i = a.int();
}
```

В этом примере выражения все равно разбираются строго восходящим методом, когда в каждый момент рассматриваются только одна операция и типы ее операндов.

7.4 Литералы

Для классов нельзя определить литеральные значения, подобному тому как 1.2 и 12e3 являются литералами типа `double`. Однако, для интерпретации значений классов могут использоваться вместо функций-членов литералы основных типов. Общим средством для построения таких значений служат конструкторы с единственным параметром. Если конструктор достаточно простой и реализуется подстановкой, вполне разумно представлять его вызов как литерал. Например, с учетом описания класса complex в <complex.h> в выражении `zz1*3+zz2*complex(1,2)` произойдет два вызова функций, а не пять. Две операции * приведут к вызову функции, а операция + и вызовы конструктора для построения `complex(3)` и `complex(1,2)` будут реализованы подстановкой.

7.5 Большие объекты

При выполнении любой бинарной операции для типа `complex` реализующие эту операцию функции будут передаваться как параметры копии обоих операндов. Дополнительные расходы, вызванные

копированием двух значений типа `double`, заметны, хотя по всей видимости допустимы.

К сожалению представление не всех классов является столь удобно компактным. Чтобы избежать избыточного копирования, можно определять функции с параметрами типа ссылки:

```
class matrix {
  double m[4][4];
public:
  matrix();
  friend matrix operator+(const matrix&, const matrix&);
  friend matrix operator*(const matrix&, const matrix&);
};
```

Ссылки позволяют без излишнего копирования использовать выражения с обычными арифметическими операциями и для больших объектов. Указатели для этой цели использовать нельзя, т.к. невозможно переопределить интерпретацию операции, если она применяется к указателю. Операцию плюс для матриц можно определить так:

```
matrix operator+(const matrix& arg1, const matrix& arg2)
{
  matrix sum;
  for (int i = 0; i<4; i++)
     for (int j=0; j<4; j++)
        sum.m[i] [j] = arg1.m[i][j] + arg2.m[i][j];
  return sum;
}
```

Здесь в функции `operator+()` операнды выбираются по ссылке, а возвращается само значение объекта. Более эффективным решением был бы возврат тоже ссылки:

```
class matrix {
  // ...
  friend matrix& operator+(const matrix&, const matrix&);
  friend matrix& operator*(const matrix&, const matrix&);
};
```

Это допустимо, но возникает проблема с выделением памяти. Поскольку ссылка на результат операции будет передаваться как ссылка на возвращаемое функцией значение, оно не может быть автоматической переменной этой функции. Поскольку операция может использоваться неоднократно в одном выражении, результат не может быть и локальной статической переменной. Как правило, результат будет записываться в отведенный в свободной памяти объект. Обычно бывает дешевле (по затратам на время выполнения и память данных и команд) копировать результирующее значение, чем размещать его в свободной памяти и затем в конечном счете освобождать выделенную память. К тому же этот способ проще запрограммировать.

7.6 Присваивание и инициализация

Рассмотрим простой строковый класс `string`:

```
struct string {
    char* p;
    int size;  // размер вектора, на который указывает p

    string(int sz) { p = new char[size=sz]; }
    ~string() { delete p; }
};
```

Строка - это структура данных, содержащая указатель на вектор символов и размер этого вектора. Вектор создается конструктором и удаляется деструктором. Но как мы видели в § 5.5.1 здесь могут возникнуть проблемы:

```
void f()
{
  string s1(10);
  string s2(20);
  s1 = s2;
}
```

Здесь будут размещены два символьных вектора, но в результате присваивания `s1 = s2` указатель на один из них будет уничтожен, и

заменится копией второго. По выходе из `f()` будет вызван для `s1` и `s2` деструктор, который дважды удалит один и тот же вектор, результаты чего по всей видимости будут плачевны. Для решения этой проблемы нужно определить соответствующее присваивание объектов типа string:

```
struct string {
  char* p;
  int size;   // размер вектора, на который указывает p

  string(int size) { p = new char[size=sz]; }
  ~string() { delete p; }
  string& operator=(const string&);
};

string& string::operator=(const string& a)
{
  if (this !=&a) {   // опасно, когда s=s
    delete p;
    p = new char[size=a.size];
    strcpy(p,a.p);
  }
  return *this;
}
```

При таком определении `string` предыдущий пример пройдет как задумано. Но после небольшого изменения в `f()` проблема возникает снова, но в ином обличии:

```
void f()
{
  string s1(10);
  string s2 = s1;  // инициализация, а не присваивание
}
```

Теперь только один объект типа `string` строится конструктором `string::string(int)`, а уничтожаться будет две строки. Дело в том, что пользовательская операция присваивания не применяется к неинициализированному объекту. Достаточно взглянуть на функцию `string::operator()`, чтобы понять причину этого: указатель p

будет тогда иметь неопределенное, по сути случайное значение.

Как правило, в операции присваивания предполагается, что ее параметры проинициализированы. Для инициализации типа той, что приведена в этом примере, это не так по определению. Следовательно, чтобы справиться с инициализацией нужна похожая, но своя функция:

```
struct string {
  char* p;
  int size;   // размер вектора, на который указывает p

  string(int size) { p = new char[size=sz]; }
  ~string() { delete p; }
  string& operator=(const string&);
  string(const string&);
};

string:string(const string& a)
{
    p=new char[size=sz];
    strcpy(p,a.p);
}
```

Инициализация объекта типа `X` происходит с помощью конструктора `X(const X&)`. Мы не перестаем повторять, что присваивание и инициализация являются разными операциями. Особенно это важно в тех случаях, когда определен деструктор. Если в классе `X` есть нетривиальный деструктор, например, производящий освобождение объекта в свободной памяти, вероятнее всего, в этом классе потребуется полный набор функций, чтобы избежать копирования объектов по членам:

```
class X {
  // ...
  X(something);       // конструктор, создающий объект
  X(const X&);        // конструктор копирования
  operator=(const X&); // присваивание:
                       // удаление и копирование
  ~X();               // деструктор, удаляющий объект
```

```
};
```

Есть еще два случая, когда приходится копировать объект: передача параметра функции и возврат ею значения. При передаче параметра неинициализированная переменная, т.е. формальный параметр инициализируется. Семантика этой операции идентична другим видам инициализации. То же происходит и при возврате функцией значения, хотя этот случай не такой очевидный. В обоих случаях используется конструктор копирования:

```
string g(string arg)
{
  return arg;
}

main()
{
  string s = "asdf";
  s = g(s);
}
```

Очевидно, после вызова `g()` значение `s` должно быть "asdf". Не трудно записать в параметр s копию значения `s`, для этого надо вызвать конструктор копирования для `string`. Для получения еще одной копии значения s по выходе из `g()` нужен еще один вызов конструктора `string(const string&)`. На этот раз инициализируется временная переменная, которая затем присваивается `s`. Для оптимизации одну, но не обе, из подобных операций копирования можно убрать. Естественно, временные переменные, используемые для таких целей, уничтожаются надлежащим образом деструктором `string::~string()` (см. § R.12.2).

Если в классе `X` операция присваивания `X::operator=(const X&)` и конструктор копирования `X::X(const X&)` явно не заданы программистом, недостающие операции будут созданы транслятором. Эти созданные функции будут копировать по членам для всех членов класса `X`. Если члены принимают простые значения, как в случае комплексных чисел, это то, что нужно, и созданные функции превратятся в простое и оптимальное поразрядное копирование. Если

для самих членов определены пользовательские операции копирования, они и будут вызываться соответствующим образом:

```
class Record {
  string name, address, profession;
  // ...
};

void f(Record& r1)
{
  Record r2 = r1;
}
```

Здесь для копирования каждого члена типа `string` из объекта `r1` будет вызываться `string::operator=(const string&)`. В нашем первом и неполноценном варианте строковый класс имеет член-указатель и деструктор. Поэтому стандартное копирование по членам для него почти наверняка неверно. Транслятор может предупреждать о таких ситуациях.

7.7 Индексация

Операторная функция `operator[]` задает для объектов классов интерпретацию индексации. Второй параметр этой функций (индекс) может иметь произвольный тип. Это позволяет, например, определять ассоциативные массивы. В качестве примера можно переписать определение из § 2.3.10, где ассоциативный массив использовался в небольшой программе, подсчитывающей число вхождений слов в файле.

Там для этого использовалась функция. Мы определим настоящий тип ассоциативного массива:

```
class assoc {
  struct pair {
    char* name;
    int val;
  };
```

```
    pair* vec;
    int max;
    int free;

    assoc(const assoc&);            // предотвращает копирование
    assoc& operator=(const assoc&); // предотвращает копирование
  public:
    assoc(int);
    int& operator[](const char*);
    void print_all();
  };
```

В объекте `assoc` хранится вектор из структур `pair` размером `max`.

В переменной `free` хранится индекс первого свободного элемента вектора.

Чтобы предотвратить копирование объектов assoc, конструктор копирования и операция присваивания описаны как частные. Конструктор выглядит так:

```
assoc::assoc(int s)
{
  max = (s<16) ? 16 : s;
  free = 0;
  vec = new pair[max];
}
```

В реализации используется все тот же неэффективный алгоритм поиска, что и в § 2.3.10. Но теперь, если вектор переполняется, объект `assoc` увеличивается:

```
#include <string.h>

int& assoc::operator[](const char* p)
/*
  работает с множеством пар (структур pair):
  проводит поиск p, возвращает ссылку на
```

```
  целое значение из найденной пары,
  создает новую пару, если p не найдено
*/
{
  register pair* pp;

  for (pp=&vec[free-1]; vec<=pp; pp-- )
     if (strcmp(p,pp->name) == 0) return pp->val;
  if (free == max) { //переполнение: вектор увеличивается
    pair* nvec = new pair[max*2];
    for (int i=0; i<max; i++) nvec[i] = vec[i];
    delete vec;
    vec = nvec;
    max = 2*max;
  }

  pp = &vec[free++];
  pp->name = new char[strlen(p)+1];
  strcpy(pp->name,p);
  pp->val = 0;    // начальное значение = 0
  return pp->val;
}
```

Поскольку представление объекта `assoc` скрыто от пользователя, нужно иметь возможность напечатать его каким-то образом. В следующем разделе будет показано как определить настоящий итератор для такого объекта.

Здесь же мы ограничимся простой функцией печати:

```
void assoc::print_all()
{
  for (int i = 0; i<free; i++)
     cout << vec[i].name << ": " << vec[i].val << '\n';
}
```

Наконец, можно написать тривиальную программу:

```
main() // подсчет числа вхождений во входной
               // поток каждого слова
```

```
{
 const MAX = 256;  // больше длины самого длинного слова
 char buf[MAX];
 assoc vec(512);
 while (cin>>buf) vec[buf]++;
 vec.print_all();
}
```

Опытные программисты могут заметить, что второй комментарий можно легко опровергнуть. Решить возникающую здесь проблему предлагается в упражнении § 7.14 [20]. Дальнейшее развитие понятие ассоциативного массива получит в § 8.8.

Функция `operator[]()` должна быть членом класса. Отсюда следует, что эквивалентность `x[y] == y[x]` может не выполняться, если `x` объект класса. Обычные отношения эквивалентности, справедливые для операций со встроенными типами, могут не выполняться для пользовательских типов (§ 7.2.2, см. также § 7.9).

7.8 Вызов функции

Вызов функции, т.е. конструкцию выражение(список-выражений), можно рассматривать как бинарную операцию, в которой выражение является левым операндом, а список-выражений - правым. Операцию вызова можно перегружать как и другие операции. В функции `operator()()` список фактических параметров вычисляется и проверяется по типам согласно обычным правилам передачи параметров. Перегрузка операции вызова имеет смысл прежде всего для типов, с которыми возможна только одна операция, а также для тех типов, одна из операций над которыми имеет настолько важное значение, что все остальные в большинстве случаев можно не учитывать.

Мы не дали определения итератора для ассоциативного массива типа `assoc`. Для этой цели можно определить специальный класс `assoc_iterator`, задача которого выдавать элементы из `assoc` в некотором порядке. В итераторе необходимо иметь доступ к данным, хранимым в `assoc`, поэтому он должен быть описан как `friend`:

```
class assoc {
friend class assoc_iterator;
    pair* vec;
    int max;
    int free;
public:
    assoc(int);
    int& operator[](const char*);
};
```

Итератор можно определить так:

```
class assoc_iterator {
  const assoc* cs;   // массив assoc
  int i;             // текущий индекс
public:
  assoc_iterator(const assoc& s) { cs = &s; i = 0; }
  pair* operator()()
    { return (i<cs->free)? &cs->vec[i++] : 0; }
};
```

Массив `assoc` объекта `assoc_iterator` нужно инициализировать, и при каждом обращении к нему с помощью операторной функции `()` будет возвращаться указатель на новую пару (структура `pair`) из этого массива. При достижении конца массива возвращается 0:

```
main()  // подсчет числа вхождений во входной
        // поток каждого слова
{
 const MAX = 256;  // больше длины самого длинного слова
 char buf[MAX];
 assoc vec(512);
 while (cin>>buf) vec[buf]++;
 assoc_iterator next(vec);
 pair* p;
 while ( p = next(vec) )
    cout << p->name << ": " << p->val << '\n';
}
```

Итератор подобного вида имеет преимущество перед набором функций, решающим ту же задачу: итератор может иметь собственные частные данные, в которых можно хранить информацию о ходе итерации. Обычно важно и то, что можно одновременно запустить сразу несколько итераторов одного типа.

Конечно, использование объектов для представления итераторов непосредственно никак не связано с перегрузкой операций. Одни предпочитают использовать тип итератора с такими операциями, как `first()`, `next()` и `last()`, другим больше нравится перегрузка операции `++`, которая позволяет получить итератор, используемый как указатель (см. § 8.8). Кроме того, операторная функция `operator()` активно используется для выделения подстрок и индексации многомерных массивов.

Функция `operator()` должна быть функцией-членом.

7.9 Косвенное обращение

Операцию косвенного обращения к члену -> можно определить как унарную постфиксную операцию. Это значит, если есть класс

```
class Ptr {
  // ...
  X* operator->();
};
```

объекты класса `Ptr` могут использоваться для доступа к членам класса `X` также, как для этой цели используются указатели:

```
void f(Ptr p)
{
  p->m = 7;  // (p.operator->())->m = 7
}
```

Превращение объекта `p` в указатель `p.operator->()` никак не зависит от члена m, на который он указывает. Именно по этой причине `operator->()` является унарной постфиксной операцией. Однако, мы не вводим новых синтаксических обозначений, так что имя члена

по-прежнему должно идти после -> :

```
void g(Ptr p)
 {
 X* q1 = p->;  // синтаксическая ошибка
 X* q2 = p.operator->(); // нормально
                }
```

Перегрузка операции -> прежде всего используется для создания "хитрых указателей", т.е. объектов, которые, помимо использования как указатели, позволяют проводить некоторые операции при каждом обращении к указуемому объекту с их помощью. Например, можно определить класс `RecPtr` для организации доступа к объектам класса `Rec`, хранимым на диске. Параметром конструктора `RecPtr` является имя, которое будет использоваться для поиска объекта на диске. При обращении к объекту с помощью функции `RecPtr::operator->()` он переписывается в основную память, а в конце работы деструктор `RecPtr` записывает измененный объект обратно на диск.

```
class RecPtr {
    Rec* in_core_address;
    const char* identifier;
    // ...
 public:
    RecPtr(const char* p)
    : identifier(p) { in_core_address = 0; }
    ~RecPtr()
      { write_to_disc(in_core_address,identifier); }
    Rec* operator->();
 };

 Rec* RecPtr::operator->()
 {
  if (in_core_address == 0)
      in_core_address = read_from_disc(identifier);
  return in_core_address;
 }
```

Использовать это можно так:

```
main(int argc, const char* argv)
{
  for (int i = argc; i; i--) {
    RecPtr p(argv[i]);
    p->update();
  }
}
```

На самом деле, тип `RecPtr` должен определяться как шаблон типа (см. § 8), а тип структуры Record будет его параметром. Кроме того, настоящая программа будет содержать обработку ошибок и взаимодействие с диском будет организовано не столь примитивно.

Для обычных указателей операция -> эквивалентна операциям, использующим `*` и `[]`. Так, если описано

```
Y* p;
```

то выполняется соотношение

```
p->m == (*p).m == p[0].m
```

Как всегда, для определенных пользователем операций такие соотношения не гарантируются. Там, где все-таки такая эквивалентность требуется, ее можно обеспечить:

```
class X {
  Y* p;
public:
  Y* operator->() { return p; }
  Y& operator*() { return *p; }
  Y& operator[](int i) { return p[i]; }
};
```

Если в вашем классе определено более одной подобной операции, разумно будет обеспечить эквивалентность, точно так же, как разумно предусмотреть для простой переменной x некоторого класса, в котором есть операции `++`, `+=` `=` и `+`, чтобы операции `++x` и `x+=1` были эквивалентны `x=x+1`.

Перегрузка -> как и перегрузка [] может играть важную роль для целого класса настоящих программ, а не является просто экспериментом ради любопытства. Дело в том, что в программировании понятие косвенности является ключевым, а перегрузка -> дает ясный, прямой и эффективный способ представления этого понятия в программе. Есть другая точка зрения на операцию ->, как на средство задать в C++ ограниченный, но полезный вариант понятия делегирования (см. § 12.2.8 и 13.9).

7.10 Инкремент и декремент

Если мы додумались до "хитрых указателей", то логично попробовать переопределить операции инкремента ++ и декремента --, чтобы получить для классов те возможности, которые эти операции дают для встроенных типов. Такая задача особенно естественна и необходима, если ставится цель заменить тип обычных указателей на тип "хитрых указателей", для которого семантика остается прежней, но появляются некоторые действия динамического контроля. Пусть есть программа с распространенной ошибкой:

```
void f1(T a)   // традиционное использование
{
  T v[200];
  T* p = &v[0];
  p--;
  *p = a;   // Приехали: 'p' настроен вне массива,
            // и это не обнаружено
  ++p;
  *p = a;   // нормально
}
```

Естественно желание заменить указатель p на объект класса CheckedPtrToT, по которому косвенное обращение возможно только при условии, что он действительно указывает на объект. Применять инкремент и декремент к такому указателю будет можно только в том случае, что указатель настроен на объект в границах массива и в результате этих операций получится объект в границах того же массива:

```
class CheckedPtrToT {
  // ...
};

void f2(T a)  // вариант с контролем
{
  T v[200];
  CheckedPtrToT p(&v[0],v,200);
  p--;
  *p = a;  // динамическая ошибка:
        // `p' вышел за границы массива
  ++p;
  *p = a;  // нормально
}
```

Инкремент и декремент являются единственными операциями в C++, которые можно использовать как постфиксные и префиксные операции. Следовательно, в определении класса `CheckedPtrToT` мы должны предусмотреть отдельные функции для префиксных и постфиксных операций инкремента и декремента:

```
class CheckedPtrToT {
  T* p;
  T* array;
  int size;
public:
      // начальное значение `p'
      // связываем с массивом `a' размера `s'
    CheckedPtrToT(T* p, T* a, int s);
      // начальное значение `p'
      // связываем с одиночным объектом
    CheckedPtrToT(T* p);

    T* operator++();     // префиксная
    T* operator++(int);  // постфиксная

    T* operator--();     // префиксная
    T* operator--(int);  // постфиксная
```

```
    T& operator*();     // префиксная
};
```

Параметр типа `int` служит указанием, что функция будет вызываться для постфиксной операции. На самом деле этот параметр является искусственным и никогда не используется, а служит только для различия постфиксной и префиксной операции. Чтобы запомнить, какая версия функции `operator++` используется как префиксная операция, достаточно помнить, что префиксной является версия без искусственного параметра, что верно и для всех других унарных арифметических и логических операций. Искусственный параметр используется только для "особых" постфиксных операций `++` и `--`.

С помощью класса `CheckedPtrToT` пример можно записать так:

```
void f3(T a)  // вариант с контролем
{
  T v[200];
  CheckedPtrToT p(&v[0],v,200);
  p.operator--(1);
  p.operator*() = a; // динамическая ошибка:
                     // `p' вышел за границы массива
  p.operator++();
  p.operator*() = a; // нормально
}
```

В упражнении § 7.14 [19] предлагается завершить определение класса `CheckedPtrToT`, а другим упражнением (§ 9.10[2]) является преобразование его в шаблон типа, в котором для сообщений о динамических ошибках используются особые ситуации. Примеры использования операций `++` и `--` для итераций можно найти в § 8.8.

7.11 Строковый класс

Теперь можно привести более осмысленный вариант класса `string`. В нем подсчитывается число ссылок на строку, чтобы минимизировать копирование, и используются как константы стандартные строки C++.

```
#include <iostream.h>
#include <string.h>

class string {
  struct srep {
    char* s;       // указатель на строку
    int n;         // счетчик числа ссылок
    srep() { n = 1; }
  };
  srep *p;

public:
  string(const char *);   // string x = "abc"
  string();               // string x;
  string(const string &); // string x = string ...
  string& operator=(const char *);
  string& operator=(const string &);
  ~string();
  char& operator[](int i);

  friend ostream& operator<<(ostream&, const string&);
  friend istream& operator>>(istream&, string&);

  friend int operator==(const string &x, const char *s)
    { return strcmp(x.p->s,s) == 0; }

  friend int operator==(const string &x, const string &y)
    { return strcmp(x.p->s,y.p->s) == 0; }

  friend int operator!=(const string &x, const char *s)
    { return strcmp(x.p->s,s) != 0; }

  friend int operator!=(const string &x, const string &y)
    { return strcmp(x.p->s,y.p->s) != 0; }
                    };
```

Конструкторы и деструкторы тривиальны:

```
string::string()
```

```
{
  p = new srep;
  p->s = 0;
}

string::string(const string& x)
{
  x.p->n++;
  p = x.p;
}

string::string(const char* s)
{
  p = new srep;
  p->s = new char[ strlen(s)+1 ];
  strcpy(p->s, s);
}

string::~string()
{
  if (--p->n == 0) {
     delete[]  p->s;
     delete p;
  }
}
```

Как и всегда операции присваивания похожи на конструкторы. В них нужно позаботиться об удалении первого операнда, задающего левую часть присваивания:

```
string& string::operator=(const char* s)
{
  if (p->n > 1) {  // отсоединяемся от старой строки
     p->n--;
     p = new srep;
  }
  else    // освобождаем строку со старым значением
     delete[] p->s;
```

```
  p->s = new char[ strlen(s)+1 ];
  strcpy(p->s, s);
  return *this;
 }

 string& string::operator=(const string& x)
 {
  x.p->n++;  // защита от случая ``st = st''
  if (--p->n == 0) {
    delete[] p->s;
    delete p
  }
  p = x.p;
  return *this;
 }
```

Операция вывода показывает как используется счетчик числа ссылок. Она сопровождает как эхо каждую введенную строку (ввод происходит с помощью операции <<, приведенной ниже):

```
 ostream& operator<<(ostream& s, const string& x)
 {
   return s << x.p->s << " [" << x.p->n << "]\n";
 }
```

Операция ввода происходит с помощью стандартной функции ввода символьной строки (§ 10.3.1):

```
 istream& operator>>(istream& s, string& x)
 {
   char buf[256];
   s >> buf;   // ненадежно: возможно переполнение buf
               // правильное решение см. в 10.3.1
   x = buf;
   cout << "echo: " << x << '\n';
   return s;
  }
```

Операция индексации нужна для доступа к отдельным символам.

Индекс контролируется:

```
void error(const char* p)
  {
    cerr << p << '\n';
    exit(1);
  }

 char& string::operator[](int i)
{
 if (i<0 || strlen(p->s)<i) error("недопустимое значение индекса");
   return p->s[i];
}
```

В основной программе просто даны несколько примеров применения строковых операций. Слова из входного потока читаются в строки, а затем строки печатаются. Это продолжается до тех пор, пока не будет обнаружена строка `done`, или закончатся строки для записи слов, или закончится входной поток. Затем печатаются все строки в обратном порядке и программа завершается.

```
int main()
 {
   string x[100];
   int n;

   cout << " здесь начало \n";

   for ( n = 0; cin>>x[n]; n++) {
      if (n==100) {
        error("слишком много слов");
        return 99;
      }
      string y;
      cout << (y = x[n]);
      if (y == "done") break;

   }
   cout << "теперь мы идем по словам в обратном порядке \n";
```

```
    for (int i=n-1; 0<=i; i--) cout << x[i];
    return 0;
  }
```

7.12 Друзья и члены

В заключении можно обсудить, когда при обращении в закрытую часть пользовательского типа стоит использовать функции-члены, а когда функции-друзья. Некоторые функции, например конструкторы, деструкторы и виртуальные функции (§ R.12), обязаны быть членами, но для других есть возможность выбора. Поскольку, описывая функцию как член, мы не вводим нового глобального имени, при отсутствии других доводов следует использовать функции-члены.

Рассмотрим простой класс X:

```
class X {
  // ...

  X(int);

  int m1();
  int m2() const;

  friend int f1(X&);
  friend int f2(const X&);
  friend int f3(X);
                };
```

Вначале укажем, что члены `X::m1()` и `X::m2()` можно вызывать только для объектов класса `X`. Преобразование `X(int)` не будет применяться к объекту, для которого вызваны `X::m1()` или `X::m2()`:

```
void g()
{
  1.m1();  // ошибка: X(1).m1() не используется
  1.m2();  // ошибка: X(1).m2() не используется
```

```
}
```

Глобальная функция `f1()` имеет то же свойство (§ 4.6.3), поскольку ее параметр - ссылка без спецификации `const`. С функциями `f2()` и `f3()` ситуация иная:

```
void h()
{
  f1(1);  // ошибка: f1(X(1)) не используется
  f2(1);  // нормально: f2(X(1));
  f3(1);  // нормально: f3(X(1));
}
```

Следовательно операция, изменяющая состояние объекта класса, должна быть членом или глобальной функцией с параметром-ссылкой без спецификации `const`. Операции над основными типами, которые требуют в качестве операндов адреса (=, *, ++ и т.д.), для пользовательских типов естественно определять как члены.

Обратно, если требуется неявное преобразование типа для всех операндов некоторой операции, то реализующая ее функция должна быть не членом, а глобальной функцией и иметь параметр типа ссылки со спецификацией `const` или нессылочный параметр. Так обычно обстоит дело с функциями, реализующими операции, которые для основных типов не требуют адресов в качестве операндов (+, -, || и т.д.).

Если операции преобразования типа не определены, то нет неопровержимых доводов в пользу функции-члена перед функцией-другом с параметром-ссылкой и наоборот. Бывает, что программисту просто одна форма записи вызова нравится больше, чем другая. Например, многим для обозначения функции обращения матрицы `m` больше нравится запись `inv(m)`, чем `m.inv()`. Конечно, если функция `inv()` обращает саму матрицу `m`, а не возвращает новую, обратную m, матрицу, то `inv()` должна быть членом.

При всех прочих равных условиях лучше все-таки остановиться на функции-члене. Можно привести такие доводы. Нельзя гарантировать, что когда-нибудь не будет определена операция обращения. Нельзя во всех случаях гарантировать, что будущие изменения не повлекут за

собой изменения в состоянии объекта. Запись вызова функции-члена ясно показывает программисту, что объект может быть изменен, тогда как запись с параметром-ссылкой далеко не столь очевидна. Далее, выражения допустимые в функции-члене могут быть существенно короче эквивалентных выражений в глобальной функции. Глобальная функция должна использовать явно заданные параметры, а в функции-члене можно неявно использовать указатель `this`. Наконец, поскольку имена членов не являются глобальными именами, они обычно оказываются короче, чем имен глобальных функций.

7.13 Предостережения

Как и всякое другое языковое средство, перегрузка операций может использоваться разумно и неразумно. В частности, возможностью придавать новый смысл обычным операциям можно воспользоваться так, что программа будет совершенно непостижимой. Представьте, каково будет читателю, если в своей программе вы переопределили операцию + так, чтобы она обозначала вычитание. Описанный здесь механизм перегрузки будет защищать программиста и пользователя от таких безрассудств. Поэтому программист не может изменить ни смысл операций над основными типами данных, такими, как `int`, ни синтаксис выражений и приоритеты операций для них.

По всей видимости перегрузку операций имеет смысл использовать для подражания традиционному использованию операций. Запись с обычным вызовом функции можно использовать в тех случаях, когда традиционной записи с базовой операцией не существует, или, когда набор операций, которые допускают перегрузку, не достаточен, чтобы записать с его помощью нужные действия.

7.14 Упражнения

1. (*2) Определите итератор для класса `string`. Определите операцию конкатенации + и операцию `+=`, значащую "добавить в конец строки". Какие еще операции вы хотели бы и смогли определить для этого класса?
2. (*1.5) Определите для строкового класса операцию выделения подстроки с помощью перегрузки ().

3. (*3) Определите класс `string` таким образом, чтобы операцию выделения подстроки можно было применять к левой части присваивания. Вначале напишите вариант, в котором строку можно присваивать подстроке той же длины, а затем вариант с различными длинами строк.
4. (*2) Разработайте класс `string` таким образом, чтобы объекты его трактовались при передаче параметров и присваивании как значения, т.е. чтобы в классе string копировались сами представления строк, а не только управляющие структуры.
5. (*3) Измените класс `string` из предыдущего упражнения так, чтобы строки копировались только при необходимости. Это значит, что нужно хранить одно общее представления двух одинаковых строк до тех пор, пока одна из них не изменится. Не пытайтесь задать операцию выделения подстроки, которую одновременно можно применять и к левой части присваивания.
6. (*4) Определите класс `string`, обладающий перечисленными в предыдущих упражнениях свойствами: объекты его трактуются как значения, копирование является отложенным (т.е. происходит только при необходимости) и операцию выделения подстроки можно применять к левой части присваивания.
7. (*2) Какие преобразования типа используются в выражениях следующей программы?

```
struct X {
  int i;
  X(int);
  operator+(int);
};

struct Y {
  int i;
  Y(X);
  operator+(X);
  operator int();
};

extern X operator*(X,Y);
extern int f(X);
```

```
X x = 1;
Y y = x;
int i = 2;

int main()
{
 i + 10;    y + 10;    y + 10 * y;
 x + y + i;  x * X +i;  f(7);
 f(y);      y + y;     106 + y;
}
```

Определите `X` и `Y` как целые типы. Измените программу так, чтобы ее можно было выполнить и она напечатала значения всех правильных выражений.

8. (*2) Определите класс `INT`, который будет эквивалентен типу `int`. Подсказка: определите функцию `INT::operator int()`.
9. (*1) Определите класс `RINT`, который будет эквивалентен типу `int`, за исключением того, что допустимыми будут только операции: + (унарный и бинарный), - (унарный и бинарный), *, / и %. Подсказка: не надо определять `RINT::operator int()`.
10. (*3) Определите класс `LINT`, эквивалентный классу `RINT`, но в нем для представления целого должно использоваться не менее 64 разрядов.
11. (*4) Определите класс, реализующий арифметику с произвольной точностью. Подсказка: Придется использовать память так, как это делается в классе `string`.
12. (*2) Напишите программу, в которой благодаря макрокомандам и перегрузке будет невозможно разобраться. Совет: определите для типа `INT +` как -, и наоборот; с помощью макроопределения задайте `int` как `INT`. Кроме того, большую путаницу можно создать, переопределяя широко известные функции, и используя параметры типа ссылки и задавая вводящие в заблуждение комментарии.
13. (*3) Обменяйтесь решениями упражнения [12] с вашим другом. Попробуйте понять, что делает его программа, не запуская ее. Если вы сделаете это упражнение, вам станет ясно, чего надо избегать.
14. (*2) Перепишите примеры с классами `complex` (§ 7.3), `tiny` (

§ 7.3.2) и `string` (§ 7.11), не используя дружественные функции. Используйте только функции-члены. Проверьте новые версии этих классов. Сравните их с версиями, в которых используются дружественные функции. Обратитесь к упражнению 5.3.

15. (*2) Определите тип `vec4` как вектор из четырех чисел с плавающей точкой. Определите для него функцию `operator[]`. Для комбинаций векторов и чисел с плавающей точкой определите операции: +, -, *, /, =, +=, -=, *= и /=.
16. (*3) Определите класс `mat4` как вектор из четырех элементов типа vec4. Определите для него функцию `operator[]`, возвращающую `vec4`. Определите для этого типа обычные операции с матрицами. Определите в mat4 функцию, производящую преобразование Гаусса с матрицей.
17. (*2) Определите класс `vector`, аналогичный классу `vec4`, но здесь размер вектора должен задаваться как параметр конструктора `vector::vector(int)`.
18. (*3) Определите класс matrix, аналогичный классу `mat4`, но здесь размерности матрицы должны задаваться как параметры конструктора `matrix::matrix(int,int)`.
19. (*3) Завершите определение класса `CheckedPtrToT` из § 7.10 и проверьте его. Чтобы определение этого класса было полным, необходимо определить, по крайней мере, такие операции: *, ->, =, ++ и --. Не выдавайте динамическую ошибку, пока действительно не произойдет обращение по указателю с неопределенным значением.
20. (*1.5) Перепишите пример с программой подсчета слов из § 7.7 так, чтобы в ней не было заранее заданной максимальной длины слова.

Шаблоны типа

В этой лекции вводится понятие шаблона типа. С его помощью можно достаточно просто определить и реализовать без потерь в эффективности выполнения программы и, не отказываясь от статического контроля типов, такие контейнерные классы, как списки и ассоциативные массивы. Кроме того, шаблоны типа позволяют определить сразу для целого семейства типов обобщенные (генерические) функции, например, такие, как sort (сортировка). В качестве примера шаблона типов и его связи с другими конструкциями языка приводится семейство списочных классов. Чтобы показать способы получения программы из в значительной степени независимых частей, приводится несколько вариантов шаблонной функции sort(). В конце определяется простой шаблон типа для ассоциативного массива и показывается на двух небольших демонстрационных программах, как им пользоваться.

8.1 Введение

Вот ваша цитата

Бьерн Страуструп

Одним из самых полезных видов классов является контейнерный класс, т.е. такой класс, который хранит объекты каких-то других типов. Списки, массивы, ассоциативные массивы и множества - все это контейнерные классы. С помощью описанных в лекциях 5 и 7 средств можно определить класс, как контейнер объектов единственного, известного типа. Например, в § 5.3.2 определяется множество целых. Но контейнерные классы обладают тем интересным свойством, что тип содержащихся в них объектов не имеет особого значения для создателя контейнера, но для пользователя конкретного контейнера этот тип является существенным. Следовательно, тип содержащихся объектов должен быть параметром контейнерного класса, и создатель такого класса будет определять его с помощью типа-параметра. Для каждого конкретного контейнера (т.е. объекта контейнерного класса) пользователь будет указывать каким должен быть тип содержащихся в нем объектов. Примером такого контейнерного класса был шаблон типа

`Vector` из § 1.4.3.

В этой лекции исследуется простой шаблон типа `stack` (стек) и в результате вводится понятие шаблонного класса. Затем рассматриваются более полные и правдоподобные примеры нескольких родственных шаблонов типа для списка. Вводятся шаблонные функции и формулируются правила, что может быть параметром таких функций. В конце приводится шаблон типа для ассоциативного массива.

8.2 Простой шаблон типа

Шаблон типа для класса задает способ построения отдельных классов, подобно тому, как описание класса задает способ построения его отдельных объектов. Можно определить стек, содержащий элементы произвольного типа:

```
template<class T>
class stack {
  T* v;
  T* p;
  int sz;

public:
  stack(int s) { v = p = new T[sz=s]; }
  ~stack() { delete[] v; }

  void push(T a) { *p++ = a; }
  T pop() { return *--p; }

  int size() const { return p-v; }
};
```

Для простоты не учитывался контроль динамических ошибок. Не считая этого, пример полный и вполне правдоподобный.

Префикс `template<class T>` указывает, что описывается шаблон типа с параметром `T`, обозначающим тип, и что это обозначение будет использоваться в последующем описании. После того, как

идентификатор `T` указан в префиксе, его можно использовать как любое другое имя типа. Область видимости `T` продолжается до конца описания, начавшегося префиксом `template<class T>`. Отметим, что в префиксе `T` объявляется типом, и оно не обязано быть именем класса. Так, ниже в описании объекта `sc` тип `T` оказывается просто char.

Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном типа), и его можно использовать как все имена класса. Например, ниже определяется объект `sc` класса `stack<char>`:

```
stack<char> sc(100);   // стек символов
```

Если не считать особую форму записи имени, класс `stack<char>` полностью эквивалентен классу определенному так:

```
class stack_char {
  char* v;
  char* p;
  int sz;
public:
  stack_char(int s) { v = p = new char[sz=s]; }
  ~stack_char() { delete[] v; }

  void push(char a) { *p++ = a; }
  char pop() { return *--p; }

  int size() const { return p-v; }
};
```

Можно подумать, что шаблон типа - это хитрое макроопределение, подчиняющееся правилам именования, типов и областей видимости, принятым в C++. Это, конечно, упрощение, но это такое упрощение, которое помогает избежать больших недоразумений. В частности, применение шаблона типа не предполагает каких-либо средств динамической поддержки помимо тех, которые используются для обычных "ручных" классов. Не следует так же думать, что оно приводит к сокращению программы.

Обычно имеет смысл вначале отладить конкретный класс, такой, например, как `stack_char`, прежде, чем строить на его основе шаблон типа `stack<T>`. С другой стороны, для понимания шаблона типа полезно представить себе его действие на конкретном типе, например `int` или `shape*`, прежде, чем пытаться представить его во всей общности.

Имея определение шаблонного класса `stack`, можно следующим образом определять и использовать различные стеки:

```
stack<shape*> ssp(200);  // стек указателей на фигуры
stack<Point> sp(400);    // стек структур Point

void f(stack<complex>& sc)  // параметр типа `ссылка на
                            // complex'
{
  sc.push(complex(1,2));
  complex z = 2.5*sc.pop();

  stack<int>*p = 0;         // указатель на стек целых
  p = new stack<int>(800);  // стек целых размещается
   // в свободной памяти
  for ( int i = 0; i<400; i++) {
     p->push(i);
     sp.push(Point(i,i+400));
  }

  // ...
                        }
```

Поскольку все функции-члены класса `stack` являются подстановками, и в этом примере транслятор создает вызовы функций только для размещения в свободной памяти и освобождения.

Функции в шаблоне типа могут и не быть подстановками, шаблонный класс `stack` с полным правом можно определить и так:

```
template<class T> class stack {
   T* v;
```

```
    T* p;
    int sz;
  public:
    stack(int);
    ~stack();

    void push(T);
    T pop();

    int size() const;
  };
```

В этом случае определение функции-члена `stack` должно быть дано где-то в другом месте, как это и было для функций- членов обычных, нешаблонных классов. Подобные функции так же параметризируются типом, служащим параметром для их шаблонного класса, поэтому определяются они с помощью шаблона типа для функции. Если это происходит вне шаблонного класса, это надо делать явно:

```
template<class T> void stack<T>::push(T a)
{
  *p++ = a;
}

template<class T> stack<T>::stack(int s)
{
  v = p = new T[sz=s];
}
```

Отметим, что в пределах области видимости имени `stack<T>` уточнение `<T>` является избыточным, и `stack<T>::stack` - имя конструктора.

Задача системы программирования, а вовсе не программиста, предоставлять версии шаблонных функций для каждого фактического параметра шаблона типа. Поэтому для приводившегося выше примера система программирования должна создать определения конструкторов для классов `stack<shape*>`, `stack<Point>` и `stack<int>`, деструкторов для `stack<shape*>` и `stack<Point>`, версии

функций `push()` для `stack<complex>`, `stack<int>` и `stack<Point>` и версию функции `pop()` для `stack<complex>`. Такие создаваемые функции будут совершенно обычными функциями-членами, например:

```
void stack<complex>::push(complex a) { *p++ = a; }
```

Здесь отличие от обычной функции-члена только в форме имени класса. Точно так же, как в программе может быть только одно определение функции-члена класса, возможно только одно определение шаблона типа для функции-члена шаблонного класса. Если требуется определение функции-члена шаблонного класса для конкретного типа, то задача системы программирования найти шаблон типа для этой функции-члена и создать нужную версию функции. В общем случае система программирования может рассчитывать на указания от программиста, которые помогут найти нужный шаблон типа.

Важно составлять определение шаблона типа таким образом, чтобы его зависимость от глобальных данных была минимальной. Дело в том, шаблон типа будет использоваться для порождения функций и классов на основе заранее неизвестного типа и в неизвестных контекстах. Практически любая, даже слабая зависимость от контекста может проявиться как проблема при отладке программы пользователем, который, вероятнее всего, не был создателем шаблона типа. К совету избегать, насколько это возможно, использований глобальных имен, следует относиться особенно серьезно при разработке шаблона типа.

8.3 Шаблоны типа для списка

На практике при разработке класса, служащего коллекцией объектов, часто приходится учитывать взаимоотношения использующихся в реализации классов, управление памятью и необходимость определить итератор по содержимому коллекции. Часто бывает так, что несколько родственных классов разрабатываются совместно (§ 12.2). В качестве примера мы предложим семейство классов, представляющих односвязные списки и шаблоны типа для них.

8.3.1 Список с принудительной связью

Вначале определим простой список, в котором предполагается, что в каждом заносимом в список объекте есть поле связи. Потом этот список будет использоваться как строительный материал для создания более общих списков, в которых объект не обязан иметь поле связи. Сперва в описаниях классов будет приведена только общая часть, а реализация будет дана в следующем разделе. Это делается затем, чтобы вопросы проектирования классов не затемнялись деталями их реализации.

Начнем с типа `slink`, определяющего поле связи в односвязном списке:

```
struct slink {
  slink* next;
  slink() { next = 0; }
  slink(slink* p) { next = p; }
};
```

Теперь можно определить класс, который может содержать объекты любого, производного от `slink`, класса:

```
class slist_base {
   // ...
public:
   int insert(slink*); // добавить в начало списка
   int append(slink*); // добавить к концу списка
   slink* get();       // удалить и возвратить начало списка
   // ...
};
```

Такой класс можно назвать списком с принудительной связью, поскольку его можно использовать только в том случае, когда все элементы имеют поле `slink`, которое используется как указатель на `slist_base`. Само имя `slist_base` (базовый односвязный список) говорит, что этот класс будет использоваться как базовый для односвязных списочных классов. Как обычно, при разработке семейства родственных классов возникает вопрос, как выбирать имена для различных членов семейства. Поскольку имена классов не могут перегружаться, как это делается для имен функций, для обуздания размножения имен перегрузка нам не поможет.

Класс `slist_base` можно использовать так:

```
void f()
{
  slist_base slb;
  slb.insert(new slink);
  // ...
  slink* p = slb.get();
  // ...
  delete p;
}
```

Но поскольку структура `slink` не может содержать никакой информации помимо связи, этот пример не слишком интересен. Чтобы воспользоваться `slist_base`, надо определить полезный, производный от `slink`, класс. Например, в трансляторе используются узлы дерева программы name (имя), которые приходится связывать в список:

```
class name : public slink {
   // ...
};

void f(const char* s)
{
  slist_base slb;
  slb.insert(new name(s));
  // ...
  name* p = (name*)slb.get();
  // ...
  delete p;
}
```

Здесь все нормально, но поскольку определение класса `slist_base` дано через структуру `slink`, приходится использовать явное приведение типа для преобразования значения типа `slink*`, возвращаемого функцией `slist_base::get()`, в `name*`. Это некрасиво. Для большой программы, в которой много списков и производных от `slink` классов, это к тому же чревато ошибками. Нам

пригодилась бы надежная по типу версия класса `slist_base`:

```
template<class T>
 class Islist : private slist_base {
 public:
   void insert(T* a) { slist_base::insert(a); }
   T* get() { return (T*) slist_base::get(); }
   // ...
 };
```

Приведение в функции `Islist::get()` совершенно оправдано и надежно, поскольку в классе `Islist` гарантируется, что каждый объект в списке действительно имеет тип `T` или тип производного от `T` класса. Отметим, что `slist_base` является частным базовым классом `Islist`. Мы не хотим, чтобы пользователь случайно натолкнулся на ненадежные детали реализации.

Имя `Islist` (intrusive singly linked list) обозначает односвязный список с принудительной связью. Этот шаблон типа можно использовать так:

```
void f(const char* s)
{
  Islist<name> ilst;
  ilst.insert(new name(s));
  // ...
  name* p = ilst.get();
  // ...
  delete p
}
```

Попытки некорректного использования будет выявлены на стадии трансляции:

```
class expr : public slink {
   // ...
};

void g(expr* e)
{
```

```
    Islist<name> ilst;
    ilst.insert(e);  // ошибка: Islist<name>::insert(),
                     // а нужно name*
    // ...
  }
```

Нужно отметить несколько важных моментов относительно нашего примера. Во-первых, решение надежно в смысле типов (преграда тривиальным ошибкам ставится в очень ограниченной части программы, а именно, в функциях доступа из `Islist`). Во-вторых, надежность типов достигается без увеличения затрат времени и памяти, поскольку функции доступа из `Islist` тривиальны и реализуются подстановкой. В-третьих, поскольку вся настоящая работа со списком делается в реализации класса `slist_base` (пока еще не представленной), никакого дублирования функций не происходит, а исходный текст реализации, т.е. функции `slist_base`, вообще не должен быть доступен пользователю. Это может быть существенно в коммерческом использовании служебных программ для списков. Кроме того, достигается разделение между интерфейсом и его реализацией, и становится возможной смена реализации без перетрансляции программ пользователя. Наконец, простой список с принудительной связью близок по использованию памяти и времени к оптимальному решению. Иными словами, такой подход близок к оптимальному по времени, памяти, упрятыванию данных и контролю типов и в тоже время он обеспечивает большую гибкость и компактность выражений.

К сожалению, объект может попасть в `Islist` только, если он является производным от `slink`. Значит нельзя иметь список `Islist` из значений типа `int`, нельзя составить список из значений какого-то ранее определенного типа, не являющегося производным от `slink`. Кроме того, придется постараться, чтобы включить объект в два списка `Islist` (§ 6.5.1).

8.3.2 Список без принудительной связи

После "экскурса" в вопросы построения и использования списка с принудительной связью перейдем к построению списков без

принудительной связи. Это значит, что элементы списка не обязаны содержать дополнительную информацию, помогающую в реализации списочного класса. Поскольку мы больше не можем рассчитывать, что объект в списке имеет поле связи, такую связь надо предусмотреть в реализации:

```
template<class T>
struct Tlink : public slink {
  T info;
  Tlink(const T& a) : info(a) { }
};
```

Класс `Tlink<T>` хранит копию объектов типа `T` помимо поля связи, которое идет от его базового класса `slink`. Отметим, что используется инициализатор в виде `info(a)`, а не присваивание `info=a`. Это существенно для эффективности операции в случае типов, имеющих нетривиальные конструкторы копирования и операции присваивания (§ 7.11). Для таких типов (например, для `String`) определив конструктор как

```
Tlink(const T& a) { info = a; }
```

мы получим, что будет строиться стандартный объект `String`, а уже затем ему будет присваиваться значение.

Имея класс, определяющий связь, и класс `Islist`, получить определение списка без принудительной связи совсем просто:

```
template<class T>
 class Slist : private slist_base {
 public:
   void insert(const T& a)
     { slist_base::insert(new Tlink<T>(a)); }
   void append(const T& a)
     { slist_base::append(new Tlink<T>(a)); }
   T get();
   // ...
};
```

```
template<class T>
T Slist<T>::get()
{
  Tlink<T>* lnk = (Tlink<T>*) slist_base::get();
  T i = lnk->info;
  delete lnk;
  return i;
}
```

Работать со списком `Slist` так же просто, как и со списком `Islist`. Различие в том, что можно включать в `Slist` объект, класс которого не является производным от `slink`, а также можно включать один объект в два списка:

```
void f(int i)
{
  Slist<int> lst1;
  Slist<int> lst2;

  lst1.insert(i);
  lst2.insert(i);
  // ...

  int i1 = lst1.get();
  int i2 = lst2.get();
  // ...
}
```

Однако, список с принудительной связью, например `Islist`, позволял создавать существенно более эффективную программу и давал более компактное представление. Действительно, при каждом включении объекта в список `Slist` нужно разместить объект `Tlink`, а при каждом удалении объекта из `Slist` нужно удалить объект `Tlink`, причем каждый раз копируется объект типа `T`. Когда возникает такая проблема дополнительных расходов, могут помочь два приема. Во-первых, `Tlink` является прямым кандидатом для размещения с помощью практически оптимальной функции размещения специального назначения (см. § 5.5.6). Тогда дополнительные расходы при выполнении программы сократятся до обычно приемлемого уровня.

Во-вторых, полезным оказывается такой прием, когда объекты хранятся в "первичном" списке, имеющем принудительную связь, а списки без принудительной связи используются только, когда требуется включение объекта в несколько списков:

```
void f(name* p)
{
  Islist<name> lst1;
  Slist<name*> lst2;

  lst1.insert(p);   // связь через объект `*p'
  lst2.insert(p);   // для хранения `p' используется
                    // отдельный объект типа список
  // ...
}
```

Конечно, подобные трюки можно делать только в отдельном компоненте программы, чтобы не допустить путаницы списочных типов в интерфейсах различных компонент. Но это именно тот случай, когда ради эффективности и компактности программы на них стоит идти.

Поскольку конструктор `Slist` копирует параметр для `insert()`, список `Slist` пригоден только для таких небольших объектов, как целые, комплексные числа или указатели. Если для объектов копирование слишком накладно или неприемлемо по смысловым причинам, обычно выход бывает в том, чтобы вместо объектов помещать в список указатели на них. Это сделано в приведенной выше функции `f()` для `lst2`.

Отметим, что раз параметр для `Slist::insert()` копируется, передача объекта производного класса функции `insert()`, ожидающей объект базового класса, не пройдет гладко, как можно было (по наивности) подумать:

```
class smiley : public circle { /* ... */ };

void g1(Slist<circle>& olist, const smiley& grin)
{
```

```
  olist.insert(grin);  // ловушка!
}
```

В список будет включена только часть `circle` объекта типа *`smiley`*. Отметим, что эта неприятность будет обнаружена транслятором в том случае, который можно считать наиболее вероятным. Так, если бы рассматриваемый базовый класс был абстрактным, транслятор запретил бы "урезание" объекта производного класса:

```
void g2(Slist<shape>& olist, const circle& c)
{
  olist.insert(c);  // ошибка: попытка создать объект
                    // абстрактного класса
}
```

Чтобы избежать "урезания" объекта нужно использовать указатели:

```
void g3(Slist<shape*>& plist, const smiley& grin)
{
  plist.insert(&grin);  // прекрасно
}
```

Не нужно использовать параметр-ссылку для шаблонного класса:

```
void g4(Slist<shape&>& rlist, const smiley& grin)
{
  rlist.insert(grin); // ошибка: будут созданы команды,
                      // содержащие ссылку на ссылку (shape&&)
}
```

При генерации по шаблону типа ссылки, используемые подобным образом, приведут к ошибкам в типах. Генерация по шаблону типа для функции

```
Slist::insert(T&);
```

приведет к появлению недопустимой функции

```
Slist::insert(shape&&);
```

Ссылка не является объектом, поэтому нельзя иметь ссылку на ссылку.

Поскольку список указателей является полезной конструкцией, имеет смысл дать ему специальное имя:

```
template<class T>
 class Splist : private Slist<void*> {
 public:
  void insert(T* p) { Slist<void*>::insert(p); }
  void append(T* p) { Slist<void*>::append(p); }
  T* get() { return (T*) Slist<void*>::get(); }
};

class Isplist : private slist_base {
public:
  void insert(T* p) { slist_base::insert(p); }
  void append(T* p) { slist_base::append(p); }
  T* get() { return (T*) slist_base::get(); }
};
```

Эти определения к тому же улучшают контроль типов и еще больше сокращают необходимость дублировать функции.

Часто бывает полезно, чтобы тип элемента, указываемый в шаблоне типа, сам был шаблонным классом. Например, разреженную матрицу, содержащую даты, можно определить так:

```
typedef Slist< Slist<date> > dates;
```

Обратите внимание на наличие пробелов в этом определении. Если между первой и второй угловой скобкой > нет пробелов, возникнет синтаксическая ошибка, поскольку >> в определении

```
typedef Slist<Slist<date>> dates;
```

будет трактоваться как операция сдвига вправо. Как обычно, вводимое в `typedef` имя служит синонимом обозначаемого им типа, а не является новым типом. Конструкция `typedef` полезна для именования для длинных имен шаблонных классов также, как она полезна для любых других длинных имен типов.

Отметим, что параметр шаблона типа, который может по-разному использоваться в его определении, должен все равно указываться среди списка параметров шаблона один раз. Поэтому шаблон типа, в котором используется объект `T` и список элементов `T`, надо определять так:

```
template<class T> class mytemplate {
  T ob;
  Slist<T> slst;
  // ...
};
```

а вовсе не так:

```
template<class T, class Slist<t> > class mytemplate {
  T obj;
  Slist<T> slst;
  // ...
};
```

В § 8.6 и § R.14.2 даны правила, что может быть параметром шаблона типа.

8.3.3 Реализация списка

Реализация функций `slist_base` очевидна. Единственная трудность связана с обработкой ошибок. Например, что делать если пользователь с помощью функции `get()` пытается взять элемент из пустого списка. Подобные ситуации разбираются в функции обработки ошибок `slist_handler()`. Более развитый метод, рассчитанный на особые ситуации, будет обсуждаться в лекции 9.

Приведем полное описание класса `slist_base`:

```
class slist_base {
   slink* last;  // last->next является началом списка
 public:
   void insert(slink* a);  // добавить в начало списка
   void append(slink* a);  // добавить в конец списка
```

```
    slink* get();          // удалить и возвратить
                           // начало списка
    void clear() { last = 0; }

    slist_base() { last = 0; }
    slist_base(slink* a) { last = a->next = a; }

    friend class slist_base_iter;
  };
```

Чтобы упростить реализацию обеих функций `insert` и `append`, хранится указатель на последний элемент замкнутого списка:

```
void slist_base::insert(slink* a) // добавить в начало списка
{
  if (last)
    a->next = last->next;
  else
    last = a;
  last->next = a;
}
```

Заметьте, что `last->next` - первый элемент списка.

```
void slist_base::append(slink* a) // добавить в конец списка
{
  if (last) {
    a->next = last->next;
    last = last->next = a;
  }
  else
    last = a->next = a;
}

                    slink* slist_base::get() // удалить и возвратить начало списк
                    {
 if (last == 0)
   slist_handler("нельзя взять из пустого списка");
 slink* f = last->next;
```

```
  if (f== last)
    last = 0;
  else
    last->next = f->next;
  return f;
                    }
```

Возможно более гибкое решение, когда `slist_handler` - указатель на функцию, а не сама функция. Тогда вызов

```
slist_handler("нельзя взять из пустого списка");
```

будет задаваться так

```
(*slist_handler)(" нельзя взять из пустого списка");
```

Как мы уже делали для функции `new_handler` (§ 3.2.6), полезно завести функцию, которая поможет пользователю создавать свои обработчики ошибок:

```
typedef void (*PFV)(const char*);

PFV set_slist_handler(PFV a)
{
 PFV old = slist_handler;
 slist_handler = a;
 return old;
}

PFV slist_handler = &default_slist_handler;
```

Особые ситуации, которые обсуждаются в лекции 9, не только дают альтернативный способ обработки ошибок, но и способ реализации `slist_handler`.

8.3.4 Итерация

В классе `slist_base` нет функций для просмотра списка, можно

только вставлять и удалять элементы. Однако, в нем описывается как друг класс `slist_base_iter`, поэтому можно определить подходящий для списка итератор. Вот один из возможных, заданный в том стиле, какой был показан в § 7.8:

```
class slist_base_iter {
  slink* ce;      // текущий элемент
  slist_base* cs; // текущий список
public:
  inline slist_base_iter(slist_base& s);
  inline slink* operator()()
};

slist_base_iter::slist_base_iter(slist_base& s)
{
  cs = &s;
  ce = cs->last;
}

slink* slist_base_iter::operator()()
  // возвращает 0, когда итерация кончается
{
  slink* ret = ce ? (ce=ce->next) : 0;
  if (ce == cs->last) ce = 0;
  return ret;
}
```

Исходя из этих определений, легко получить итераторы для `Slist` и `Islist`. Сначала надо определить дружественные классы для итераторов по соответствующим контейнерным классам:

```
template<class T> class Islist_iter;

template<class T> class Islist {
  friend class Islist_iter<T>;
  // ...
};

template<class T> class Slist_iter;
```

```
template<class T> class Slist {
  friend class Slist_iter<T>;
  // ...
};
```

Обратите внимание, что имена итераторов появляются без определения их шаблонного класса. Это способ определения в условиях взаимной зависимости шаблонов типа.

Теперь можно определить сами итераторы:

```
template<class T>
 class Islist_iter : private slist_base_iter {
 public:
  Islist_iter(Islist<T>& s) : slist_base_iter(s) { }

  T* operator()()
    { return (T*) slist_base_iter::operator()(); }
 };

 template<class T>
 class Slist_iter : private slist_base_iter {
 public:
  Slist_iter(Slist<T>& s) : slist_base_iter(s) { }
  inline T* operator()();
 };

 T* Slist_iter::operator()()
 {
  return ((Tlink<T>*) slist_base_iter::operator()())->info;
 }
```

Заметьте, что мы опять использовали прием, когда из одного базового класса строится семейство производных классов (а именно, шаблонный класс). Мы используем наследование, чтобы выразить общность классов и избежать ненужного дублирования функций. Трудно переоценить стремление избежать дублирования функций при реализации таких простых и часто используемых классов как списки и

итераторы. Пользоваться этими итераторами можно так:

```
void f(name* p)
{
  Islist<name> lst1;
  Slist<name> lst2;

  lst1.insert(p);
  lst2.insert(p);
  // ...

  Islist_iter<name> iter1(lst1);
  const name* p;
  while (p=iter1()) {
    list_iter<name> iter2(lst1);
    const name* q;
    while (q=iter2()) {
      if (p == q) cout << "найден" << *p << '\n';
    }
  }
}
```

Есть несколько способов задать итератор для контейнерного класса. Разработчик программы или библиотеки должен выбрать один из них и придерживаться его. Приведенный способ может показаться слишком хитрым. В более простом варианте можно было просто переименовать `operator()()` как `next()`. В обоих вариантах предполагается взаимосвязь между контейнерным классом и итератором для него, так что можно при выполнении итератора обработать случаи, когда элементы добавляются или удаляются из контейнера. Этот и некоторые другие способы задания итераторов были бы невозможны, если бы итератор зависел от функции пользователя, в которой есть указатели на элементы из контейнера. Как правило, контейнер или его итераторы реализуют понятие "установить итерацию на начало" и понятие "текущего элемента".

Если понятие текущего элемента предоставляет не итератор, а сам контейнер, итерация происходит в принудительном порядке по отношению к контейнеру аналогично тому, как поля связи

принудительно хранятся в объектах из контейнера. Значит трудно одновременно вести две итерации для одного контейнера, но расходы на память и время при такой организации итерации близки к оптимальным. Приведем пример:

```
class slist_base {
  // ...
  slink* last;  // last->next голова списка
  slink* current;  // текущий элемент
public:
  // ...
  slink* head() { return last?last->next:0; }
  slink* current() { return current; }
  void set_current(slink* p) { current = p; }
  slink* first() { set_current(head()); return current; }
  slink* next();
  slink* prev();
};
```

Подобно тому, как в целях эффективности и компактности программы можно использовать для одного объекта как список с принудительной связью, так и список без нее, для одного контейнера можно использовать принудительную и непринудительную итерацию:

```
void f(Islist<name>& ilst)
// медленный поиск имен-дубликатов
{
  list_iter<name> slow(ilst);  // используется итератор
  name* p;
  while (p = slow()) {
   ilst.set_current(p); // рассчитываем на текущий элемент
   name* q;
   while (q = ilst.next())
     if (strcmp(p->string,q->string) == 0)
       cout << "дубликат" << p << '\n';
  }
}
```

Еще один вид итераторов показан в § 8.8.

8.4 Шаблоны типа для функций

Использование шаблонных классов означает наличие шаблонных функций-членов. Помимо этого, можно определить глобальные шаблонные функции, т.е. шаблоны типа для функций, не являющихся членами класса. Шаблон типа для функций порождает семейство функций точно также, как шаблон типа для класса порождает семейство классов. Эту возможность мы обсудим на последовательности примеров, в которых приводятся варианты функции сортировки `sort()`. Каждый из вариантов в последующих разделах будет иллюстрировать общий метод.

Как обычно мы сосредоточимся на организации программы, а не на разработке ее алгоритма, поэтому использоваться будет тривиальный алгоритм. Все варианты шаблона типа для `sort()` нужны для того, чтобы показать возможности языка и полезные приемы программирования. Варианты не упорядочены в соответствии с тем, насколько они хороши. Кроме того, можно обсудить и традиционные варианты без шаблонов типа, в частности, передачу указателя на функцию, производящую сравнение.

8.4.1 Простой шаблон типа для глобальной функции

Начнем с простейшего шаблона для `sort()`:

```
template<class T> void sort(Vector<T>&);

void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
  sort(vi);      // sort(Vector<int>& v);
  sort(vc);      // sort(Vector<String>& v);
  sort(vi2);     // sort(Vector<int>& v);
  sort(vs);      // sort(Vector<char*>& v);
}
```

Какая именно функция `sort()` будет вызываться, определяется фактическим параметром. Программист дает определение шаблона типа для функции, а задача системы программирования обеспечить создание правильных вариантов функции по шаблону и вызов соответствующего варианта. Например, простой шаблон с алгоритмом пузырьковой сортировки можно определить так:

```
template<class T> void sort(Vector<T>& v)
/*
   Сортировка элементов в порядке возрастания
   Используется сортировка по методу пузырька
*/
{
  unsigned n = v.size();

  for (int i=0; i<n-1; i++)
     for (int j=n-1; i<j; j--)
        if (v[j] < v[j-1]) {  // меняем местами v[j] и v[j-1]
           T temp = v[j];
           v[j] = v[j-1];
           v[j-1] = temp;
        }
  }
```

Советуем сравнить это определение с функцией сортировки с тем же алгоритмом из § 4.6.9. Существенное отличие этого варианта в том, что вся необходимая информация передается в единственном параметре `v`. Поскольку тип сортируемых элементов известен (из типа фактического параметра, можно непосредственно сравнивать элементы, а не передавать указатель на производящую сравнение функцию. Кроме того, нет нужды возиться с операцией `sizeof`. Такое решение кажется более красивым и к тому же оно более эффективно, чем обычное. Все же оно сталкивается с трудностью. Для некоторых типов операция < не определена, а для других, например `char*`, ее определение противоречит тому, что требуется в приведенном определении шаблонной функции. (Действительно, нам нужно сравнивать не указатели на строки, а сами строки). В первом случае попытка создать вариант `sort()` для таких типов закончится неудачей (на что и следует надеяться) , а во втором появиться функция, производящая

неожиданный результат.

Чтобы правильно сортировать вектор из элементов `char*` мы можем просто задать самостоятельно подходящее определение функции `sort(Vector<char*>&)`:

```
void sort(Vector<char*>& v)
 {
   unsigned n = v.size();

   for (int i=0; i<n-1; i++)
     for ( int j=n-1; i<j; j--)
        if (strcmp(v[j],v[j-1])<0) {
        // меняем местами v[j] и v[j-1]
          char* temp = v[j];
          v[j] = v[j-1];
          v[j-1] = temp;
        }
}
```

Поскольку для векторов из указателей на строки пользователь дал свое особое определение функции `sort()`, оно и будет использоваться, а создавать для нее определение по шаблону с параметром типа `Vector<char*>&` не нужно. Возможность дать для особо важных или "необычных" типов свое определение шаблонной функции дает ценное качество гибкости в программировании и может быть важным средством доведения программы до оптимальных характеристик.

8.4.2 Производные классы позволяют ввести новые операции

В предыдущем разделе функция сравнения была "встроенной" в теле `sort()` (просто использовалась операция <). Возможно другое решение, когда ее предоставляет сам шаблонный класс `Vector`. Однако, такое решение имеет смысл только при условии, что для типов элементов возможно осмысленное понятие сравнения. Обычно в такой ситуации функцию `sort()` определяют только для векторов, на

которых определена операция < :

```
template<class T> void sort(SortableVector<T>& v)
 {
   unsigned n = v.size();

   for (int i=0; i<n-1; i++)
      for (int j=n-1; i<j; j--)
         if (v.lessthan(v[j],v[j-1])) {
         // меняем местами v[j] и v[j-1]
           T temp = v[j];
           v[j] = v[j-1];
           v[j-1] = temp;
         }
}
```

Класс `SortableVector` (сортируемый вектор) можно определить так:

```
template<class T> class SortableVector
                    : public Vector<T>, public Comparator<T> {
                  public:
                     SortableVector(int s) : Vector<T>(s) { }
                  };
```

Чтобы это определение имело смысл еще надо определить шаблонный класс `Comparator` (сравниватель):

```
template<class T> class Comparator {
 public:
   inline static lessthan(T& a, T& b)  // функция "меньше"
     { return strcmp(a,b)<0; }
   // ...
};
```

Чтобы устранить тот эффект, что в нашем случае операция < дает не тот результат для типа `char*`, мы определим специальный вариант класса сравнивателя:

```
class Comparator<char*> {
```

```
public:
  inline static lessthan(const char* a, const char* b)
  // функция "меньше"
    { return strcmp(a,b)<0; }
  // ...
            };
```

Описание специального варианта шаблонного класса для `char*` полностью подобно тому, как в предыдущем разделе мы определили специальный вариант шаблонной функции для этой же цели. Чтобы описание специального варианта шаблонного класса сработало, транслятор должен обнаружить его до использования. Иначе будет использоваться создаваемый по шаблону класс. Поскольку класс должен иметь в точности одно определение в программе, использовать и специальный вариант класса, и вариант, создаваемый по шаблону, будет ошибкой.

Поскольку у нас уже специальный вариант класса `Comparator` для `char*`, специальный вариант класса `SortableVector` для `char*` не нужен, и можем, наконец, попробовать сортировку:

```
void f(SortableVector<int>& vi,
       SortableVector<String>& vc,
       SortableVector<int>& vi2,
       SortableVector<char*>& vs)
{
  sort(vi);
  sort(vc);
  sort(vi2);
  sort(vs);
}
```

Возможно иметь два вида векторов и не очень хорошо, но, по крайней мере, `SortableVector` является производным от `Vector`. Значит если в функции не нужна сортировка, то в ней и не надо знать о классе `SortableVector`, а там, где нужно, сработает неявное преобразование ссылки на производный класс в ссылку на общий базовый класс. Мы ввели производный от `Vector` и `Comparator` класс `SortableVector` (вместо того, чтобы добавить функции к

классу, производному от одного `Vector`) просто потому, что класс `Comparator` уже напрашивался в предыдущим примере. Такой подход типичен при создании больших библиотек. Класс `Comparator` естественный кандидат для библиотеки, поскольку в нем можно указать различные требования к операциям сравнения для разных типов.

8.4.3 Передача операций как параметров функций

Можно не задавать функцию сравнения как часть типа Vector, а передавать ее как второй параметр функции `sort()`. Этот параметр является объектом класса, в котором определена реализация операции сравнения:

```
template<class T> void sort(Vector<T>& v, Comparator<T>& cmp)
{
  unsigned n = v.size();

  for (int i = 0; i<n-1; i++)
     for ( int j = n-1; i<j; j--)
        if (cmp.lessthan(v[j],v[j-1])) {
        // меняем местами v[j] и v[j-1]
           T temp = v[j];
           v[j] = v[j-1];
           v[j-1] = temp;
        }
}
```

Этот вариант можно рассматривать как обобщение традиционного приема, когда операция сравнения передается как указатель на функцию.

Воспользоваться этим можно так:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
  Comparator<int> ci;
```

```
    Comparator<char*> cs;
    Comparator<String> cc;

    sort(vi,ci);   // sort(Vector<int>&);
    sort(vc,cc);   // sort(Vector<String>&);
    sort(vi2,ci);  // sort(Vector<int>&);
    sort(vs,cs);   // sort(Vector<char*>&);
  }
```

Отметим, что включение в шаблон класса `Comparator` как параметра гарантирует, что функция `lessthan` будет реализовываться подстановкой. В частности, это полезно, если в шаблонной функции используется несколько функций, а не одна операция сравнения, и особенно это полезно, когда эти функции зависят от хранящихся в том же объекте данных.

8.4.4 Неявная передача операций

В примере из предыдущего раздела объекты `Comparator` на самом деле никак не использовались в вычислениях. Это просто "искусственные" параметры, нужные для правильного контроля типов. Введение таких параметров достаточно общий и полезный прием, хотя и не слишком красивый. Однако, если объект используется только для передачи операции (как и было в нашем случае), т.е. в вызываемой функции не используется ни значение, ни адрес объекта, то можно вместо этого передавать операцию неявно:

```
template<class T> void sort(Vector<T>& v)
  {
    unsigned n = v.size();

    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (Comparator<T>::lessthan(v[j],v[j-1])) {
            // меняем местами v[j] и v[j-1]
              T temp = v[j];
              v[j] = v[j-1];
              v[j-1] = temp;
```

```
        }
    }
```

В результате мы приходим к первоначальному варианту использования `sort()`:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{

    sort(vi);   // sort(Vector<int>&);
    sort(vc);   // sort(Vector<String>&);
    sort(vi2);  // sort(Vector<int>&);
    sort(vs);   // sort(Vector<char*>&);
}
```

Основное преимущество этого варианта, как и двух предыдущих, по сравнению с исходным вариантом в том, что часть программы, занятая собственно сортировкой, отделена от частей, в которых находятся такие операции, работающие с элементами, как, например, `lessthan`. Необходимость подобного разделения растет с ростом программы, и особенный интерес это разделение представляет при проектировании библиотек. Здесь создатель библиотеки не может знать типы параметров шаблона, а пользователи не знают (или не хотят знать) специфику используемых в шаблоне алгоритмов. В частности, если бы в функции `sort()` использовался более сложный, оптимизированный и рассчитанный на коммерческое применение алгоритм, пользователь не очень бы стремился написать свою особую версию для типа `char*`, как это было сделано в § 8.4.1. Хотя реализация класса `Comparator` для специального случая `char*` тривиальна и может использоваться и в других ситуациях.

8.4.5 Введение операций с помощью параметров шаблонного класса

Возможны ситуации, когда неявность связи между шаблонной функцией `sort()` и шаблонным классом `Comparator` создает трудности. Неявную связь легко упустить из виду и в то же время разобраться в ней может быть непросто. Кроме того, поскольку эта связь "встроена" в функцию `sort()`, невозможно использовать эту функцию для сортировки векторов одного типа, если операция сравнения рассчитана на другой тип (см. упражнение 3 в § 8.9). Поместив функцию `sort()` в класс, мы можем явно задавать связь с классом `Comparator`:

```
template<class T, class Comp> class Sort {
 public:
   static void sort(Vector<T>&);
 };
```

Не хочется повторять тип элемента, и это можно не делать, если использовать `typedef` в шаблоне `Comparator`:

```
template<class T> class Comparator {
 public:
   typedef T T;  // определение Comparator<T>::T
   static int lessthan(T& a, T& b) {
     return a < b;
   }
   // ...
 };
```

В специальном варианте для указателей на строки это определение выглядит так:

```
class Comparator<char*> {
 public:
   typedef char* T;
   static int lessthan(T a, T b) {
     return strcmp(a,b) < 0;
   }
   // ...
 };
```

После этих изменений можно убрать параметр, задающий тип элемента,

из класса `Sort`:

```
template<class T, class Comp> class Sort {
 public:
   static void sort(Vector<T>&);
 };
```

Теперь можно использовать сортировку так:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
  Sort< int,Comparator<int> >::sort(vi);
  Sort< String,Comparator<String> >::sort(vc);
  Sort< int,Comparator<int> >::sort(vi2);
  Sort< char*,Comparator<char*> >::sort(vs);
}
```

и определить функцию `sort()` следующим образом:

```
template<class T, class Comp>
void Sort<T,Comp>::sort(Vector<T>& v)
{
  for (int i=0; i<n-1; i++)
     for (int j=n-1; i<j; j--)
        if (Comp::lessthan(v[j],v[j-1])) {
          T temp = v[j];
          v[j] = v[j-1];
          v[j-1] = temp;
        }
}
```

Последний вариант ярко демонстрирует как можно соединять в одну программу отдельные ее части. Этот пример можно еще больше упростить, если использовать класс сравнителя (Comp) в качестве единственного параметра шаблона. В этом случае в определениях класса `Sort` и функции `Sort::sort()` тип элемента будет обозначаться как `Comp::T`.

8.5 Разрешение перегрузки для шаблонной функции

К параметрам шаблонной функции нельзя применять никаких преобразований типа. Вместо этого при необходимости создаются новые варианты функции:

```
template<class T> T sqrt(t);

void f(int i, double d, complex z)
{
  complex z1 = sqrt(i);  // sqrt(int)
  complex z2 = sqrt(d);  // sqrt(double)
  complex z3 = sqrt(z);  // sqrt(complex)
  // ...
}
```

Здесь для всех трех типов параметров будет создаваться по шаблону своя функция `sqrt`. Если пользователь захочет чего-нибудь иного, например вызвать `sqrt(double)`, задавая параметр `int`, нужно использовать явное преобразование типа:

```
template<class T> T sqrt(T);

void f(int i, double d, complex z)
{
  complex z1 = sqrt(double(i));  // sqrt(double)
  complex z2 = sqrt(d);  // sqrt(double)
  complex z3 = sqrt(z);  // sqrt(complex)
  // ...
}
```

В этом примере по шаблону будут создаваться определения только для sqrt(double) и sqrt(complex).

Шаблонная функция может перегружаться как простой, так и шаблонной функцией того же имени. Разрешение перегрузки как шаблонных, так и обычных функций с одинаковыми именами происходит за три шага:

Эти правила слишком строгие, и, по всей видимости будут ослаблены, чтобы разрешить преобразования ссылок и указателей, а, возможно, и другие стандартные преобразования. Как обычно, при таких преобразованиях будет действовать контроль однозначности.

1. Найти функцию с точным сопоставлением параметров (§ R.13.2); если такая есть, вызвать ее.
2. Найти шаблон типа, по которому можно создать вызываемую функцию с точным сопоставлением параметров; если такая есть, вызвать ее.
3. Попробовать правила разрешения для обычных функций (§ R13.2); если функция найдена по этим правилам, вызвать ее, иначе вызов является ошибкой.

В любом случае, если на первом шаге найдено более одной функции, вызов считается неоднозначным и является ошибкой. Например:

```
template<class T>
 T max(T a, T b) { return a>b?a:b; };

                  void f(int a, int b, char c, char d)
                  {
 int m1 = max(a,b);  // max(int,int)
 char m2 = max(c,d); // max(char,char)
 int m3 = max(a,c);  // ошибка: невозможно
                  // создать max(int,char)
                  }
```

Поскольку до генерации функции по шаблону не применяется никаких преобразований типа (правило [2]), последний вызов в этом примере нельзя разрешить как `max(a,int(c))`. Это может сделать сам пользователь, явно описав функцию `max(int,int)`. Тогда вступает в силу правило [3]:

```
template<class T>
  T max(T a, T b) { return a>b?a:b; }

int max(int,int);
```

```
void f(int a, int b, char c, char d)
{
  int m1 = max(a,b);     // max(int,int)
  char m2 = max(c,d);    // max(char,char)
  int m3 = max(a,c);     // max(int,int)
}
```

Программисту не нужно давать определение функции `max(int,int)`, оно по умолчанию будет создано по шаблону.

Можно определить шаблон `max` так, чтобы сработал первоначальный вариант нашего примера:

```
template<class T1, class T2>
   T1 max(T1 a, T2 b) { return a>b?a:b; };

void f(int a, int b, char c, char d)
{
  int m1 = max(a,b);    // int max(int,int)
  char m2 = max(c,d);   // char max(char,char)
  int m3 = max(a,c);    // max(int,char)
}
```

Однако, в C и C++ правила для встроенных типов и операций над ними таковы, что использовать подобный шаблон с двумя параметрами может быть совсем непросто. Так, может оказаться неверно задавать тип результата функции как первый параметр (T1), или, по крайней мере, это может привести к неожиданному результату, например для вызова

```
max(c,i);  // char max(char,int)
```

Если в шаблоне для функции, которая может иметь множество параметров с различными арифметическими типами, используются два параметра, то в результате по шаблону будет порождаться слишком большое число определений разных функций. Более разумно добиваться преобразования типа, явно описав функцию с нужными типами.

8.6 Параметры шаблона типа

Параметр шаблона типа не обязательно должен быть именем типа (см. § R.14.2). Помимо имен типов можно задавать строки, имена функций и выражения-константы. Иногда бывает нужно задать как параметр целое:

```
template<class T, int sz> class buffer {
    T v[sz];   // буфер объектов произвольного типа
    // ...
};

void f()
{
  buffer<char,128> buf1;
  buffer<complex,20> buf2;
  // ...
}
```

Мы сделали `sz` параметром шаблона `buffer`, а не его объектов, и это означает, что размер буфера должен быть известен на стадии трансляции, чтобы его объекты было можно размещать, не используя свободную память. Благодаря этому свойству такие шаблоны как `buffer` полезны для реализации контейнерных классов, поскольку для последних первостепенным фактором, определяющим их эффективность, является возможность размещать их вне свободной памяти. Например, если в реализации класса `string` короткие строки размещаются в стеке, это дает существенный выигрыш для программы, поскольку в большинстве задач практически все строки очень короткие. Для реализации таких типов как раз и может пригодиться шаблон `buffer`.

Каждый параметр шаблона типа для функции должен влиять на тип функции, и это влияние выражается в том, что он участвует по крайней мере в одном из типов формальных параметров функций, создаваемых по шаблону. Это нужно для того, чтобы функции можно было выбирать и создавать, основываясь только на их параметрах:

```
template<class T> void f1(T);     // нормально
```

```
template<class T> void f2(T*);    // нормально
template<class T> T f3(int);      // ошибка
template<int i> void f4(int[][i]);  // ошибка
template<int i> void f5(int = i);   // ошибка
template<class T, class C> void f6(T);  // ошибка
template<class T> void f7(const T&, complex);  // нормально
template<class T> void f8(Vector< List<T> >);  // нормально
```

Здесь все ошибки вызваны тем, что параметр-тип шаблона никак не влияет на формальные параметры функций.

Подобного ограничения нет в шаблонах типа для классов. Дело в том, что параметр для такого шаблона нужно указывать всякий раз, когда описывается объект шаблонного класса. С другой стороны, для шаблонных классов возникает вопрос: когда два созданных по шаблону типа можно считать одинаковыми? Два имени шаблонного класса обозначают один и тот же класс, если совпадают имена их шаблонов, а используемые в этих именах параметры имеют одинаковые значения (с учетом возможных определений `typedef`, вычисления выражений-констант и т.д.). Вернемся к шаблону `buffer`:

```
template<class T, int sz>
 class buffer {
   T v[sz];
   // ...
 };

void f()
{
  buffer<char,20> buf1;
  buffer<complex,20> buf2;
  buffer<char,20> buf3;
  buffer<char,100> buf4;

  buf1 = buf2;  // ошибка: несоответствие типов
  buf1 = buf3;  // нормально
  buf1 = buf4;  // ошибка: несоответствие типов
  // ...
}
```

Если в шаблоне типа для класса используются параметры, задающие не типы, возможно появление конструкций, выглядящих двусмысленно:

```
template<int i>
class X { /* ... */ };

void f(int a, int b)
{
  X < a > b>;  // Как это понимать: X<a> b и потом
               // недопустимая лексема, или X< (a>b) >; ?
}
```

Этот пример синтаксически ошибочен, поскольку первая угловая скобка > завершает параметр шаблона. В маловероятном случае, когда вам понадобится параметр шаблона, являющийся выражением "больше чем", используйте скобки: `X< (a>b)>`.

8.7 Шаблоны типа и производные классы

Мы уже видели, что сочетание производных классов (наследование) и шаблонов типа может быть мощным средством. Шаблон типа выражает общность между всеми типами, которые используются как его параметры, а базовый класс выражает общность между всеми представлениями (объектами) и называется интерфейсом. Здесь возможны некоторые простые недоразумения, которых надо избегать.

Два созданных по одному шаблону типа будут различны и между ними невозможно отношение наследования кроме единственного случая, когда у этих типов идентичны параметры шаблона. Например:

```
template<class T>
class Vector { /* ... */ }

Vector<int> v1;
Vector<short> v2;
Vector<int> v3;
```

Здесь `v1` и `v3` одного типа, а `v2` имеет совершенно другой тип. Из того факта, что `short` неявно преобразуется в `int`, не следует, что есть

неявное преобразование `Vector<short>` в `Vector<int>`:

```
v2 = v3;  // несоответствие типов
```

Но этого и следовало ожидать, поскольку нет встроенного преобразования int[] в short[].

Аналогичный пример:

```
class circle: public shape { /* ... */ };

Vector<circle*> v4;
Vector<shape*> v5;
Vector<circle*> v6;
```

Здесь `v4` и `v6` одного типа, а `v5` имеет совершенно другой тип. Из того факта, что существует неявное преобразование `circle` в `shape` и `circle*` в `shape*`, не следует, что есть неявные преобразования `Vector<circle*>` в `Vector<shape*>` или `Vector<circle*>*` в `Vector<shape*>*`:

```
v5 = v6;  // несоответствие типов
```

Дело в том, что в общем случае структура (представление) класса, созданного по шаблону типа, такова, что для нее не предполагаются отношения наследования. Так, созданный по шаблону класс может содержать объект типа, заданного в шаблоне как параметр, а не просто указатель на него. Кроме того, допущение подобных преобразований приводит к нарушению контроля типов:

```
void f(Vector<circle>* pc)
{
  Vector<shape>* ps = pc;  // ошибка: несоответствие типов
  (*ps)[2] = new square;   // круглую ножку суем в квадратное
          // отверстие (память выделена для
          // square, а используется для circle
}
```

На примерах шаблонов `Islist`, `Tlink`, `Slist`, `Splist`, `Islist_iter`, `Slist_iter` и `SortableVector` мы видели, что

шаблоны типа дают удобное средство для создания целых семейств классов. Без шаблонов создание таких семейств только с помощью производных классов может быть утомительным занятием, а значит, ведущим к ошибкам. С другой стороны, если отказаться от производных классов и использовать только шаблоны, то появляется множество копий функций-членов шаблонных классов, множество копий описательной части шаблонных классов и во множестве повторяются функции, использующие шаблоны типа.

8.7.1 Задание реализации с помощью параметров шаблона

В контейнерных классах часто приходится выделять память. Иногда бывает необходимо (или просто удобно) дать пользователю возможность выбирать из нескольких вариантов выделения памяти, а также позволить ему задавать свой вариант. Это можно сделать несколькими способами. Один из способов состоит в том, что определяется шаблон типа для создания нового класса, в интерфейс которого входит описание соответствующего контейнера и класса, производящего выделение памяти по способу, описанному в § 6.7.2:

```
template<class T, class A> class Controlled_container
    : public Container<T>, private A {
    // ...
    void some_function()
    {
      // ...
      T* p = new(A::operator new(sizeof(T))) T;
      // ...
    }
    // ...
};
```

Шаблон типа здесь необходим, поскольку мы создаем контейнерный класс. Наследование от `Container<T>` нужно, чтобы класс `Controlled_container` можно было использовать как контейнерный класс. Шаблон типа с параметром A позволит нам

использовать различные функции размещения:

```
class Shared : public Arena { /* ... */ };
class Fast_allocator { /* ... */ };

Controlled_container<Process_descriptor,Shared> ptbl;

Controlled_container<Node,Fast_allocator> tree;

Controlled_container<Personell_record,Persistent> payroll;
```

Это универсальный способ предоставлять производным классам содержательную информацию о реализации. Его положительными качествами являются систематичность и возможность использовать функции-подстановки. Для этого способа характерны необычно длинные имена. Впрочем, как обычно, `typedef` позволяет задать синонимы для слишком длинных имен типов:

```
typedef
Controlled_container<Personell_record,Persistent> pp_record;

pp_record payroll;
```

Обычно шаблон типа для создания такого класса как `pp_record` используют только в том случае, когда добавляемая информация по реализации достаточно существенна, чтобы не вносить ее в производный класс ручным программированием. Примером такого шаблона может быть общий (возможно, для некоторых библиотек стандартный) шаблонный класс Comparator (§ 8.4.2), а также нетривиальные (возможно, стандартные для некоторых библиотек) классы `Allocator` (классы для выделения памяти). Отметим, что построение производных классов в таких примерах идет по "основному проспекту", который определяет интерфейс с пользователем (в нашем примере это `Container`). Но есть и "боковые улицы", задающие детали реализации.

8.8 Ассоциативный массив

Из всех универсальных невстроенных типов самым полезным, по всей видимости, является ассоциативный массив. Его часто называют таблицей (map), а иногда словарем, и он хранит пары значений. Имея одно из значений, называемое ключом, можно получить доступ к другому, называемому просто значением. Ассоциативный массив можно представлять как массив, в котором индекс не обязан быть целым:

```
template<class K, class V> class Map {
  // ...
public:
  V& operator[](const K&); // найти V, соответствующее K
 // и вернуть ссылку на него
  // ...
};
```

Здесь ключ типа K обозначает значение типа `V`. Предполагается, что ключи можно сравнивать с помощью операций == и <, так что массив можно хранить в упорядоченном виде. Отметим, что класс `Map` отличается от типа assoc из § 7.8 тем, что для него нужна операция "меньше чем", а не функция хэширования.

Приведем простую программу подсчета слов, в которой используются шаблон `Map` и тип `String`:

```
#include <String.h>
#include <iostream.h>
#include "Map.h"

int main()
{
  Map<String,int> count;
  String word;

  while (cin >> word) count[word]++;

  for (Mapiter<String,int> p = count.first(); p; p++)
    cout << p.value() << '\t' << p.key() << '\n';
```

```
  return 0;
}
```

Мы используем тип `String` для того, чтобы не беспокоиться о выделении памяти и переполнении ее, о чем приходится помнить, используя тип `char*`. Итератор `Mapiter` нужен для выбора по порядку всех значений массива. Итерация в `Mapiter` задается как имитация работы с указателями. Если входной поток имеет вид

```
It was new. It was singular. It was simple. It must succeed.
```

программа выдаст

```
4       It
1       must
1       new.
1       simple.
1       singular.
1       succeed.
3       was.
```

Конечно, определить ассоциативный массив можно многими способами, а имея определение `Map` и связанного с ним класса итератора, мы можем предложить много способов для их реализации. Здесь выбран тривиальный способ реализации. Используется линейный поиск, который не подходит для больших массивов. Естественно, рассчитанная на коммерческое применение реализация будет создаваться, исходя из требований быстрого поиска и компактности представления (см. упражнение 4 из § 8.9).

Мы используем список с двойной связью `Link`:

```
template<class K, class V> class Map;
template<class K, class V> class Mapiter;

template<class K, class V> class Link {
  friend class Map<K,V>;
  friend class Mapiter<K,V>;
private:
```

```
    const K key;
    V value;

    Link* pre;
    Link* suc;

    Link(const K& k, const V& v) : key(k), value(v) { }
    ~Link() { delete suc; } // рекурсивное удаление всех
            // объектов в списке
};
```

Каждый объект `Link` содержит пару (ключ, значение). Классы описаны в `Link` как друзья, и это гарантирует, что объекты `Link` можно создавать, работать с ними и уничтожать только с помощью соответствующих классов итератора и `Map`. Обратите внимание на предварительные описания шаблонных классов `Map` и `Mapiter`.

Шаблон `Map` можно определить так:

```
template<class K, class V> class Map {
    friend class Mapiter<K,V>;
    Link<K,V>* head;
    Link<K,V>* current;
    V def_val;
    K def_key;
    int sz;

    void find(const K&);
    void init() { sz = 0; head = 0; current = 0; }

public:

    Map() { init(); }
    Map(const K& k, const V& d)
      : def_key(k), def_val(d) { init(); }
    ~Map() { delete head; }  // рекурсивное удаление
     // всех объектов в списке
    Map(const Map&);
    Map& operator= (const Map&);
```

```
    V& operator[] (const K&);

    int size() const { return sz; }
    void clear() { delete head; init(); }
    void remove(const K& k);

        // функции для итерации

    Mapiter<K,V> element(const K& k)
    {
      (void) operator[](k);  // сделать k текущим элементом
      return Mapiter<K,V>(this,current);
    }
    Mapiter<K,V> first();
    Mapiter<K,V> last();
  };
```

Элементы хранятся в упорядоченном списке с двойной связью. Для простоты ничего не делается для ускорения поиска (см. упражнение 4 из § 8.9). Ключевой здесь является функция `operator[]()`:

```
template<class K, class V>
V& Map<K,V>::operator[] (const K& k)
{
  if (head == 0) {
    current = head = new Link<K,V>(k,def_val);
    current->pre = current->suc = 0;
    return current->value;
  }

  Link<K,V>* p = head;
  for (;;) {
    if (p->key == k) { // найдено
      current = p;
      return current->value;
    }

  if (k < p->key) { // вставить перед p (в начало)
```

```
    current = new Link<K,V>(k,def_val);
    current->pre = p->pre;
    current->suc = p;
    if (p == head)  // текущий элемент становится начальным
      head = current;
    else
      p->pre->suc = current;
    p->pre = current;
    return current->value;
  }

  Link<K,V>* s = p->suc;
  if (s == 0) {  // вставить после p (в конец)
    current = new Link<K,V>(k,def_val);
    current->pre = p;
    current->suc = 0;
    p->suc = current;
    return current->value;
  }
  p = s;
}
      }
```

Операция индексации возвращает ссылку на значение, которое соответствует заданному как параметр ключу. Если такое значение не найдено, возвращается новый элемент со стандартным значением. Это позволяет использовать операцию индексации в левой части присваивания. Стандартные значения для ключей и значений устанавливаются конструкторами `Map`. В операции индексации определяется значение `current`, используемое итераторами.

Реализация остальных функций-членов оставлена в качестве упражнения:

```
template<class K, class V>
void  Map<K,V>::remove(const K& k)
{
  // см. упражнение 2 из 8.10
}
```

```
template<class K, class V>
Map<K,V>::Map(const Map<K,V>& m)
{
  // копирование таблицы Map и всех ее элементов
}

template<class K, class V>
Map& Map<K,V>::operator=(const Map<K,V>& m)
{
  // копирование таблицы Map и всех ее элементов
}
```

Теперь нам осталось только определить итерацию. В классе `Map` есть функции-члены `first()`, `last()` и `element(const K&)`, которые возвращают итератор, установленный соответственно на первый, последний или задаваемый ключом-параметром элемент. Сделать это можно, поскольку элементы хранятся в упорядоченном по ключам виде.

Итератор Mapiter для Map определяется так:

```
template<class K, class V> class Mapiter {
  friend class Map<K,V>;

  Map<K,V>* m;
  Link<K,V>* p;

  Mapiter(Map<K,V>* mm, Link<K,V>* pp)
    { m = mm; p = pp; }
public:
  Mapiter() { m = 0; p = 0; }
  Mapiter(Map<K,V>& mm);

  operator void*() { return p; }

  const K& key();
  V& value();
```

```
  Mapiter& operator--(); // префиксная
  void operator--(int);  // постфиксная
  Mapiter& operator++(); // префиксная
  void operator++(int);  // постфиксная
};
```

После позиционирования итератора функции `key()` и `value()` из `Mapiter` выдают ключ и значение того элемента, на который установлен итератор.

```
template<class K, class V> const K& Mapiter<K,V>::key()
{
 if (p) return p->key; else return m->def_key;
}

template<class K, class V> V& Mapiter<K,V>::value()
{
 if (p) return p->value; else return m->def_val;
}
```

По аналогии с указателями определены операции `++` и `--` для продвижения по элементам `Map` вперед и назад:

```
Mapiter<K,V>& Mapiter<K,V>::operator--() //префиксный декремент
{
 if (p) p = p->pre;
 return *this;
}

void Mapiter<K,V>::operator--(int)  // постфиксный декремент
{
 if (p) p = p->pre;
}

Mapiter<K,V>& Mapiter<K,V>::operator++() // префиксный инкремент
{
 if (p) p = p->suc;
 return *this;
}
```

```
void Mapiter<K,V>::operator++(int)   // постфиксный инкремент
{
  if (p) p = p->suc;
}
```

Постфиксные операции определены так, что они не возвращают никакого значения. Дело в том, что затраты на создание и передачу нового объекта `Mapiter` на каждом шаге итерации значительны, а польза от него будет не велика.

Объект `Mapiter` можно инициализировать так, чтобы он был установлен на начало `Map`:

```
template<class K, class V> Mapiter<K,V>::Mapiter(Map<K,V>& mm)
{
    m == &mm; p = m->head;
 }
```

Операция преобразования `operator void*()` возвращает нуль, если итератор не установлен на элемент `Map`, и ненулевое значение иначе. Значит можно проверять итератор `iter`, например, так:

```
void f(Mapiter<const char*, Shape*>& iter)
{
  // ...
  if (iter) {
   // установлен на элемент таблицы
  }
  else {
   // не установлен на элемент таблицы
  }

  // ...
}
```

Аналогичный прием используется для контроля потоковых операций ввода-вывода в § 10.3.2.

Если итератор не установлен на элемент таблицы, его функции `key()` и `value()` возвращают ссылки на стандартные объекты.

Если после всех этих определений вы забыли их назначение, можно привести еще одну небольшую программу, использующую таблицу `Map`. Пусть входной поток является списком пар значений следующего вида:

```
hammer      2
nail      100
saw         3
saw         4
hammer      7
nail      1000
nail      250
```

Нужно отсортировать список так, чтобы значения, соответствующие одному предмету, складывались, и напечатать получившийся список вместе с итоговым значением:

```
hammer      9
nail      1350
saw         7
------------------
total     1366
```

Вначале напишем функцию, которая читает входные строки и заносит предметы с их количеством в таблицу. Ключом в этой таблице является первое слово строки:

```
template<class K, class V>
  void readlines(Map<K,V>&key)
  {
    K word;
    while (cin >> word) {
      V val  = 0;
      if (cin >> val)
        key[word] +=val;
      else
        return;
    }
```

```
}
```

Теперь можно написать простую программу, вызывающую функцию `readlines()` и печатающую получившуюся таблицу:

```
main()
{
  Map<String,int> tbl("nil",0);
  readlines(tbl);

  int total = 0;
  for (Mapiter<String,int> p(tbl); p; ++p) {
     int val = p.value();
     total +=val;
     cout << p.key() << '\t' << val << '\n';
  }

  cout << "--------------------\n";
  cout << "total\t" << total << '\n';
}
```

8.9 Упражнения

1. (*2) Определите семейство списков с двойной связью, которые будут двойниками списков с одной связью, определенных в § 8.3.
2. (*3) Определите шаблон типа `String`, параметром которого является тип символа. Покажите как его можно использовать не только для обычных символов, но и для гипотетического класса `lchar`, который представляет символы не из английского алфавита или расширенный набор символов. Нужно постараться так определить `String`, чтобы пользователь не заметил ухудшения характеристик программы по памяти и времени или в удобстве по сравнению с обычным строковым классом.
3. (*1.5) Определите класс `Record` (запись) с двумя членами-данными: count (количество) и price (цена). Упорядочите вектор из таких записей по каждому из членов. При этом нельзя изменять функцию сортировки и шаблон `Vector`.
4. (*2) Завершите определения шаблонного класса `Map`, написав

недостающие функции-члены.

5. (*2) Задайте другую реализацию `Map` из § 8.8, используя списочный класс с двойной связью.
6. (*2.5) Задайте другую реализацию `Map` из § 8.8, используя сбалансированное дерево. Такие деревья описаны в § 6.2.3 книги Д. Кнут "Искусство программирования для ЭВМ" т.1, "Мир", 1978 [K].
7. (*2) Сравните качество двух реализаций `Map`. В первой используется класс Link со своей собственной функцией размещения, а во второй - без нее.
8. (*3) Сравните производительность программы подсчета слов из § 8.8 и такой же программы, не использующей класса `Map`. Операции ввода-вывода должны одинаково использоваться в обеих программах. Сравните несколько таких программ, использующих разные варианты класса `Map`, в том числе и класс из вашей библиотеки, если он там есть.
9. (*2.5) С помощью класса `Map` реализуйте топологическую сортировку. Она описана в [K] т.1, стр. 323-332. (см. упражнение 6).
10. (*2) Модифицируйте программу из § 8.8 так, чтобы она работала правильно для длинных имен и для имен, содержащих пробелы (например, "thumb back").
11. (*2) Определите шаблон типа для чтения различных видов строк, например, таких (предмет, количество, цена).
12. (*2) Определите класс `Sort` из § 8.4.5, использующий сортировку по методу Шелла. Покажите как можно задать метод сортировки с помощью параметра шаблона. Алгоритм сортировки описан в [K] т.3, § 5.2.1 (см. упражнение 6).
13. (*1) Измените определения `Map` и `Mapiter` так, чтобы постфиксные операции `++` и `--` возвращали объект `Mapiter`.
14. (*1.5) Используйте шаблоны типа в стиле модульного программирования, как это было показано в § 8.4.5 и напишите функцию сортировки, рассчитанную сразу на `Vector<T>` и `T[]`.

Механизм обработки особых ситуаций

В этой лекции описан механизм обработки особых ситуаций и некоторые, основывающиеся на нем, способы обработки ошибок. Механизм состоит в запуске особой ситуации, которую должен перехватить специальный обработчик. Описываются правила перехвата особых ситуаций и правила реакции на неперехваченные и неожиданные особые ситуации. Целые группы особых ситуаций можно определить как производные классы. Описывается способ, использующий деструкторы и обработку особых ситуаций, который обеспечивает надежное и скрытое от пользователя управление ресурсами.

9.1 Обработка ошибок

Я прервал вас, поэтому не прерывайте меня.

Уинстон Черчилл

Создатель библиотеки способен обнаружить динамические ошибки, но не представляет, какой в общем случае должна быть реакция на них. Пользователь библиотеки способен написать реакцию на такие ошибки, но не в силах их обнаружить. Если бы он мог, то сам разобрался бы с ошибками в своей программе, и их не пришлось бы выявлять в библиотечных функциях. Для решения этой проблемы в язык введено понятие особой ситуации.

Только недавно комитетом по стандартизации C++ особые ситуации были включены в стандарт языка, но на время написания этой книги они еще не вошли в большинство реализаций.

Суть этого понятия в том, что функция, которая обнаружила ошибку и не может справиться с нею, запускает особую ситуацию, рассчитывая, что устранить проблему можно в той функции, которая прямо или опосредованно вызывала первую. Если функция рассчитана на обработку ошибок некоторого вида, она может указать это явно, как готовность перехватить данную особую ситуацию.

Рассмотрим в качестве примера как для класса `Vector` можно

представлять и обрабатывать особые ситуации, вызванные выходом за границу массива:

```
class Vector {
  int* p;
  int  sz;
public:
  class Range  {  };  // класс для особой ситуации

  int& operator[](int i);

  // ...
};
```

Предполагается, что объекты класса `Range` будут использоваться как особые ситуации, и запускать их можно так:

```
int& Vector::operator[](int i)
{
  if (0<=i && i<sz) return p[i];
  throw Range();
}
```

Если в функции предусмотрена реакция на ошибку недопустимого значения индекса, то ту часть функции, в которой эти ошибки будут перехватываться, надо поместить в оператор `try`. В нем должен быть и обработчик особой ситуации:

```
void f(Vector& v)
{
 // ...

 try {
  do_something(v);  // содержательная часть, работающая с v
 }
 catch (Vector::Range) {
 // обработчик особой ситуации Vector::Range

 // если do_something() завершится неудачно,
```

```
    // нужно как-то среагировать на это

    // сюда мы попадем только в том случае, когда
    // вызов do_something() приведет к вызову Vector::operator[]()
    // из-за недопустимого значения индекса

    }

    // ...
        }
```

Обработчиком особой ситуации называется конструкция

```
catch ( /* ... */ ) {
  // ...
}
```

Ее можно использовать только сразу после блока, начинающегося служебным словом `try`, или сразу после другого обработчика особой ситуации. Служебным является и слово `catch`. После него идет в скобках описание, которое используется аналогично описанию формальных параметров функции, а именно, в нем задается тип объектов, на которые рассчитан обработчик, и, возможно, имена параметров (см. § 9.3). Если в `do_something()` или в любой вызванной из нее функции произойдет ошибка индекса (на любом объекте `Vector`), то обработчик перехватит особую ситуацию и будет выполняться часть, обрабатывающая ошибку. Например, определения следующих функций приведут к запуску обработчика в `f()`:

```
void do_something()
{
  // ...
  crash(v);
  // ...
}

void crash(Vector& v)
{
   v[v.size()+10];  // искусственно вызываем ошибку индекса
```

```
}
```

Процесс запуска и перехвата особой ситуации предполагает просмотр цепочки вызовов от точки запуска особой ситуации до функции, в которой она перехватывается. При этом восстанавливается состояние стека, соответствующее функции, перехватившей ошибку, и при проходе по всей цепочке вызовов для локальных объектов функций из этой цепочки вызываются деструкторы. Подробно это описано в § 9.4.

Если при просмотре всей цепочки вызовов, начиная с запустившей особую ситуацию функции, не обнаружится подходящий обработчик, то программа завершается. Подробно это описано в § 9.7.

Если обработчик перехватил особую ситуацию, то она будет обрабатываться, и другие, рассчитанные на эту ситуацию обработчики не будут рассматриваться. Иными словами, активирован будет только тот обработчик, который находится в самой последней вызывавшейся функции, содержащей соответствующие обработчики. В нашем примере функция `f()` перехватит `Vector::Range`, поэтому эту особую ситуацию нельзя перехватить ни в какой вызывающей `f()` функции:

```
int ff(Vector& v)
{
  try {
        f(v);         // в f() будет перехвачена Vector::Range
  }
  catch (Vector::Range) { // значит сюда мы никогда не попадем
     // ...
  }
}
```

9.1.1 Особые ситуации и традиционная обработка ошибок

Наш способ обработки ошибок по многим параметрам выгодно отличается от более традиционных способов. Перечислим, что может сделать операция индексации `Vector::operator[]()` при

обнаружении недопустимого значения индекса:

1. завершить программу;
2. возвратить значение, трактуемое как "ошибка";
3. возвратить нормальное значение и оставить программу в неопределенном состоянии;
4. вызвать функцию, заданную для реакции на такую ошибку.

Вариант [1] ("завершить программу") реализуется по умолчанию в том случае, когда особая ситуация не была перехвачена. Для большинства ошибок можно и нужно обеспечить лучшую реакцию.

Вариант [2] ("возвратить значение "ошибка"") можно реализовать не всегда, поскольку не всегда удается определить значение "ошибка". Так, в нашем примере любое целое является допустимым значением для результата операции индексации. Если можно выделить такое особое значение, то часто этот вариант все равно оказывается неудобным, поскольку проверять на это значение приходится при каждом вызове. Так можно легко удвоить размер программы. Поэтому для обнаружения всех ошибок этот вариант редко используется последовательно.

Вариант [3] ("оставить программу в неопределенном состоянии") имеет тот недостаток, что вызывавшая функция может не заметить ненормального состояния программы. Например, во многих функциях стандартной библиотеки C для сигнализации об ошибке устанавливается соответствующее значение глобальной переменной `errno`. Однако, в программах пользователя обычно нет достаточно последовательного контроля errno, и в результате возникают наведенные ошибки, вызванные тем, что стандартные функции возвращают не то значение. Кроме того, если в программе есть параллельные вычисления, использование одной глобальной переменной для сигнализации о разных ошибках неизбежно приведет к катастрофе.

Обработка особых ситуаций не предназначалась для тех случаев, на которые рассчитан вариант [4] ("вызвать функцию реакции на ошибку"). Отметим, однако, что если особые ситуации не предусмотрены, то вместо функции реакции на ошибку можно как раз использовать только один из трех перечисленных вариантов. Обсуждение функций реакций

и особых ситуацией будет продолжено в § 9.4.3.

Механизм особых ситуаций успешно заменяет традиционные способы обработки ошибок в тех случаях, когда последние являются неполным, некрасивым или чреватым ошибками решением. Этот механизм позволяет явно отделить часть программы, в которой обрабатываются ошибки, от остальной ее части, тем самым программа становится более понятной и с ней проще работать различным сервисным программам. Свойственный этому механизму регулярный способ обработки ошибок упрощает взаимодействие между раздельно написанными частями программы.

В этом способе обработки ошибок есть для программирующих на C новый момент: стандартная реакция на ошибку (особенно на ошибку в библиотечной функции) состоит в завершении программы. Традиционной была реакция продолжать программу в надежде, что она как-то завершится сама. Поэтому способ, базирующийся на особых ситуациях, делает программу более "хрупкой" в том смысле, что требуется больше усилий и внимания для ее нормального выполнения. Но это все-таки лучше, чем получать неверные результаты на более поздней стадии развития программы (или получать их еще позже, когда программу сочтут завершенной и передадут ничего не подозревающему пользователю). Если завершение программы является неприемлемой реакцией, можно смоделировать традиционную реакцию с помощью перехвата всех особых ситуаций или всех особых ситуаций, принадлежащих специальному классу (§ 9.3.2).

Механизм особых ситуаций можно рассматривать как динамический аналог механизма контроля типов и проверки неоднозначности на стадии трансляции. При таком подходе более важной становится стадия проектирования программы, и требуется большая поддержка процесса выполнения программы, чем для программ на C. Однако, в результате получится более предсказуемая программа, ее будет проще встроить в программную систему, она будет понятнее другим программистам и с ней проще будет работать различным сервисным программам. Можно сказать, что механизм особых ситуаций поддерживает, подобно другим средствам C++, "хороший" стиль программирования, который в таких языках, как C, можно применять только не в полном объеме и на неформальном уровне.

Все же надо сознавать, что обработка ошибок остается трудной задачей, и, хотя механизм особых ситуаций более строгий, чем традиционные способы, он все равно недостаточно структурирован по сравнению с конструкциями, допускающими только локальную передачу управления.

9.1.2 Другие точки зрения на особые ситуации

"Особая ситуация" - одно из тех понятий, которые имеют разный смысл для разных людей. В C++ механизм особых ситуаций предназначен для обработки ошибок. В частности, он предназначен для обработки ошибок в программах, состоящих из независимо создаваемых компонентов.

Этот механизм рассчитан на особые ситуации, возникающие только при последовательном выполнении программы (например, контроль границ массива). Асинхронные особые ситуации такие, например, как прерывания от клавиатуры, нельзя непосредственно обрабатывать с помощью этого механизма. В различных системах существуют другие механизмы, например, сигналы, но они здесь не рассматриваются, поскольку зависят от конкретной системы.

Механизм особых ситуаций является конструкцией с нелокальной передачей управления и его можно рассматривать как вариант оператора return. Поэтому особые ситуации можно использовать для целей, никак не связанных с обработкой ошибок (§ 9.5). Все-таки основным назначением механизма особых ситуаций и темой этой лекции будет обработка ошибок и создание устойчивых к ошибкам программ.

9.2 Различение особых ситуаций

Естественно, в программе возможны несколько различных динамических ошибок. Эти ошибки можно сопоставить с особыми ситуациями, имеющими различные имена. Так, в классе `Vector` обычно приходится выявлять и сообщать об ошибках двух видов: ошибки диапазона и ошибки, вызванные неподходящим для конструктора параметром:

```
class Vector {
  int* p;
  int sz;
public:
  enum { max = 32000 };
  class Range { };  // особая ситуация индекса
  class Size { };   // особая ситуация "неверный размер"
  Vector(int sz);
  int& operator[](int i);

  // ...
        };
```

Как было сказано, операция индексации запускает особую ситуацию `Range`, если ей задан выходящий из диапазона значений индекс. Конструктор запускает особую ситуацию `Size`, если ему задан недопустимый размер вектора:

```
Vector::Vector(int sz)
{
  if (sz<0 || max<sz) throw Size();
  // ...
}
```

Пользователь класса `Vector` может различить эти две особые ситуации, если в проверяемом блоке (т.е. в блоке оператора `try`) укажет обработчики для обеих ситуаций:

```
void f()
{
  try {
     use_vectors();
  }
  catch (Vector::Range) {
     // ...
  }
  catch (Vector::Size) {
     // ...
  }
```

```
}
```

В зависимости от особой ситуации будет выполняться соответствующий обработчик. Если управление дойдет до конца операторов обработчика, следующим будет выполняться оператор, который идет после списка обработчиков:

```
void f()
{
 try {
   use_vectors();
 }
 catch (Vector::Range) {
   // исправить индекс и
   // попробовать опять:
   f();
 }
 catch (Vector::Size) {
   cerr << "Ошибка в конструкторе Vector::Size";
   exit(99);
 }
 // сюда мы попадем, если вообще не было особых ситуаций
 // или после обработки особой ситуации Range
        }
```

Список обработчиков напоминает переключатель, но здесь в теле обработчика операторы `break` не нужны. Синтаксис списка обработчиков отличен от синтаксиса вариантов `case` переключателя частично по этой причине, частично потому, чтобы показать, что каждый обработчик определяет свою область видимости (см. § 9.8).

Не обязательно все особые ситуации перехватывать в одной функции:

```
void f1()
{
 try {
   f2(v);
 }
 catch (Vector::Size) {
```

```
    // ...
    }
}

void f2(Vector& v)
{
  try {
    use_vectors();
  }
  catch (Vector::Range) {
  // ...
  }
}
```

Здесь `f2()` перехватит особую ситуацию `Range`, возникающую в `use_vectors()`, а особая ситуация `Size` будет оставлена для `f1()`.

С точки зрения языка особая ситуация считается обработанной сразу при входе в тело ее обработчика. Поэтому все особые ситуации, запускаемые при выполнении этого обработчика, должны обрабатываться в функциях, вызвавших ту функцию, которая содержит проверяемый блок. Значит в следующем примере не возникнет бесконечного цикла:

```
try {
    // ...
}
catch (input_overflow) {
   // ...
   throw input_overflow();
}
```

Здесь `input_overflow` (переполнение при вводе) - имя глобального класса. Обработчики особых ситуаций могут быть вложенными:

```
try {
   // ...
}
catch (xxii) {
```

```
  try {
    // сложная реакция
  }
  catch (xxii) {
    // ошибка в процессе сложной реакции
  }
}
```

Однако, такая вложенность редко бывает нужна в обычных программах, и чаще всего она является свидетельством плохого стиля.

9.3 Имена особых ситуаций

Особая ситуация перехватывается благодаря своему типу. Однако, запускается ведь не тип, а объект. Если нам нужно передать некоторую информацию из точки запуска в обработчик, то для этого ее следует поместить в запускаемый объект. Например, допустим нужно знать значение индекса, выходящее за границы диапазона:

```
class Vector {
  // ...
public:
  class Range {
  public:
    int index;
    Range(int i) : index(i) { }
  };
  // ...
  int& operator[](int i)
  // ...
        };

        int Vector::operator[](int i)
        {
 if (o<=i && i <sz) return p[i];
 throw Range(i);
        }
```

Чтобы исследовать недопустимое значение индекса, в обработчике

нужно дать имя объекту, представляющему особую ситуацию:

```
void  f(Vector& v)
{
  // ...

 try {
     do_something(v);
 }
 catch (Vector::Range r ) {
    cerr << "недопустимый индекс" << r.index << '\n';
    // ...
 }
 // ...
          }
```

Конструкция в скобках после служебного слова `catch` является по сути описанием и она аналогична описанию формального параметра функции. В ней указывается каким может быть тип параметра (т.е. особой ситуации) и может задаваться имя для фактической, т.е. запущенной, особой ситуации. Вспомним, что в шаблонах типов у нас был выбор для именования особых ситуаций. В каждом созданном по шаблону классе был свой класс особой ситуации:

```
template<class T> class Allocator {
   // ...
   class Exhausted { }
   // ...
   T* get();
          };

          void f(Allocator<int>& ai, Allocator<double>& ad)
          {
 try {
     // ...
 }
 catch (Allocator<int>::Exhausted) {
    // ...
 }
```

```
  catch (Allocator<double>::Exhausted) {
    // ...
  }
}
```

С другой стороны, особая ситуация может быть общей для всех созданных по шаблону классов:

```
class Allocator_Exhausted { };

template<class T> class Allocator {
  // ...
  T* get();
};

void f(Allocator<int>& ai, Allocator<double>& ad)
{
  try {
    // ...
  }
  catch (Allocator_Exhausted) {
    // ...
  }
}
```

Какой способ задания особой ситуации предпочтительней, сказать трудно. Выбор зависит от назначения рассматриваемого шаблона.

9.3.1 Группирование особых ситуаций

Особые ситуации естественным образом разбиваются на семейства. Действительно, логично представлять семейство `Matherr`, в которое входят `Overflow` (переполнение), *`Underflow`* (потеря значимости) и некоторые другие особые ситуации. Семейство `Matherr` образуют особые ситуации, которые могут запускать математические функции стандартной библиотеки.

Один из способов задания такого семейства сводится к определению

`Matherr` как типа, возможные значения которого включают `Overflow` и все остальные:

```
enum { Overflow, Underflow, Zerodivide, /* ... */ };

try {
    // ...
}
catch (Matherr m) {
    switch (m) {
    case Overflow:
        // ...
    case Underflow:
        // ...
    // ...
    }
    // ...
}
```

Другой способ предполагает использование наследования и виртуальных функций, чтобы не вводить переключателя по значению поля типа. Наследование помогает описать семейства особых ситуаций:

```
class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...
```

Часто бывает так, что нужно обработать особую ситуацию `Matherr` не зависимо от того, какая именно ситуация из этого семейства произошла. Наследование позволяет сделать это просто:

```
try {
    // ...
}
catch (Overflow) {
    // обработка Overflow или любой производной ситуации
}
```

```
catch (Matherr) {
    // обработка любой отличной от Overflow ситуации
}
```

В этом примере `Overflow` разбирается отдельно, а все другие особые ситуации из `Matherr` разбираются как один общий случай. Конечно, функция, содержащая `catch` (Matherr), не будет знать какую именно особую ситуацию она перехватывает. Но какой бы она ни была, при входе в обработчик передаваемая ее копия будет `Matherr`. Обычно это как раз то, что нужно. Если это не так, особую ситуацию можно перехватить по ссылке (см. § 9.3.2).

Иерархическое упорядочивание особых ситуаций может играть важную роль для создания ясной структуры программы. Действительно, пусть такое упорядочивание отсутствует, и нужно обработать все особые ситуации стандартной библиотеки математических функций. Для этого придется до бесконечности перечислять все возможные особые ситуации:

```
try {
    // ...
}
catch (Overflow) { /* ... */ }
catch (Underflow) { /* ... */ }
catch (Zerodivide) { /* ... */ }
// ...
```

Это не только утомительно, но и опасно, поскольку можно забыть какую-нибудь особую ситуацию. Кроме того, необходимость перечислить в проверяемом блоке все особые ситуации практически гарантирует, что, когда семейство особых ситуаций библиотеки расширится, в программе пользователя возникнет ошибка. Это значит, что при введении новой особой ситуации в библиотеки математических функций придется перетранслировать все части программы, которые содержат обработчики всех особых ситуаций из `Matherr`. В общем случае такая перетрансляция неприемлема. Часто даже нет возможности найти все требующие перетрансляции части программы. Если такая возможность есть, нельзя требовать, чтобы всегда был доступен исходной текст любой части большой программы,

или чтобы у нас были права изменять любую часть большой программы, исходный текст которой мы имеем. На самом деле, пользователь не должен думать о внутреннем устройстве библиотек. Все эти проблемы перетрансляции и сопровождения могут привести к тому, что после создания первой версии библиотеки будет нельзя вводить в ней новые особые ситуации. Но такое решение не подходит практически для всех библиотек.

Все эти доводы говорят за то, что особые ситуации нужно определять как иерархию классов (см. также § 9.6.1). Это, в свою очередь, означает, что особые ситуации могут быть членами нескольких групп:

```
class network_file_err  // ошибки файловой системы в сети
  : public network_err, // ошибки сети
    public file_system_err { // ошибки файловой системы
    // ...
};
```

Особую ситуацию `network_file_err` можно перехватить в функциях, обрабатывающих особые ситуации сети:

```
void f()
{
  try {
      // какие-то операторы
  }
  catch (network_err) {
      // ...
  }
}
```

Ее также можно перехватить в функциях, обрабатывающих особые ситуации файловой системы:

```
void g()
{
  try {
      // какие-то другие операторы
  }
```

```
    catch (file_system_err) {
        // ...
    }
}
```

Это важный момент, поскольку такой системный сервис как работа в сети должен быть прозрачен, а это означает, что создатель функции `g()` может даже и не знать, что эта функция будет выполняться в сетевом режиме.

Отметим, что в настоящее время нет стандартного множества особых ситуаций для стандартной математической библиотеки и библиотеки ввода-вывода. Задача комитетов ANSI и ISO по стандартизации C++ решить нужно ли такое множество и какие в нем следует использовать имена и классы.

Поскольку можно сразу перехватить все особые ситуации (см. § 9.3.2), нет настоятельной необходимости создавать для этой цели общий, базовый для всех особых ситуаций, класс. Однако, если все особые ситуации являются производными от пустого класса `Exception` (особая ситуация), то в интерфейсах их использование становится более регулярным (см. § 9.6). Если вы используете общий базовый класс Exception, убедитесь, что в нем ничего нет кроме виртуального деструктора. В противном случае такой класс может вступить в противоречие с предполагаемым стандартом.

9.3.2 Производные особые ситуации

Если для обработки особых ситуаций мы используем иерархию классов, то, естественно, каждый обработчик должен разбираться только с частью информации, передаваемой при особых ситуациях. Можно сказать, что, как правило, особая ситуация перехватывается обработчиком ее базового класса, а не обработчиком класса, соответствующего именно этой особой ситуации. Именование и перехват обработчиком особой ситуации семантически эквивалентно именованию и получению параметра в функции. Проще говоря, формальный параметр инициализируется значением фактического

параметра. Это означает, что запущенная особая ситуация "низводится" до особой ситуации, ожидаемой обработчиком. Например:

```
class Matherr {
  // ...
  virtual void debug_print();
};

class Int_overflow : public Matherr {
public:
   char* op;
   int opr1, opr2;;
   Int_overflow(const char* p, int a, int b)
      { cerr << op << '(' << opr1 << ',' << opr2 << ')'; }
};

void f()
{
  try {
     g();
  }
  catch (Matherr m) {
     // ...
  }
}
```

При входе в обработчик `Matherr` особая ситуация `m` является объектом `Matherr`, даже если при обращении к `g()` была запущена `Int_overflow`. Это означает, что дополнительная информация, передаваемая в `Int_overflow`, недоступна.

Как обычно, чтобы иметь доступ к дополнительной информации, можно использовать указатели или ссылки. Поэтому можно было написать так:

```
int add(int x, int y)   // сложить x и y с контролем
{
  if (x > 0 && y > 0 && x > MAXINT - y
    || x < 0 && y < 0 && x < MININT + y)
```

```
    throw Int_overflow("+", x, y);

  // Сюда мы попадаем, либо когда проверка
  // на переполнение дала отрицательный результат,
  // либо когда x и y имеют разные знаки

  return x + y;
}

void f()
{
  try {
    add(1,2);
    add(MAXINT,-2);
    add(MAXINT,2);  // а дальше - переполнение
  }
  catch (Matherr& m) {
    // ...
    m.debug_print();
  }
}
```

Здесь последнее обращение к `add` приведет к запуску особой ситуации, который, в свою очередь, приведет к вызову `Int_overflow::debug_print()`. Если бы особая ситуация передавалась по значению, а не по ссылке, то была бы вызвана функция `Matherr::debug_print()`.

Нередко бывает так, что перехватив особую ситуацию, обработчик решает, что с этой ошибкой он ничего не сможет поделать. В таком случае самое естественное запустить особую ситуацию снова в надежде, что с ней сумеет разобраться другой обработчик:

```
void h()
{
  try {
    // какие-то операторы
  }
  catch (Matherr) {
```

```
    if (can_handle_it) { // если обработка возможна,
      // сделать ее
    }
    else {
      throw;  // повторный запуск перехваченной
              // особой ситуации
    }
  }
}
```

Повторный запуск записывается как оператор `throw` без параметров. При этом снова запускается исходная особая ситуация, которая была перехвачена, а не та ее часть, на которую рассчитан обработчик `Matherr`. Иными словами, если была запущена `Int_overflow`, вызывающая `h()` функция могла бы перехватить ее как `Int_overflow`, несмотря на то, что она была перехвачена в `h()` как `Matherr` и запущена снова:

```
void k()
{
  try {
    h();
    // ...
  }
  catch (Int_overflow) {
    // ...
  }
}
```

Полезен вырожденный случай перезапуска. Как и для функций, эллипсис `...` для обработчика означает "любой параметр", поэтому оператор `catch (...)` означает перехват любой особой ситуации:

```
void m()
{
  try {
    // какие-то операторы
  }
  catch (...) {
```

```
        // привести все в порядок
        throw;
    }

}
```

Этот пример надо понимать так: если при выполнении основной части `m()` возникает особая ситуация, выполняется обработчик, который выполняет общие действия по устранению последствий особой ситуации, после этих действий особая ситуация, вызвавшая их, запускается повторно.

Поскольку обработчик может перехватить производные особые ситуации нескольких типов, порядок, в котором идут обработчики в проверяемом блоке, существенен. Обработчики пытаются перехватить особую ситуацию в порядке их описания. Приведем пример:

```
try {
    // ...
}
catch (ibuf) {
    // обработка переполнения буфера ввода
}
catch (io) {
    // обработка любой ошибки ввода-вывода
}
catch (stdlib) {
    // обработка любой особой ситуации в библиотеке
}
catch (...) {
    // обработка всех остальных особых ситуаций
}
```

Тип особой ситуации в обработчике соответствует типу запущенной особой ситуации в следующих случаях: если эти типы совпадают, или второй тип является типом доступного базового класса запущенной ситуации, или он является указателем на такой класс, а тип ожидаемой ситуации тоже указатель (§ R.4.6).

Поскольку транслятору известна иерархия классов, он способен обнаружить такие нелепые ошибки, когда обработчик `catch (...)` указан не последним, или когда обработчик ситуации базового класса предшествует обработчику производной от этого класса ситуации (§ R15.4). В обоих случаях последующий обработчик (или обработчики) не могут быть запущены, поскольку они "маскируются" первым обработчиком.

9.4 Запросы ресурсов

Если в некоторой функции потребуются определенные ресурсы, например, нужно открыть файл, отвести блок памяти в области свободной памяти, установить монопольные права доступа и т.д., для дальнейшей работы системы обычно бывает крайне важно, чтобы ресурсы были освобождены надлежащим образом. Обычно такой "надлежащий способ" реализует функция, в которой происходит запрос ресурсов и освобождение их перед выходом. Например:

```
void use_file(const char* fn)
{
  FILE* f = fopen(fn,"w");

  // работаем с f

  fclose(f);
}
```

Все это выглядит вполне нормально до тех пор, пока вы не поймете, что при любой ошибке, происшедшей после вызова `fopen()` и до вызова `fclose()`, возникнет особая ситуация, в результате которой мы выйдем из `use_file()`, не обращаясь к `fclose()`.

Стоит сказать, что та же проблема возникает и в языках, не поддерживающих особые ситуации. Так, обращение к функции `longjump()` из стандартной библиотеки C может иметь такие же неприятные последствия.

Если вы создаете устойчивую к ошибкам систему, эту проблему

придется решать. Можно дать примитивное решение:

```
void use_file(const char* fn)
{
  FILE* f = fopen(fn,"w");
  try {
     // работаем с f
  }
  catch (...) {
     fclose(f);
     throw;
  }
  fclose(f);
}
```

Вся часть функции, работающая с файлом `f`, помещена в проверяемый блок, в котором перехватываются все особые ситуации, закрывается файл и особая ситуация запускается повторно.

Недостаток этого решения в его многословности, громоздкости и потенциальной расточительности. К тому же всякое многословное и громоздкое решение чревато ошибками, хотя бы в силу усталости программиста. К счастью, есть более приемлемое решение. В общем виде проблему можно сформулировать так:

```
void acquire()
{
  // запрос ресурса 1
  // ...
  // запрос ресурса n

  // использование ресурсов

  // освобождение ресурса n
  // ...
  // освобождение ресурса 1
}
```

Как правило бывает важно, чтобы ресурсы освобождались в обратном по сравнению с запросами порядке. Это очень сильно напоминает

порядок работы с локальными объектами, создаваемыми конструкторами и уничтожаемыми деструкторами. Поэтому мы можем решить проблему запроса и освобождения ресурсов, если будем использовать подходящие объекты классов с конструкторами и деструкторами. Например, можно определить класс `FilePtr`, который выступает как тип `FILE*`:

```
class FilePtr {
  FILE* p;
public:
  FilePtr(const char* n, const char* a)
    { p = fopen(n,a); }
  FilePtr(FILE* pp) { p = pp; }
  ~FilePtr() { fclose(p); }

  operator FILE*() { return p; }
};
```

Построить объект `FilePtr` можно либо, имея объект типа `FILE*`, либо, получив нужные для `fopen()` параметры. В любом случае этот объект будет уничтожен при выходе из его области видимости, и его деструктор закроет файл. Теперь наш пример сжимается до такой функции:

```
void use_file(const char* fn)
{
  FilePtr f(fn,"w");
  // работаем с f
}
```

Деструктор будет вызываться независимо от того, закончилась ли функция нормально, или произошел запуск особой ситуации.

9.4.1 Конструкторы и деструкторы

Описанный способ управления ресурсами обычно называют "запрос ресурсов путем инициализации". Это универсальный прием, рассчитанный на свойства конструкторов и деструкторов и их

взаимодействие с механизмом особых ситуаций.

Объект не считается построенным, пока не завершил выполнение его конструктор. Только после этого возможна раскрутка стека, сопровождающая вызов деструктора объекта. Объект, состоящий из вложенных объектов, построен в той степени, в какой построены вложенные объекты.

Хорошо написанный конструктор должен гарантировать, что объект построен полностью и правильно. Если ему не удается сделать это, он должен, насколько это возможно, восстановить состояние системы, которое было до начала построения. Для простых конструкторов было бы идеально всегда удовлетворять хотя бы одному условию - правильности или законченности объектов, и никогда не оставлять объект в "наполовину построенном" состоянии. Этого можно добиться, если применять при построении членов способ "запроса ресурсов путем инициализации".

Рассмотрим класс `X`, конструктору которого требуется два ресурса: файл `x` и замок `y` (т.е. монопольные права доступа к чему-либо). Эти запросы могут быть отклонены и привести к запуску особой ситуации. Чтобы не усложнять работу программиста, можно потребовать, чтобы конструктор класса `X` никогда не завершался тем, что запрос на файл удовлетворен, а на замок нет. Для представления двух видов ресурсов мы будем использовать объекты двух классов `FilePtr` и `LockPtr` (естественно, было бы достаточно одного класса, если `x` и `y` ресурсы одного вида). Запрос ресурса выглядит как инициализация представляющего ресурс объекта:

```
class X {
  FilePtr aa;
  LockPtr bb;
  // ...
  X(const char* x, const char* y)
    : aa(x),      // запрос `x'
    bb(y)         // запрос `y'
  { }
  // ...
};
```

Теперь, как это было для случая локальных объектов, всю служебную работу, связанную с ресурсами, можно возложить на реализацию. Пользователь не обязан следить за ходом такой работы. Например, если после построения `aa` и до построения `bb` возникнет особая ситуация, то будет вызван только деструктор aa, но не `bb`.

Это означает, что если строго придерживаться этой простой схемы запроса ресурсов, то все будет в порядке. Еще более важно то, что создателю конструктора не нужно самому писать обработчики особых ситуаций.

Для требований выделить блок в свободной памяти характерен самый произвольный порядок запроса ресурсов. Примеры таких запросов уже неоднократно встречались в этой книге:

```
class X {
  int* p;
  // ...
public:
  X(int s) { p = new int[s]; init(); }
  ~X() { delete[] p; }
  // ...
};
```

Это типичный пример использования свободной памяти, но в совокупности с особыми ситуациями он может привести к ее исчерпанию. Действительно, если в `init()` запущена особая ситуация, то отведенная память не будет освобождена. Деструктор не будет вызываться, поскольку построение объекта не было завершено. Есть более надежный вариант этого примера:

```
template<class T> class MemPtr {
public:
  T* p;
  MemPtr(size_t s) { p = new T[s]; }
  ~MemPtr() { delete[] p; }
  operator T*() { return p; }
}
```

```
class X {
  MemPtr<int> cp;
  // ...
public:
  X(int s):cp(s) { init(); }
  // ...
};
```

Теперь уничтожение массива, на который указывает `p`, происходит неявно в `MemPtr`. Если `init()` запустит особую ситуацию, отведенная память будет освобождена при неявном вызове деструктора для полностью построенного вложенного объекта `cp`.

Отметим также, что стандартная стратегия выделения памяти в C++ гарантирует, что если функции `operator new()` не удалось выделить память для объекта, то конструктор для него никогда не будет вызываться. Это означает, что пользователю не надо опасаться, что конструктор или деструктор может быть вызван для несуществующего объекта.

Теоретически дополнительные расходы, требующиеся для обработки особых ситуаций, когда на самом деле ни одна из них не возникла, могут быть сведены к нулю. Однако, вряд ли это верно для ранних реализаций языка. Поэтому будет разумно в критичных внутренних циклах программы пока не использовать локальные переменные классов с деструкторами.

9.4.2 Предостережения

Не все программы должны быть устойчивы ко всем видам ошибок. Не все ресурсы являются настолько критичными, чтобы оправдать попытки защитить их с помощью описанного способа "запроса ресурсов путем инициализации". Есть множество программ, которые просто читают входные данные и выполняются до конца. Для них самой подходящей реакцией на динамическую ошибку будет просто прекращение счета (после выдачи соответствующего сообщения). Освобождение всех затребованных ресурсов возлагается на систему, а пользователь должен произвести повторный запуск программы с более

подходящими входными данными. Наша схема предназначена для задач, в которых такая примитивная реакция на динамическую ошибку неприемлема. Например, разработчик библиотеки обычно не в праве делать допущения о том, насколько устойчива к ошибкам, должна быть программа, работающая с библиотекой. Поэтому он должен учитывать все динамические ошибки и освобождать все ресурсы до возврата из библиотечной функции в пользовательскую программу. Метод "запроса ресурсов путем инициализации" в совокупности с особыми ситуациями, сигнализирующими об ошибке, может пригодиться при создании многих библиотек.

9.4.3 Исчерпание ресурса

Есть одна из вечных проблем программирования: что делать, если не удалось удовлетворить запрос на ресурс? Например, в предыдущем примере мы спокойно открывали с помощью `fopen()` файлы и запрашивали с помощью операции `new` блок свободной памяти, не задумываясь при этом, что такого файла может не быть, а свободная память может исчерпаться. Для решения такого рода проблем у программистов есть два способа:

- Повторный запрос: пользователь должен изменить свой запрос и повторить его.
- Завершение: запросить дополнительные ресурсы от системы, если их нет, запустить особую ситуацию.

Первый способ предполагает для задания приемлемого запроса содействие пользователя, во втором пользователь должен быть готов правильно отреагировать на отказ в выделении ресурсов. В большинстве случаев последний способ намного проще и позволяет поддерживать в системе разделение различных уровней абстракции.

В C++ первый способ поддержан механизмом вызова функций, а второй - механизмом особых ситуаций. Оба способа можно продемонстрировать на примере реализации и использования операции new:

```
#include <stdlib.h>
```

```
extern void* _last_allocation;

extern void* operator new(size_t size)
{
  void* p;

  while ( (p=malloc(size))==0 ) {
    if (_new_handler)
      (*_new_handler)();  // обратимся за помощью
    else
      return 0;
  }
  return _last_allocation=p;
}
```

Если операция `new()` не может найти свободной памяти, она обращается к управляющей функции `_new_handler()`. Если в `_new_handler()` можно выделить достаточный объем памяти, все нормально. Если нет, из управляющей функции нельзя возвратиться в операцию `new`, т.к. возникнет бесконечный цикл. Поэтому управляющая функция может запустить особую ситуацию и предоставить исправлять положение программе, обратившейся к `new`:

```
void my_new_handler()
{
  try_find_some_memory();   // попытаемся найти
          // свободную память
  if (found_some()) return; // если она найдена, все в порядке
  throw Memory_exhausted(); // иначе запускаем особую
                    // ситуацию "Исчерпание_памяти"
}
```

Где-то в программе должен быть проверяемый блок с соответствующим обработчиком:

```
try {
    // ...
}
```

```
catch (Memory_exhausted) {
    // ...
}
```

В функции `operator new()` использовался указатель на управляющую функцию `_new_handler`, который настраивается стандартной функцией `set_new_handler()`. Если нужно настроиться на собственную управляющую функцию, надо обратиться так

```
set_new_handler(&my_new_handler);
```

Перехватить ситуацию `Memory_exhausted` можно следующим образом:

```
void (*oldnh)() = set_new_handler(&my_new_handler);

try {
    // ...
}
catch (Memory_exhausted) {
    // ...
}
catch (...) {
    set_new_handler(oldnh);  // восстановить указатель на
            // управляющую функцию
    throw();      // повторный запуск особой ситуации
}

set_new_handler(oldnh); // восстановить указатель на
                    // управляющую функцию
```

Можно поступить еще лучше, если к управляющей функции применить описанный в § 9.4 метод "запроса ресурсов путем инициализации" и убрать обработчик `catch (...)`.

В решении, использующим `my_new_handler()`, от точки обнаружения ошибки до функции, в которой она обрабатывается, не передается никакой информации. Если нужно передать какие-то

данные, то пользователь может включить свою управляющую функцию в класс. Тогда в функции, обнаружившей ошибку, нужные данные можно поместить в объект этого класса. Подобный способ, использующий объекты-функции, применялся в § 10.4.2 для реализации манипуляторов. Способ, в котором используется указатель на функцию или объект-функция для того, чтобы из управляющей функции, обслуживающей некоторый ресурс, произвести "обратный вызов" функции, запросившей этот ресурс, обычно называется просто обратным вызовом (callback).

При этом нужно понимать, что чем больше информации передается из обнаружившей ошибку функции в функцию, пытающуюся ее исправить, тем больше зависимость между этими двумя функциями. В общем случае лучше сводить к минимуму такие зависимости, поскольку всякое изменение в одной из функций придется делать с учетом другой функции, а, возможно, ее тоже придется изменять. Вообще, лучше не смешивать отдельные компоненты программы. Механизм особых ситуаций позволяет сохранять раздельность компонентов лучше, чем обычный механизм вызова управляющих функций, которые задает функция, затребовавшая ресурс.

В общем случае разумный подход состоит в том, чтобы выделение ресурсов было многоуровневым (в соответствии с уровнями абстракции). При этом нужно избегать того, чтобы функции одного уровня зависели от управляющей функции, вызываемой на другом уровне. Опыт создания больших программных систем показывает, что со временем удачные системы развиваются именно в этом направлении.

9.4.4 Особые ситуации и конструкторы

Особые ситуации дают средство сигнализировать о происходящих в конструкторе ошибках. Поскольку конструктор не возвращает такое значение, которое могла бы проверить вызывающая функция, есть следующие обычные (т.е. не использующие особые ситуации) способы сигнализации:

1. Возвратить объект в ненормальном состоянии в расчете, что

пользователь проверит его состояние.

2. Установить значение нелокальной переменной, которое сигнализирует, что создать объект не удалось.

Особые ситуации позволяют тот факт, что создать объект не удалось, передать из конструктора вовне:

```
Vector::Vector(int size)
{
  if (sz<0 || max<sz) throw Size();
  // ...
}
```

В функции, создающей вектора, можно перехватить ошибки, вызванные недопустимым размером (`Size()`) и попытаться на них отреагировать:

```
Vector* f(int i)
{
 Vector* p;
 try {
  p = new Vector v(i);
 }
 catch (Vector::Size) {
  // реакция на недопустимый размер вектора
 }
 // ...
 return p;
}
```

Управляющая созданием вектора функция способна правильно отреагировать на ошибку. В самом обработчике особой ситуации можно применить какой-нибудь из стандартных способов диагностики и восстановления после ошибки. При каждом перехвате особой ситуации в управляющей функции может быть свой взгляд на причину ошибки. Если с каждой особой ситуацией передаются описывающие ее данные, то объем данных, которые нужно анализировать для каждой ошибки, растет. Основная задача обработки ошибок в том, чтобы обеспечить надежный и удобный способ передачи данных от исходной точки

обнаружения ошибки до того места, где после нее возможно осмысленное восстановление.

Способ "запроса ресурсов путем инициализации" - самое надежное и красивое решение в том случае, когда имеются конструкторы, требующие более одного ресурса. По сути он позволяет свести задачу выделения нескольких ресурсов к повторно применяемому, более простому, способу, рассчитанному на один ресурс.

9.5 Особые ситуации могут не быть ошибками

Если особая ситуация ожидалась, была перехвачена и не оказала плохого воздействия на ход программы, то стоит ли ее называть ошибкой? Так говорят только потому, что программист думает о ней как об ошибке, а механизм особых ситуаций является средством обработки ошибок. С другой стороны, особые ситуации можно рассматривать просто как еще одну структуру управления. Подтвердим это примером:

```
class message { /* ... */ };   // сообщение

class queue {        // очередь
  // ...
  message* get();   // вернуть 0, если очередь пуста
  // ...
};

void f1(queue& q)
{
 message* m = q.get();
 if (m == 0) {  // очередь пуста
    // ...
 }
 // используем m
}
```

Этот пример можно записать так:

```
class Empty { }  // тип особой ситуации "Пустая_очередь"
```

```
class queue {
    // ...
    message* get();  // запустить Empty, если очередь пуста
    // ...
};

void f2(queue& q)
{
  try {
    message* m = q.get();
    // используем m
  }
  catch (Empty) {  // очередь пуста
    // ...
  }
}
```

В варианте с особой ситуацией есть даже какая-то прелесть. Это хороший пример того, когда трудно сказать, можно ли считать такую ситуацию ошибкой. Если очередь не должна быть пустой (т.е. она бывает пустой очень редко, скажем один раз из тысячи), и действия в случае пустой очереди можно рассматривать как восстановление, то в функции `f2()` взгляд на особую ситуацию будет такой, которого мы до сих пор и придерживались (т.е. обработка особых ситуаций есть обработка ошибок). Если очередь часто бывает пустой, а принимаемые в этом случае действия образуют одну из ветвей нормального хода программы, то придется отказаться от такого взгляда на особую ситуацию, а функцию `f2()` надо переписать:

```
class queue {
  // ...
  message* get();  // запустить Empty, если очередь пуста
  int empty();
  // ...
};

void f3(queue& q)
{
  if (q.empty()) { // очередь пуста
```

```
    // ...
  }
  else {
    message* m = q.get();
    // используем m
  }
}
```

Отметим, что вынести из функции `get()` проверку очереди на пустоту можно только при условии, что к очереди нет параллельных обращений.

Не так то просто отказаться от взгляда, что обработка особой ситуации есть обработка ошибки. Пока мы придерживаемся такой точки зрения, программа четко подразделяется на две части: обычная часть и часть обработки ошибок. Такая программа более понятна. К сожалению, в реальных задачах провести четкое разделение невозможно, поэтому структура программы должна (и будет) отражать этот факт. Допустим, очередь бывает пустой только один раз (так может быть, если функция `get()` используется в цикле, и пустота очереди говорит о конце цикла). Тогда пустота очереди не является чем-то странным или ошибочным. Поэтому, используя для обозначения конца очереди особую ситуацию, мы расширяем представление об особых ситуациях как ошибках. С другой стороны, действия, принимаемые в случае пустой очереди, явно отличаются от действий, принимаемых в ходе цикла (т.е. в обычном случае).

Механизм особых ситуаций является менее структурированным, чем такие локальные структуры управления как операторы `if` или `for`. Обычно он к тому же является не столь эффективным, если особая ситуация действительно возникла. Поэтому особые ситуации следует использовать только в том случае, когда нет хорошего решения с более традиционными управляющими структурами, или оно, вообще, невозможно. Например, в случае пустой очереди можно прекрасно использовать для сигнализации об этом значение, а именно нулевое значение указателя на строку message, значит особая ситуация здесь не нужна. Однако, если бы из класса queue мы получали вместо указателя значение типа `int`, то могло не найтись такого значения, обозначающего пустую очередь. В таком случае функция `get()`

становится эквивалентной операции индексации из § 9.1, и более привлекательно представлять пустую очередь с помощью особой ситуации. Последнее соображение подсказывает, что в самом общем шаблоне типа для очереди придется для обозначения пустой очереди использовать особую ситуацию, а работающая с очередью функция будет такой:

```
void f(Queue<X>& q)
{
  try {
     for (;;) {  // ``бесконечный цикл"
   // прерываемый особой ситуацией
          X m = q.get();
          // ...
     }
  }
  catch (Queue<X>::Empty) {
       return;
  }
}
```

Если приведенный цикл выполняется тысячи раз, то он, по всей видимости, будет более эффективным, чем обычный цикл с проверкой условия пустоты очереди. Если же он выполняется только несколько раз, то обычный цикл почти наверняка эффективней.

В очереди общего вида особая ситуация используется как способ возврата из функции `get()`. Использование особых ситуаций как способа возврата может быть элегантным способом завершения функций поиска. Особенно это подходит для рекурсивных функций поиска в дереве. Однако, применяя особые ситуации для таких целей, легко перейти грань разумного и получить маловразумительную программу. Все-таки всюду, где это действительно оправдано, надо придерживаться той точки зрения, что обработка особой ситуации есть обработка ошибки. Обработка ошибок по самой своей природе занятие сложное, поэтому ценность имеют любые методы, которые дают ясное представление ошибок в языке и способ их обработки.

9.6 Задание интерфейса

Запуск или перехват особой ситуации отражается на взаимоотношениях функций. Поэтому имеет смысл задавать в описании функции множество особых ситуаций, которые она может запустить:

```
void f(int a) throw (x2, x3, x4);
```

В этом описании указано, что `f()` может запустить особые ситуации `x2`, `x3` и `x4`, а также ситуации всех производных от них типов, но больше никакие ситуации она не запускает. Если функция перечисляет свои особые ситуации, то она дает определенную гарантию всякой вызывающей ее функции, а именно, если попытается запустить иную особую ситуацию, то это приведет к вызову функции `unexpected()`.

Стандартное предназначение `unexpected()` состоит в вызове функции `terminate()`, которая, в свою очередь, обычно вызывает `abort()`. Подробности даны в § 9.7.

По сути определение

```
void f() throw (x2, x3, x4)
{
  // какие-то операторы
}
```

эквивалентно такому определению

```
void f()
{
  try {
    // какие-то операторы
  }
  catch (x2) {  // повторный запуск
   throw;
  }
  catch (x3) {  // повторный запуск
   throw;
  }
  catch (x4) {  // повторный запуск
   throw;
```

```
    }
    catch (...) {
     unexpected();
    }
  }
```

Преимущество явного задания особых ситуаций функции в ее описании перед эквивалентным способом, когда происходит проверка на особые ситуации в теле функции, не только в более краткой записи. Главное здесь в том, что описание функции входит в ее интерфейс, который видим для всех вызывающих функций. С другой стороны, определение функции может и не быть универсально доступным. Даже если у вас есть исходные тексты всех библиотечных функций, обычно желание изучать их возникает не часто.

Если в описании функции не указаны ее особые ситуации, считается, что она может запустить любую особую ситуацию.

```
int f();  // может запустить любую особую ситуацию
```

Если функция не будет запускать никаких особых ситуаций, ее можно описать, явно указав пустой список:

```
int g() throw (); // не запускает никаких особых ситуаций
```

Казалось было бы логично, чтобы по умолчанию функция не запускала никаких особых ситуаций. Но тогда пришлось бы описывать свои особые ситуации практически для каждой функции Это, как правило, требовало бы ее перетрансляции, а кроме того препятствовало бы общению с функциями, написанными на других языках. В результате программист стал бы стремиться отключить механизм особых ситуаций и писал бы излишние операторы, чтобы обойти их. Пользователь считал бы такие программы надежными, поскольку мог не заметить подмены, но это было бы совершенно неоправдано.

9.6.1 Неожиданные особые ситуации

Если к описанию особых ситуаций относиться не достаточно серьезно,

то результатом может быть вызов `unexpected()`, что нежелательно во всех случаях, кроме отладки. Избежать вызова `unexpected()` можно, если хорошо организовать структуру особых ситуаций и описание интерфейса. С другой стороны, вызов `unexpected()` можно перехватить и сделать его безвредным.

Если компонент `Y` хорошо разработан, все его особые ситуации могут быть только производными одного класса, скажем `Yerr`. Поэтому, если есть описание

```
class someYerr : public Yerr { /* ... */ };
```

то функция, описанная как

```
void f() throw (Xerr, Yerr, IOerr);
```

будет передавать любую особую ситуацию типа `Yerr` вызывающей функции. В частности, обработка особой ситуации типа `someYerr` в `f()` сведется к передаче ее вызывающей `f()` функции.

Бывают случаи, когда окончание программы при появлении неожиданной особой ситуации является слишком строгим решением. Допустим функция `g()` написана для несетевого режима в распределенной системе. Естественно, в `g()` ничего неизвестно об особых ситуациях, связанных с сетью, поэтому при появлении любой из них вызывается `unexpected()`. Значит для использования `g()` в распределенной системе нужно предоставить обработчик сетевых особых ситуаций или переписать `g()`. Если допустить, что переписать `g()` невозможно или нежелательно, проблему можно решить, переопределив действие функции `unexpected()`. Для этого служит функция `set_unexpected()`. Вначале мы определим класс, который позволит нам применить для функций `unexpected()` метод "запроса ресурсов путем инициализации":

```
typedef void(*PFV)();
PFV set_unexpected(PFV);

class STC {   // класс для сохранения и восстановления
  PFV old;    // функций unexpected()
```

```
public:
  STC(PFV f) { old = set_unexpected(f); }
  ~STC() { set_unexpected(old); }
};
```

Теперь мы определим функцию, которая должна в нашем примере заменить `unexpected()`:

```
void rethrow() { throw; }  // перезапуск всех сетевых
                           // особых ситуаций
```

Наконец, можно дать вариант функции `g()`, предназначенный для работы в сетевом режиме:

```
void networked_g()
{
 STC xx(&rethrow); // теперь unexpected() вызывает rethrow()
 g();
}
```

В предыдущем разделе было показано, что `unexpected()` потенциально вызывается из обработчика `catch (...)`. Значит в нашем случае обязательно произойдет повторный запуск особой ситуации. Повторный запуск, когда особая ситуация не запускалась, приводит к вызову `terminate()`. Поскольку обработчик `catch (...)` находится вне той области видимости, в которой была запущена сетевая особая ситуация, бесконечный цикл возникнуть не может.

Есть еще одно, довольно опасное, решение, когда на неожиданную особую ситуацию просто "закрывают глаза":

```
void muddle_on() { cerr << "не замечаем особой ситуации\n"; }
// ...
STC xx(&muddle_on);  // теперь действие unexpected() сводится
                     // просто к печати сообщения
```

Такое переопределение действия `unexpected()` позволяет нормально вернуться из функции, обнаружившей неожиданную особую ситуацию. Несмотря на свою очевидную опасность, это решение используется. Например, можно "закрыть глаза" на особые ситуации в

одной части системы и отлаживать другие ее части. Такой подход может быть полезен в процессе отладки и развития системы, перенесенной с языка программирования без особых ситуаций. Все-таки, как правило лучше, если ошибки проявляются как можно раньше.

Возможно другое решение, когда вызов `unexpected()` преобразуется в запуск особой ситуации `Fail` (неудача):

```
void fail() { throw Fail; }
// ...
STC yy(&fail);
```

При таком решении вызывающая функция не должна подробно разбираться в возможном результате вызываемой функции: эта функции завершится либо успешно (т.е. возвратится нормально), либо неудачно (т.е. запустит `Fail`). Очевидный недостаток этого решения в том, что не учитывается дополнительная информация, которая может сопровождать особую ситуацию. Впрочем, при необходимости ее можно учесть, если передавать информацию вместе с `Fail`.

9.7 Неперехваченные особые ситуации

Если особая ситуация запущена и не перехвачена, то вызывается функция `terminate()`. Она же вызывается, когда система поддержки особых ситуаций обнаруживает, что структура стека нарушена, или когда в процессе обработки особой ситуации при раскручивании стека вызывается деструктор, и он пытается завершить свою работу, запустив особую ситуацию.

Действие `terminate()` сводится к выполнению самой последней функции, заданной как параметр для `set_terminate()`:

```
typedef void (*PFV)();
        PFV set_terminate(PFV);
```

Функция `set_terminate()` возвращает указатель на ту функцию, которая была задана как параметр в предыдущем обращении к ней.

Необходимость такой функции как `terminate()` объясняется тем, что

иногда вместо механизма особых ситуаций требуются более грубые приемы. Например, `terminate()` можно использовать для прекращения процесса, а, возможно, и для повторного запуска системы. Эта функция служит экстренным средством, которое применяется, когда отказала стратегия обработки ошибок, рассчитанная на особые ситуации, и самое время применить стратегию более низкого уровня.

Функция `unexpected()` используется в сходных, но не столь серьезных случаях, а именно, когда функция запустила особую ситуацию, не указанную в ее описании. Действие функции `unexpected()` сводится к выполнению самой последней функции, заданной как параметр для функции `set_unexpected()`.

По умолчанию `unexpected()` вызывает `terminate()`, а та, в свою очередь, вызывает функцию `abort()`. Предполагается, что такое соглашение устроит большинство пользователей.

Предполагается, что функция `terminate()` не возвращается в обратившуюся к ней функцию.

Напомним, что вызов `abort()` свидетельствует о ненормальном завершении программы. Для нормального выхода из программы используется функция `exit()`. Она возвращает значение, которое показывает окружающей системе насколько корректно закончилась программа.

9.8 Другие способы обработки ошибок

Механизм особых ситуаций нужен для того, чтобы из одной части программы можно было сообщить в другую о возникновении в первой "особой ситуации". При этом предполагается, что части программы написаны независимо друг от друга, и в той части, которая обрабатывает особую ситуацию, возможна осмысленная реакция на ошибку.

Как же должен быть устроен обработчик особой ситуации? Приведем несколько вариантов:

```
int f(int arg)
```

```
{
  try {
    g(arg);
  }
  catch (x1) {
    // исправить ошибку и повторить
    g(arg);
  }
  catch (x2) {
    // произвести вычисления и вернуть результат
    return 2;
  }
  catch (x3) {
    // передать ошибку
    throw;
  }
  catch (x4) {
    // вместо x4 запустить другую особую ситуацию
    throw xxii;
  }
  catch (x5) {
    // исправить ошибку и продолжить со следующего оператора
  }
  catch (...) {
    // отказ от обработки ошибки
    terminate();
  }
  // ...
}
```

Укажем, что в обработчике доступны переменные из области видимости, содержащей проверяемый блок этого обработчика. Переменные, описанные в других обработчиках или других проверяемых блоках, конечно, недоступны:

```
void f()
{
  int i1;
  // ...
```

```
    try {
      int i2;
      // ...
    }
    catch (x1) {
      int i3;
      // ...
    }
    catch (x4) {
      i1 = 1;   // нормально
      i2 = 2;   // ошибка: i2 здесь невидимо
      i3 = 3;   // ошибка: i3 здесь невидимо
    }
  }
```

Нужна общая стратегия для эффективного использования обработчиков в программе. Все компоненты программы должны согласованно использовать особые ситуации и иметь общую часть для обработки ошибок. Механизм обработки особых ситуаций является нелокальным по своей сути, поэтому так важно придерживаться общей стратегии. Это предполагает, что стратегия обработки ошибок должна разрабатываться на самых ранних стадиях проектов. Кроме того, эта стратегия должна быть простой (по сравнению со сложностью всей программы) и ясной. Последовательно проводить сложную стратегию в такой сложной по своей природе области программирования, как восстановление после ошибок, будет просто невозможно.

Прежде всего стоит сразу отказаться от того, что одно средство или один прием можно применять для обработки всех ошибок. Это только усложнит систему. Удачная система, обладающая устойчивостью к ошибкам, должна строиться как многоуровневая. На каждом уровне надо обрабатывать настолько много ошибок, насколько это возможно без нарушения структуры системы, оставляя обработку других ошибок более высоким уровням. Назначение `terminate()` поддержать такой подход, предоставляя возможность экстренного выхода из такого положения, когда нарушен сам механизм обработки особых ситуаций, или когда он используется полностью, но особая ситуация оказалась неперехваченной. Функция `unexpected()` предназначена для выхода из такого положения, когда не сработало основанное на описании всех

особых ситуаций средство защиты. Это средство можно представлять как брандмауер, т.е. стену, окружающую каждую функцию, и препятствующую распространению ошибки. Попытка проводить в каждой функции полный контроль, чтобы иметь гарантию, что функция либо успешно завершится, либо закончится неудачно, но одним из определенных и корректных способов, не может принести успех. Причины этого могут быть различными для разных программ, но для больших программ можно назвать следующие:

1. работа, которую нужно провести, чтобы гарантировать надежность каждой функции, слишком велика, и поэтому ее не удастся провести достаточно последовательно;
2. появятся слишком большие дополнительные расходы памяти и времени, которые будут недопустимы для нормальной работы системы (будет тенденция неоднократно проверять на одну и ту же ошибку, а значит постоянно будут проверяться переменные с правильными значениями);
3. таким ограничениям не будут подчиняться функции, написанные на других языках;
4. такое понятие надежности является чисто локальным и оно настолько усложняет систему, что становится дополнительной нагрузкой для ее общей надежности.

Однако, разбить программу на отдельные подсистемы, которые либо успешно завершаются, либо заканчиваются неудачно, но одним из определенных и корректных способов, вполне возможно, важно и даже выгодно. Таким свойством должны обладать основные библиотеки, подсистемы или ключевые функции. Описание особых ситуаций должно входить в интерфейсы таких библиотек или подсистем.

Иногда приходится от одного стиля реакции на ошибку переходить на другой. Например, можно после вызова стандартной функции C проверять значение errno и, возможно, запускать особую ситуацию, а можно, наоборот, перехватывать особую ситуацию и устанавливать значение errno перед выходом из стандартной функции в C-программу:

```
void callC()
{
  errno = 0;
```

```
  cfunction();
  if (errno) throw some_exception(errno);
}

void fromC()
{
  try {
    c_pl_pl_function();
  }
  catch (...) {
    errno = E_CPLPLFCTBLEWIT;
  }
}
```

При такой смене стилей важно быть последовательным, чтобы изменение реакции на ошибку было полным.

Обработка ошибок должна быть, насколько это возможно, строго иерархической системой. Если в функции обнаружена динамическая ошибка, то не нужно обращаться за помощью для восстановления или выделения ресурсов к вызывающей функции. При таких обращениях в структуре системы возникают циклические зависимости, в результате чего ее труднее понять, и возможно возникновение бесконечных циклов в процессе обработки и восстановления после ошибки.

Чтобы часть программы, предназначенная для обработки ошибок была более упорядоченной, стоит применять такие упрощающие дело приемы, как "запрос ресурсов путем инициализации", и исходить из таких упрощающих дело допущений, что "особые ситуации являются ошибками".

9.9 Упражнения

1. (*2) Обобщите класс `STC` до шаблона типа, который позволяет хранить и устанавливать функции разных типов.
2. (*3) Дополните класс `CheckedPtrToT` из § 7.10 до шаблона типа, в котором особые ситуации сигнализируют о динамических ошибках.

3. (*3) Напишите функцию `find` для поиска в бинарном дереве узлов по значению поля типа `char*`. Если найден узел с полем, имеющим значение "hello", она должна возвращать указатель на него. Для обозначения неудачного поиска используйте особую ситуацию.
4. (*1) Определите класс `Int`, совпадающий во всем со встроенным типом `int` за исключением того, что вместо переполнения или потери значимости в этом классе запускаются особые ситуации. Подсказка: см. § 9.3.2.
5. (*2) Перенесите из стандартного интерфейса C в вашу операционную систему основные операции с файлами: открытие, закрытие, чтение и запись. Реализуйте их как функции на C++ с тем же назначением, что и функций на C, но в случае ошибок запускайте особые ситуации.
6. (*1) Напишите полное определение шаблона типа `Vector` с особыми ситуациями `Range` и `Size`. Подсказка: см. § 9.3.
7. (*1) Напишите цикл для вычисления суммы элементов вектора, определенного в упражнении 6, причем не проверяйте размер вектора. Почему это плохое решение?
8. (*2.5) Допустим класс `Exception` используется как базовый для всех классов, задающих особые ситуации. Каков должен быть его вид? Какая от него могла быть польза? Какие неудобства может вызвать требование обязательного использования этого класса?
9. (*2) Напишите класс или шаблон типа, который поможет реализовать обратный вызов.
10. (*2) Напишите класс `Lock` (замок) для какой-нибудь системы, допускающей параллельное выполнение.
11. (*1) Пусть определена функция

```
int main() { /* ... */ }
```

Измените ее так, чтобы в ней перехватывались все особые ситуации, преобразовывались в сообщения об ошибке и вызов `abort()`. Подсказка: в функции `fromC()` из § 9.8 учтены не все случаи.

Потоки

В языке C++ нет средств для ввода-вывода. Их и не нужно, поскольку такие средства можно просто и элегантно создать на самом языке. Описанная здесь библиотека потокового ввода-вывода реализует строгий типовой и вместе с тем гибкий и эффективный способ символьного ввода и вывода целых, вещественных чисел и символьных строк, а также является базой для расширения, рассчитанного на работу с пользовательскими типами данных. Пользовательский интерфейс библиотеки находится в файле <iostream.h>. Эта лекция посвящена самой потоковой библиотеке, некоторым способам работы с ней и определенным приемам реализации библиотеки.

10.1 Введение

"Доступно только то, что видимо"

Б. Керниган

Широко известна трудность задачи проектирования и реализации стандартных средств ввода-вывода для языков программирования. Традиционно средства ввода-вывода были рассчитаны исключительно на небольшое число встроенных типов данных. Однако, в нетривиальных программах на C++ есть много пользовательских типов данных, поэтому необходимо предоставить возможность ввода-вывода значений таких типов. Очевидно, что средства ввода-вывода должны быть простыми, удобными, надежными в использовании и, что важнее всего, адекватными. Пока никто не нашел решения, которое удовлетворило бы всех; поэтому необходимо дать возможность пользователю создавать иные средства ввода-вывода, а также расширять стандартные средства ввода-вывода в расчете на определенное применение.

Цель создания C++ была в том, чтобы пользователь мог определить новые типы данных, работа с которыми была бы столь же удобна и эффективна как и со встроенными типами. Таким образом, кажется разумным потребовать, чтобы средства ввода-вывода для C++ программировались с использованием возможностей C++, доступных

каждому. Представленные здесь потоковые средства ввода-вывода появились в результате попытки удовлетворить этим требованиям.

Основная задача потоковых средств ввода-вывода - это процесс преобразования объектов определенного типа в последовательность символов и наоборот. Существуют и другие схемы ввода-вывода, но указанная является основной, и если считать символ просто набором битов, игнорируя его естественную связь с алфавитом, то многие схемы двоичного ввода-вывода можно свести к ней. Поэтому программистская суть задачи сводится к описанию связи между объектом определенного типа и бестиповой (что существенно) строкой.

Последующие разделы описывают основные части потоковой библиотеки C++:

10.2 Вывод: То, что для прикладной программы представляется выводом, на самом деле является преобразованием таких объектов как `int`, `char *`, `complex` или `Employee_record` в последовательность символов. Описываются средства для записи объектов встроенных и пользовательских типов данных.

10.3 Ввод: Описаны функции для ввода символов, строк и значений встроенных и пользовательских типов данных.

10.4 Форматирование: Часто существуют определенные требования к виду вывода, например, `int` должно печататься десятичными цифрами, указатели в шестнадцатеричной записи, а вещественные числа должны быть с явно заданной точностью фиксированного размера. Обсуждаются функции форматирования и определенные программистские приемы их создания, в частности, манипуляторы.

10.5 Файлы и потоки: Каждая программа на C++ может использовать по умолчанию три потока - стандартный вывод (cout), стандартный ввод (cin) и стандартный поток ошибок (cerr). Чтобы работать с какими-либо устройствами или файлами надо создать потоки и привязать их к этим устройствам или файлам. Описывается механизм открытия и закрытия файлов и связывания файлов с потоками.

10.6 Ввод-вывод для C: обсуждается функция `printf` из файла <stdio.h> для C а также связь между библиотекой для C и <iostream.h>

для C++.

Укажем, что существует много независимых реализаций потоковой библиотеки ввода-вывода и набор средств, описанных здесь, будет только подмножеством средств, имеющихся в вашей библиотеке. Говорят, что внутри любой большой программы есть маленькая программа, которая стремится вырваться наружу. В этой лекции предпринята попытка описать как раз маленькую потоковую библиотеку ввода-вывода, которая позволит оценить основные концепции потокового ввода-вывода и познакомить с наиболее полезными средствами. Используя только средства, описанные здесь, можно написать много программ; если возникнет необходимость в более сложных средствах, обратитесь за деталями к вашему руководству по C ++. Заголовочный файл <iostream.h> определяет интерфейс потоковой библиотеки. В ранних версиях потоковой библиотеки использовался файл <stream.h>. Если существуют оба файла, <iostream.h> определяет полный набор средств, а <stream.h> определяет подмножество, которое совместимо с ранними, менее богатыми потоковыми библиотеками.

Естественно, для пользования потоковой библиотекой вовсе не нужно знание техники ее реализации, тем более, что техника может быть различной для различных реализаций. Однако, реализация ввода-вывода является задачей, диктующей определенные условия, значит приемы, найденные в процессе ее решения, можно применить и для других задач, а само это решение достойно изучения.

10.2 Вывод

Строгую типовую и единообразную работу как со встроенными, так и с пользовательскими типами можно обеспечить, если использовать единственное перегруженное имя функции для различных операций вывода.

Например:

```
put(cerr,"x = "); // cerr - выходной поток ошибок
put(cerr,x);
put(cerr,'\n');
```

Тип аргумента определяет какую функцию надо вызывать в каждом случае. Такой подход применяется в нескольких языках, однако, это слишком длинная запись. За счет перегрузки операции << , чтобы она означала "вывести" ("put to"), можно получить более простую запись и разрешить программисту выводить в одном операторе последовательность объектов, например так:

```
cerr << "x = " << x << '\n';
```

Здесь cerr обозначает стандартный поток ошибок. Так, если `x` типа `int` со значением 123, то приведенный оператор выдаст

```
x = 123
```

и еще символ конца строки в стандартный поток ошибок. Аналогично, если `x` имеет пользовательский тип `complex` со значением (1,2.4), то указанный оператор выдаст

```
x = (1,2.4)
```

в поток cerr. Такой подход легко использовать пока x такого типа, для которого определена операция <<, а пользователь может просто доопределить << для новых типов.

Мы использовали операцию вывода, чтобы избежать многословности, неизбежной, если применять функцию вывода. Но почему именно символ <<? Невозможно изобрести новую лексему (лекцию 7). Кандидатом для ввода и вывода была операция присваивания, но большинство людей предпочитает, чтобы операции ввода и вывода были различны. Более того, порядок выполнения операции = неподходящий, так `cout=a=b` означает `cout=(a=b)`. Пробовали использовать операции < и >, но к ним так крепко привязано понятие "меньше чем" и "больше чем", что операции ввода-вывода с ними во всех практически случаях не поддавались прочтению.

Операции << и >> похоже не создают таких проблем. Они асиметричны, что позволяет приписывать им смысл "в" и "из". Они не относятся к числу наиболее часто используемых операций над встроенными типами, а приоритет << достаточно низкий, чтобы писать арифметические выражения в качестве операнда без скобок:

```
cout << "a*b+c=" << a*b+c << '\n';
```

Скобки нужны, если выражение содержит операции с более низким приоритетом:

```
cout << "a^b|c=" << (a^b|c) << '\n';
```

Операцию сдвига влево можно использовать в операции вывода, но, конечно, она должна быть в скобках:

```
cout << "a<<b=" << (a<<b) << '\n';
```

10.2.1 Вывод встроенных типов

Для управления выводом встроенных типов определяется класс `ostream` с операцией `<<` (вывести):

```
class ostream : public virtual ios {
   // ...
public:
   ostream& operator<<(const char*);    //строки
   ostream& operator<<(char);
   ostream& operator<<(short i)
     { return *this << int(i); }
   ostream& operator<<(int);
   ostream& operator<<(long);
   ostream& operator<<(double);
   ostream& operator<<(const void*);    // указатели
    // ...
          };
```

Естественно, в классе `ostream` должен быть набор функций `operator<<()` для работы с беззнаковыми типами.

Функция `operator<<` возвращает ссылку на класс `ostream`, из которого она вызывалась, чтобы к ней можно было применить еще раз `operator<<`. Так, если `x` типа `int`, то

```
cerr << "x = " << x;
```

понимается как

```
(cerr.operator<<("x = ")).operator<<(x);
```

В частности, это означает, что если несколько объектов выводятся с помощью одного оператора вывода, то они будут выдаваться в естественном порядке: слева - направо.

Функция `ostream::operator<<(int)` выводит целые значения, а функция `ostream::operator<<(char)` - символьные. Поэтому функция

```
void val(char c)
{
  cout << "int('"<< c <<"') = " << int(c) << '\n';
}
```

печатает целые значения символов и с помощью программы

```
main()
{
  val('A');
  val('Z');
}
```

будет напечатано

```
int('A') = 65
int('Z') = 90
```

Здесь предполагается кодировка символов ASCII, на вашей машине может быть иной результат. Обратите внимание, что символьная константа имеет тип `char`, поэтому `cout<<'Z'` напечатает букву Z, а вовсе не целое 90.

Функция `ostream::operator<<(const void*)` напечатает значение указателя в такой записи, которая более подходит для используемой системы адресации.

Программа

```
main()
{
 int i = 0;
 int* p = new int(1);
 cout << "local " << &i
      << ", free store " << p << '\n';
}
```

выдаст на машине, используемой автором,

```
local 0x7fffead0, free store 0x500c
```

Для других систем адресации могут быть иные соглашения об изображении значений указателей.

Обсуждение базового класса `ios` отложим до 10.4.1.

10.2.2 Вывод пользовательских типов

Рассмотрим пользовательский тип данных:

```
class complex {
   double re, im;
public:
   complex(double r = 0, double i = 0) { re=r; im=i; }

   friend double real(complex& a) { return a.re; }
   friend double imag(complex& a) { return a.im; }

   friend complex operator+(complex, complex);
   friend complex operator-(complex, complex);
   friend complex operator*(complex, complex);
   friend complex operator/(complex, complex);
   //...
};
```

Для нового типа `complex` операцию `<<` можно определить так:

```
ostream& operator<<(ostream&s, complex z)
```

```
{
 return s << '(' << real(z) << ',' << imag(z) << ')';
};
```

и использовать как `operator<<` для встроенных типов. Например,

```
main()
{
   complex x(1,2);
   cout << "x = " << x << '\n';
}
```

выдаст

```
x = (1,2)
```

Для определения операции вывода над пользовательскими типами данных не нужно модифицировать описание класса `ostream`, не требуется и доступ к структурам данных, скрытым в описании класса. Последнее очень кстати, поскольку описание класса `ostream` находится среди стандартных заголовочных файлов, доступ по записи к которым закрыт для большинства пользователей, и изменять которые они вряд ли захотят, даже если бы могли. Это важно и по той причине, что дает защиту от случайной порчи этих структур данных. Кроме того имеется возможность изменить реализацию `ostream`, не затрагивая пользовательских программ.

10.3 Ввод

Ввод во многом сходен с выводом. Есть класс istream, который реализует операцию ввода `>>` ("ввести из" - "input from") для небольшого набора стандартных типов. Для пользовательских типов можно определить функцию `operator>>`.

10.3.1 Ввод встроенных типов

Класс `istream` определяется следующим образом:

```
class istream : public virtual ios {
    //...
public:
    istream& operator>>(char*);     // строка
    istream& operator>>(char&);     // символ
    istream& operator>>(short&);
    istream& operator>>(int&);
    istream& operator>>(long&);
    istream& operator>>(float&);
    istream& operator>>(double&);
    //...
};
```

Функции ввода `operator>>` определяются так:

```
istream& istream::operator>>(T& tvar)
{
 // пропускаем обобщенные пробелы
 // каким-то образом читаем T в `tvar'
  return *this;
 }
```

Теперь можно ввести в `VECTOR` последовательность целых, разделяемых пробелами, с помощью функции:

```
int readints(Vector<int>& v)
// возвращаем число прочитанных целых
{
   for (int i = 0; i<v.size(); i++)
   {
     if (cin>>v[i]) continue;
     return i;
   }
   // слишком много целых для размера Vector
   // нужна соответствующая обработка ошибки
 }
```

Появление значения с типом, отличным от `int`, приводит к прекращению операции ввода, и цикл ввода завершается. Так, если мы

вводим

```
1 2 3 4 5. 6 7 8.
```

то функция `readints()` прочитает пять целых чисел

```
1 2 3 4 5
```

Символ точка останется первым символом, подлежащим вводу. Под пробелом, как определено в стандарте C, понимается обобщенный пробел, т.е. пробел, табуляция, конец строки, перевод строки или возврат каретки. Проверка на обобщенный пробел возможна с помощью функции `isspace()` из файла <ctype.h>.

В качестве альтернативы можно использовать функции get():

```
class istream : public virtual ios {
    //...
    istream& get(char& c);     // символ
    istream& get(char* p, int n, char ='n');   // строка
};
```

В них обобщенный пробел рассматривается как любой другой символ, и они предназначены для таких операций ввода, когда не делается никаких предположений о вводимых символах.

Функция `istream::get(char&)` вводит один символ в свой параметр. Поэтому программу посимвольного копирования можно написать так:

```
main()
{
  char c;
  while (cin.get(c)) cout << c;
}
```

Такая запись выглядит несимметрично, и у операции `>>` для вывода символов есть двойник под именем `put()`, так что можно писать и так:

```
main()
```

```
{
  char c;
  while (cin.get(c)) cout.put(c);
}
```

Функция с тремя параметрами `istream::get()` вводит в символьный вектор не более `n` символов, начиная с адреса `p`. При всяком обращении к `get()` все символы, помещенные в буфер (если они были), завершаются 0, поэтому если второй параметр равен `n`, то введено не более `n-1` символов. Третий параметр определяет символ, завершающий ввод. Типичное использование функции `get()` с тремя параметрами сводится к чтению строки в буфер заданного размера для ее дальнейшего разбора, например так:

```
void f()
{
   char buf[100];
   cin >> buf; // подозрительно
   cin.get(buf,100,'\n');      // надежно
   //...
 }
```

Операция `cin>>buf` подозрительна, поскольку строка из более чем 99 символов переполнит буфер. Если обнаружен завершающий символ, то он остается в потоке первым символом подлежащим вводу. Это позволяет проверять буфер на переполнение:

```
void f()
{
  char buf[100];

  cin.get(buf,100,'\n');   // надежно

  char c;
  if (cin.get(c) && c!='\n') {
    // входная строка больше, чем ожидалось
  }
  //...
 }
```

Естественно, существует версия `get()` для типа `unsigned char`.

В стандартном заголовочном файле <ctype.h> определены несколько функций, полезных для обработки при вводе:

```
int isalpha(char)   // 'a'..'z' 'A'..'Z'
int isupper(char)   // 'A'..'Z'
int islower(char)   // 'a'..'z'
int isdigit(char)   // '0'..'9'
int isxdigit(char)  // '0'..'9' 'a'..'f' 'A'..'F'
int isspace(char)   // ' ' '\t' возвращает конец строки
   // и перевод формата
int iscntrl(char)   // управляющий символ в диапазоне
   // (ASCII 0..31 и 127)
int ispunct(char)   // знак пунктуации, отличен от приведенных выше
int isalnum(char)   // isalpha() | isdigit()
int isprint(char)   // видимый: ascii ' '..'~'
int isgraph(char)     // isalpha() | isdigit() | ispunct()
int isascii(char c)   { return 0<=c && c<=127; }
```

Все они, кроме `isascii()`, работают с помощью простого просмотра, используя символ как индекс в таблице атрибутов символов. Поэтому вместо выражения типа

```
(('a'<=c && c<='z') || ('A'<=c && c<='Z')) // буква
```

которое не только утомительно писать, но оно может быть и ошибочным (на машине с кодировкой *EBCDIC* оно задает не только буквы), лучше использовать вызов стандартной функции `isalpha()`, который к тому же более эффективен.

В качестве примера приведем функцию `eatwhite()`, которая читает из потока обобщенные пробелы:

```
istream& eatwhite(istream& is)
{
    char c;
    while (is.get(c)) {
        if (isspace(c)==0) {
            is.putback(c);
```

```
            break;
        }
    }
    return is;
}
```

В ней используется функция `putback()`, которая возвращает символ в поток, и он становится первым подлежащим чтению.

10.3.2 Состояния потока

С каждым потоком (`istream` или `ostream`) связано определенное состояние. Нестандартные ситуации и ошибки обрабатываются с помощью проверки и установки состояния подходящим образом.

Узнать состояние потока можно с помощью операций над классом `ios`:

```
class ios { //ios является базовым для ostream и istream
    //...
public:
    int eof() const;    // дошли до конца файла
    int fail() const;   // следующая операция будет неудачна
    int bad() const;    // поток испорчен
    int good() const;   // следующая операция будет успешной
    //...
};
```

Последняя операция ввода считается успешной, если состояние задается `good()` или `eof()`. Если состояние задается `good()`, то последующая операция ввода может быть успешной, в противном случае она будет неудачной. Применение операции ввода к потоку в состоянии, задаваемом не `good()`, считается пустой операцией. Если произошла неудача при попытке чтения в переменную `v`, то значение `v` не изменилось (оно не изменится, если v имеет тип, управляемый функциями члена из `istream` или `ostream`). Различие между состояниями, задаваемыми как `fail()` или как `bad()` уловить трудно, и оно имеет смысл только для разработчиков операций ввода. Если состояние есть `fail()`, то считается, что поток не поврежден, и

никакие символы не пропали; о состоянии `bad()` ничего сказать нельзя.

Значения, обозначающие эти состояния, определены в классе `ios`:

```
class ios {
    //...
public:
    enum io_state {
        goodbit=0,
        eofbit=1,
        filebit=2,
        badbit=4,
    };
    //...
};
```

Истинные значения состояний зависят от реализации, и указанные значения приведены только, чтобы избежать синтаксически неправильных конструкций.

Проверять состояние потока можно следующим образом:

```
switch (cin.rdstate()) {
case ios::goodbit:
    // последняя операция с cin была успешной
    break;
case ios::eofbit:
    // в конце файла
    break;
 case ios::filebit:
    // некоторый анализ ошибки
    // возможно неплохой
    break;
  case ios::badbit:
    //  cin возможно испорчен
    break;
  }
```

В более ранних реализациях для значений состояний использовались

глобальные имена. Это приводило к нежелательному засорению пространства именования, поэтому новые имена доступны только в пределах класса `ios`. Если вам необходимо использовать старые имена в сочетании с новой библиотекой, можно воспользоваться следующими определениями:

```
const int _good = ios::goodbit;
const int _bad = ios::badbit;
const int _file = ios::filebit;
const int _eof = ios::eofbit;

typedef ios::io_state state_value ;
```

Разработчики библиотек должны заботиться о том, чтобы не добавлять новых имен к глобальному пространству именования. Если элементы перечисления входят в общий интерфейс библиотеки, они всегда должны использоваться в классе с префиксами, например, как `ios::goodbit` и `ios::io_state`.

Для переменной любого типа, для которого определены операции << и >>, цикл копирования записывается следующим образом:

```
while (cin>>z) cout << z << '\n';
```

Если поток появляется в условии, то проверяется состояние потока, и условие выполняется (т.е. результат его не 0) только для состояния `good()`. Как раз в приведенном выше цикле проверяется состояние потока istream, что является результатом операции `cin>>z`. Чтобы узнать, почему произошла неудача в цикле или условии, надо проверить состояние. Такая проверка для потока реализуется с помощью операции приведения (7.3.2).

Так, если z является символьным вектором, то в приведенном цикле читается стандартный ввод и выдается для каждой строки стандартного вывода по одному слову (т.е. последовательности символов, не являющихся обобщенными пробелами). Если `z` имеет тип `complex`, то в этом цикле с помощью операций, определенных в 10.2.2 и 10.2.3, будут копироваться комплексные числа. Шаблонную функцию копирования для потоков со значениями произвольного типа можно

написать следующим образом:

```
complex z;
iocopy(z,cin,cout);    // копирование complex

double d;
iocopy(d,cin,cout);    // копирование double
char c;
iocopy(c,cin,cout);    // копирование char
```

Поскольку надоедает проверять на корректность каждую операцию ввода- вывода, то распространенным источником ошибок являются именно те места в программе, где такой контроль существенен. Обычно операции вывода не проверяют, но иногда они могут завершиться неудачно. Потоковый ввод- вывод разрабатывался из того принципа, чтобы сделать исключительные ситуации легкодоступными, и тем самым упростить обработку ошибок в процессе ввода-вывода.

10.3.3 Ввод пользовательских типов

Операцию ввода для пользовательского типа можно определить в точности так же, как и операцию вывода, но для операции ввода существенно, чтобы второй параметр имел тип ссылки, например:

```
istream& operator>>(istream& s, complex& a)
/*
   формат input рассчитан на complex; "f" обозначает float:
      f
      ( f )
      ( f , f )
*/
{
   double re = 0, im = 0;
   char   c = 0;

   s >> c;
   if (c == '(') {
      s >> re >> c;
```

```
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(ios::badbit); // установим состояние
    }
    else {
        s.putback(c);
        s >> re;
    }

    if (s) a = complex(re,im);
    return s;
}
```

Несмотря на сжатость кода, обрабатывающего ошибки, на самом деле учитывается большая часть ошибок. Инициализация локальной переменной c нужна для того, чтобы в нее не попало случайное значение, например '(', в случае неудачной операции. Последняя проверка состояния потока гарантирует, что параметр a получит значение только при успешном вводе.

Операция, устанавливающая состояние потока, названа `clear()` (здесь `clear` - очистить), поскольку чаще всего она используется для восстановления состояния потока как `good()` ; значением по умолчанию для параметра `ios::clear()` является `ios::goodbit`.

10.4 Форматирование

Все примеры из 10.2 содержали неформатированный вывод, который являлся преобразованием объекта в последовательность символов, задаваемую стандартными правилами, длина которой также определяется этими правилами. Часто программистам требуются более развитые возможности. Так, возникает потребность контролировать размер памяти, необходимой для операции вывода, и формат, используемый для выдачи чисел. Точно так же допустимо управление некоторыми аспектами ввода.

10.4.1 Класс ios

Большинство средств управления вводом-выводом сосредоточены в классе ios, который является базовым для `ostream` и `istream`. По сути здесь находится управление связью между `istream` или `ostream` и буфером, используемым для операций ввода-вывода. Именно класс `ios` контролирует, как символы попадают в буфер и как они выбираются оттуда. Так, в классе `ios` есть член, содержащий информацию об используемой при чтении или записи целых чисел системы счисления (десятичная, восьмеричная или шестнадцатеричная), о точности вещественных чисел и т.п., а также функции для проверки и установки значений переменных, управляющих потоком.

```
class ios {
   //...
public:
   ostream* tie(ostream* s);   // связать input и output
   ostream* tie();             // возвратить "tie"

int width(int w);              // установить поле width
int width() const;

char fill(char);               // установить символ заполнения
char fill() const;             // вернуть символ заполнения

long flags(long f);
long flags() const;

long setf(long setbits, long field);
long setf(long);
long unsetf(long);

int precision(int);            // установить точность для float
int precision() const;
int rdstate(); const;          // состояния потоков, см. 10.3.2
int eof() const;
int fail() const;
int bad() const;
int good() const;
void clear(int i=0);
```

```
//...
        };
```

В 10.3.2 описаны функции, работающие с состоянием потока, остальные приведены ниже.

10.4.1.1 Связывание потоков

Функция `tie()` может установить и разорвать связь между `ostream` и `istream`. Рассмотрим пример:

```
main()
{
 String s;
 cout << "Password: ";
 cin >> s;
 // ...
 }
```

Как можно гарантировать, что приглашение `Password:` появится на экране прежде, чем выполнится операция чтения? Вывод в `cout` и ввод из `cin` буферизуются, причем независимо, поэтому `Password:` появится только по завершении программы, когда закроется буфер вывода.

Решение состоит в том, чтобы связать `cout` и `cin` с помощью операции `cin.tie(cout)`.

Если `ostream` связан с потоком `istream`, то буфер вывода выдается при каждой операции ввода над `istream`. Тогда операции

```
cout << "Password: ";
cin >> s;
```

эквивалентны

```
cout << "Password: ";
cout.flush();
```

```
cin >> s;
```

Обращение `is.tie(0)` разрывает связь между потоком `is` и потоком, с которым он был связан, если такой был. Подобно другим потоковым функциям, устанавливающим определенное значение, `tie(s)` возвращает предыдущее значение, т.е. значение связанного потока перед обращением или 0. Вызов без параметра `tie()` возвращает текущее значение.

10.4.1.2 Поля вывода

Функция `width()` устанавливает минимальное число символов, использующееся в последующей операции вывода числа или строки. Так в результате следующих операций

```
cout.width(4);
cout << '(' << 12 << ')';
```

получим число 12 в поле размером 4 символа, т.е.

```
(12)
```

Заполнение поля заданными символами или выравнивание можно установить с помощью функции `fill()`, например:

```
cout.width(4);
cout.fill('#');
cout << '(' << "ab" << ')';
```

напечатает

```
###(ab)
```

По умолчанию поле заполняется пробелами, а размер поля по умолчанию есть 0, что означает "столько символов, сколько нужно". Вернуть размеру поля стандартное значение можно с помощью вызова

```
cout.width(0); // ``столько символов, сколько надо"
```

Функция `width()` задает минимальное число символов. Если появится больше символов, они будут напечатаны все, поэтому

```
cout.width(4);
cout << '(' << "121212" << ')\n';
```

напечатает

```
(121212)
```

Причина, по которой разрешено переполнение поля, а не усечение вывода, в том, чтобы избежать зависания при выводе. Лучше получить правильную выдачу, выглядящую некрасиво, чем красивую выдачу, являющуюся неправильной.

Вызов `width()` влияет только на одну следующую за ним операцию вывода, поэтому

```
cout.width(4);
cout.fill('#');
cout << '(' << 12 << '),(' << '(' <<12 << ')\n';
```

напечатает

```
(##12),((12)
```

а не

```
(##12),(##12)
```

как можно было бы ожидать. Однако, заметьте, что если бы влияние распространялось на все операции вывода чисел и строк, получился бы еще более неожиданный результат:

```
(##12#),(##12#
)
```

С помощью стандартного манипулятора, показанного в 10.4.2.1, можно более элегантно задавать размера поля вывода.

10.4.1.3 Состояние формата

В классе `ios` содержится состояние формата, которое управляется функциями `flags()` и `setf()`. По сути эти функции нужны, чтобы установить или отменить следующие флаги:

```
class ios {
public:
    // управляющие форматом флаги:
  enum {
    skipws=01,          // пропуск обобщенных пробелов для input
    // поле выравнивания:
    left=02,            // добавление перед значением
    right=04,           // добавление после значения
    internal=010,       // добавление между знаком и значением
    // основание целого:
    dec=020,            // восьмеричное
    oct=040,            // десятичное
    hex=0100,           // шестнадцатеричное
    showbase=0200,      // показать основание целого
    showpoint=0400,     // выдать нули в конце
    uppercase=01000,    // 'E', 'X' , а не 'e', 'x'
    showpos=02000,      // '+' для положительных чисел
    // запись числа типа float:
    scientific=04000,   // .dddddd Edd
    fixed=010000,       // dddd.dd
    // сброс в выходной поток:
    unitbuf=020000,     // после каждой операции
    stdio=040000        // после каждого символа
  };
  //...
};
```

Смысл флагов будет разъяснен в последующих разделах. Конкретные значения флагов зависят от реализации и даны здесь только для того, чтобы избежать синтаксически неверных конструкций.

Определение интерфейса как набора флагов и операций для их установки или отмены - это оцененный временем, хотя и несколько

устаревший прием. Основное его достоинство в том, что пользователь может собрать воедино набор флагов, например, так:

```
const int my_io_options =
   ios::left|ios::oct|ios::showpoint|ios::fixed;
```

Такое множество флагов можно задавать как параметр одной операции

```
cout.flags(my_io_options);
```

а также просто передавать между функциями одной программы:

```
void your_function(int ios_options);

void my_function()
{
   // ...
  your_function(my_io_options);
  // ...
}
```

Множество флагов можно установить с помощью функции `flags()`, например:

```
void your_function(int ios_options)
{
 int old_options = cout.flags(ios_options);
 // ...
 cout.flags(old_options);  // reset options
 }
```

Функция `flags()` возвращает старое значение множества флагов. Это позволяет переустановить значения всех флагов, как показано выше, а также задать значение отдельному флагу. Например вызов

```
myostream.flags(myostream.flags()|ios::showpos);
```

заставляет класс `myostream` выдавать положительные числа со знаком + и, в то же время, не меняет значения других флагов. Получается старое значение множества флагов, к которому добавляется

с помощью операции | флаг showpos. Функция setf( ) делает то же самое, поэтому эквивалентная запись имеет вид

```
myostream.setf(ios::showpos);
```

После установки флаг сохраняет значение до явной отмены.

Все-таки управление вводом-выводом с помощью установки и отмены флагов - грубое и ведущее к ошибкам решение. Если только вы тщательно не изучите свое справочное руководство и не будете применять флаги только в простых случаях, как это делается в последующих разделах, то лучше использовать манипуляторы (описанные в 10.4.2.1). Приемы работы с состоянием потока лучше изучить на примере реализации класса, чем изучая интерфейс класса.

10.4.1.4 Вывод целых

Прием задания нового значения множества флагов с помощью операции | и функций flags() и setf() работает только тогда, когда один бит определяет значение флага. Не такая ситуация при задании системы счисления целых или вида выдачи вещественных. Здесь значение, определяющее вид выдачи, нельзя задать одним битом или комбинацией отдельных битов.

Решение, принятое в <iostream.h>, сводится к использованию версии функции setf(), работающей со вторым "псевдопараметром", который показывает какой именно флаг мы хотим добавить к новому значению.

Поэтому обращения

```
cout.setf(ios::oct,ios::basefield); // восьмеричное
cout.setf(ios::dec,ios::basefield); // десятичное
cout.setf(ios::hex,ios::basefield); // шестнадцатеричное
```

установят систему счисления, не затрагивая других компонентов состояния потока. Если система счисления установлена, она используется до явной переустановки, поэтому

```
cout << 1234 << ' '; // десятичное по умолчанию
cout << 1234 << ' ';

cout.setf(ios::oct,ios::basefield); // восьмеричное
cout << 1234 << ' ';
cout << 1234 << ' ';

cout.setf(ios::hex,ios::basefield); // шестнадцатеричное
cout << 1234 << ' ';
cout << 1234 << ' ';
```

напечатает

```
1234 1234 2322 2322 4d2 4d2
```

Если появится необходимость указывать систему счисления для каждого выдаваемого числа, следует установить флаг `showbase`. Поэтому, добавив перед приведенными выше обращениями

```
cout.setf(ios::showbase);
```

мы получим

```
1234 1234 02322 02322 0x4d2 0x4d2
```

Стандартные манипуляторы, приведенные в § 10.4.2.1, предлагают более элегантный способ определения системы счисления при выводе целых.

10.4.1.5 Выравнивание полей

С помощью обращений к `setf()` можно управлять расположением символов в пределах поля:

```
cout.setf(ios::left,ios::adjustfield);     // влево
cout.setf(ios::right,ios::adjustfield);    // вправо
cout.setf(ios::internal,ios::adjustfield); // внутреннее
```

Будет установлено выравнивание в поле вывода, определяемом

функцией `ios::width()`, причем не затрагивая других компонентов состояния потока.

Выравнивание можно задать следующим образом:

```
cout.width(4);
cout << '(' << -12 << ')\n';

cout.width(4);
cout.setf(ios::left,ios::adjustfield);
cout << '(' << -12 << ')\n';

cout.width(4);
cout.setf(ios::internal,ios::adjustfield);
cout << '(' << -12) << '\n';
```

что выдаст

```
(-12)
( -12 )
(-12)
```

Если установлен флаг выравнивания `internal` (внутренний), то символы добавляются между знаком и величиной. Как видно, стандартным является выравнивание вправо.

10.4.1.6 Вывод плавающих чисел.

Вывод вещественных величин также управляется с помощью функций, работающих с состоянием потока. В частности, обращения:

```
cout.setf(ios::scientific,ios::floatfield);
cout.setf(ios::fixed,ios::floatfield);
cout.setf(0,ios::floatfield);   // вернуться к стандартному
```

установят вид печати вещественных чисел без изменения других компонентов состояния потока.

Например:

```
cout << 1234.56789 << '\n';

cout.setf(ios::scientific,ios::floatfield);
cout << 1234.56789 << '\n';

cout.setf(ios::fixed,ios::floatfield);
cout << 1234.56789 << '\n';
```

напечатает

```
1234.57
1.234568e+03
1234.567890
```

После точки печатается `n` цифр, как задается в обращении

```
cout.precision(n)
```

По умолчанию `n` равно 6. Вызов функции `precision` влияет на все операции ввода-вывода с вещественными до следующего обращения к precision, поэтому

```
cout.precision(8);
cout << 1234.56789 << '\n';
cout << 1234.56789 << '\n';

cout.precision(4);
cout << 1234.56789 << '\n';
cout << 1234.56789 << '\n';
```

выдаст

```
1234.5679
1234.5679
1235
1235
```

Заметьте, что происходит округление, а не отбрасывание дробной части.

Стандартные манипуляторы, введенные в § 10.4.2.1, предлагают более

элегантный способ задания формата вывода вещественных.

10.4.2 Манипуляторы

К ним относятся разнообразные операции, которые приходится применять сразу перед или сразу после операции ввода-вывода. Например:

```
cout << x;
cout.flush();
cout << y;

cin.eatwhite();
cin >> x;
```

Если писать отдельные операторы как выше, то логическая связь между операторами неочевидна, а если утеряна логическая связь, программу труднее понять.

Идея манипуляторов позволяет такие операции как `flush()` или `eatwhite()` прямо вставлять в список операций ввода-вывода. Рассмотрим операцию `flush()`. Можно определить класс с операцией `operator<<()`, в котором вызывается `flush()`:

```
class Flushtype { };

ostream& operator<<(ostream& os, Flushtype)
{
  return flush(os);
}
```

определить объект такого типа

```
Flushtype FLUSH;
```

и добиться выдачи буфера, включив `FLUSH` в список объектов, подлежащих выводу:

```
cout << x << FLUSH << y << FLUSH ;
```

Теперь установлена явная связь между операциями вывода и сбрасывания буфера. Однако, довольно быстро надоест определять класс и объект для каждой операции, которую мы хотим применить к поточной операции вывода. К счастью, можно поступить лучше. Рассмотрим такую функцию:

```
typedef ostream& (*Omanip) (ostream&);

ostream& operator<<(ostream& os, Omanip f)
{
  return f(os);
}
```

Здесь операция вывода использует параметры типа "указатель на функцию, имеющую аргумент `ostream&` и возвращающую `ostream&`". Отметив, что `flush()` есть функция типа "функция с аргументом `ostream&` и возвращающая `ostream&` ", мы можем писать

```
cout << x << flush << y << flush;
```

получив вызов функции `flush()`. На самом деле в файле <iostream.h> функция `flush()` описана как

```
ostream& flush(ostream&);
```

а в классе есть операция `operator<<`, которая использует указатель на функцию, как указано выше:

```
class ostream : public virtual ios {
   // ...
public:
  ostream& operator<<(ostream& ostream& (*)(ostream&));
  // ...
};
```

В приведенной ниже строке буфер выталкивается в поток `cout` дважды в подходящее время:

```
cout << x << flush << y << flush;
```

Похожие определения существуют и для класса `istream`:

```
istream& ws(istream& is ) { return is.eatwhite(); }

class istream : public virtual ios {
 // ...
public:
  istream& operator>>(istream&, istream& (*) (istream&));
  // ...
};
```

поэтому в строке

```
cin >> ws >> x;
```

действительно обобщенные пробелы будут убраны до попытки чтения в x. Однако, поскольку по умолчанию для операции `>>` пробелы "съедаются" и так, данное применение `ws()` избыточно.

Находят применение и манипуляторы с параметрами. Например, может появиться желание с помощью

```
cout << setprecision(4) << angle;
```

напечатать значение вещественной переменной `angle` с точностью до четырех знаков после точки.

Для этого нужно уметь вызывать функцию, которая установит значение переменной, управляющей в потоке точностью вещественных. Это достигается, если определить `setprecision(4)` как объект, который можно "выводить" с помощью operator<<():

```
class Omanip_int {
    int i;
    ostream& (*f) (ostream&,int);
public:
    Omanip_int(ostream& (*ff) (ostream&,int), int ii)
        : f(ff), i(ii) { }
    friend ostream& operator<<(ostream& os, Omanip& m)
        { return m.f(os,m.i); }
```

```
};
```

Конструктор `Omanip_int` хранит свои аргументы в `i` и `f`, а с помощью `operator<<` вызывается `f()` с параметром `i`. Часто объекты таких классов называют объект-функция. Чтобы результат строки

```
cout << setprecision(4) << angle
```

был таким, как мы хотели, необходимо чтобы обращение `setprecision(4)` создавало безымянный объект класса `Omanip_int`, содержащий значение 4 и указатель на функцию, которая устанавливает в потоке ostream значение переменной, задающей точность вещественных:

```
ostream& _set_precision(ostream&,int);

Omanip_int setprecision(int i)
{
    return Omanip_int(&_set_precision,i);
}
```

Учитывая сделанные определения, `operator<<()` приведет к вызову `precision(i)`.

Утомительно определять классы наподобие `Omanip_int` для всех типов аргументов, поэтому определим шаблон типа:

```
template<class T> class OMANIP {
    T i;
    ostream& (*f) (ostream&,T);
public:
    OMANIP(ostream (*ff) (ostream&,T), T ii)
        : f(ff), i(ii) { }

friend ostream& operator<<(ostream& os, OMANIP& m)
    { return m.f(os,m.i) }
};
```

С помощью `OMANIP` пример с установкой точности можно сократить

так:

```
ostream& precision(ostream& os,int)
{
    os.precision(i);
    return os;
}

OMANIP<int> setprecision(int i)
{
    return OMANIP<int>(&precision,i);
}
```

В файле <iomanip.h> можно найти шаблон типа `OMANIP`, его двойник для `istream` - шаблон типа `SMANIP`, а `SMANIP` - двойник для `ioss`. Некоторые из стандартных манипуляторов, предлагаемых поточной библиотекой, описаны ниже. Отметим,что программист может определить новые необходимые ему манипуляторы, не затрагивая определений `istream`, `ostream`, `OMANIP` или `SMANIP`.

Идею манипуляторов предложил А. Кениг. Его вдохновили процедуры разметки (layout) системы ввода-вывода Алгола68. Такая техника имеет много интересных приложений помимо ввода-вывода. Суть ее в том, что создается объект, который можно передавать куда угодно и который используется как функция. Передача объекта является более гибким решением, поскольку детали выполнения частично определяются создателем объекта, а частично тем, кто к нему обращается.

10.4.2.1 Стандартные манипуляторы ввода-вывода

Это следующие манипуляторы:

```
// Simple manipulators:
ios& oct(ios&);  // в восьмеричной записи
ios& dec(ios&);  // в десятичной записи
ios& hex(ios&);  // в шестнадцатеричной записи
ostream& endl(ostream&);   // добавить '\n' и вывести
ostream& ends(ostream&);   // добавить '\0' и вывести
```

```
ostream& flush(ostream&);  // выдать поток

istream& ws(istream&);     // удалить обобщенные пробелы

// Манипуляторы имеют параметры:

SMANIP<int>  setbase(int b);
SMANIP<int>  setfill(int f);
SMANIP<int>  setprecision(int p);
SMANIP<int>  setw(int w);
SMANIP<long> resetiosflags(long b);
SMANIP<long> setiosflags(long b);
```

Например,

```
cout << 1234 << ' '
     << hex  << 1234 << ' '
     << oct  << 1234 << endl;
```

напечатает

```
1234 4d2 2322
```

и

```
cout << setw(4) << setfill('#') << '(' << 12 << ')\n';
cout << '(' << 12 << ')\n';
```

напечатает

```
(##12)
(12)
```

Не забудьте включить файл <iomanip.h>, если используете манипуляторы с параметрами.

10.4.3 Члены ostream

В классе `ostream` есть лишь несколько функций для управления

выводом, большая часть таких функций находится в классе `ios`.

```
class ostream : public virtual ios {
    //...
public:
    ostream& flush();

    ostream&  seekp(streampos);
    ostream&  seekp(streamoff, seek_dir);
    streampos tellp();
    //...
};
```

Как мы уже говорили, функция `flush()` опустошает буфер в выходной поток. Остальные функции используются для позиционирования в ostream при записи. Окончание на букву p указывает, что именно позиция используется при выдаче символов в заданный поток. Конечно эти функции имеют смысл, только если поток присоединен к чему-либо, что допускает позиционирование, например файл. Тип `streampos` представляет позицию символа в файле, а тип `streamoff` представляет смещение относительно позиции, заданной `seek_dir`. Все они определены в классе ios:

```
class ios {
    //...
    enum seek_dir {
        beg=0,  // от начала файла
        cur=1,  // от текущей позиции в файле
        end=2   // от конца файла
    };
    //...
};
```

Позиции в потоке отсчитываются от 0, как если бы файл был массивом из n символов:

```
char file[n-1];
```

и если `fout` присоединено к `file`, то

```
fout.seek(10);
fout<<'#';
```

поместит # в file[10].

10.4.4 Члены istream

Как и для `ostream`, большинство функций форматирования и управления вводом находится не в классе `iostream`, а в базовом классе `ios`.

```
class istream : public virtual ios {
    //...
public:
    int         peek()
    istream&    putback(char c);

    istream&    seekg(streampos);
    istream&    seekg(streamoff, seek_dir);
    streampos   tellg();
    //...
};
```

Функции позиционирования работают как и их двойники из `ostream`. Окончание на букву `g` показывает, что именно позиция используется при вводе символов из заданного потока. Буквы `p` (put) и `g` (get) нужны, поскольку мы можем создать производный класс `iostreams` из классов `ostream` и `istream`, и в нем необходимо следить за позициями ввода и вывода.

С помощью функции `peek()` программа может узнать следующий символ, подлежащий вводу, не затрагивая результата последующего чтения. С помощью функции `putback()`, как показано в § 10.3.3, можно вернуть ненужный символ назад в поток, чтобы он был прочитан в другое время.

10.5 Файлы и потоки

Ниже приведена программа копирования одного файла в другой. Имена файлов берутся из командной строки программы:

```
#include <fstream.h>
#include <libc.h>

void error(char* s, char* s2 ="")
{
  cerr << s << ' ' << s2 << '\n';
  exit(1);
}

int main(int argc, char* argv[])
{
   if (argc != 3) error("wrong number of arguments");

   ifstream from(argv[1]);
   if (!from) error("cannot open input file",argv[1]);

   ostream to(argv[2]);
   if (!to) error("cannot open output file",argv[2]);

   char ch;
   while (from.get(ch)) to.put(ch);

   if (!from.eof() || to.bad())
      error("something strange happened");

   return 0;
}
```

Для открытия выходного файла создается объект класса `ofstream` - выходной поток файла, использующий в качестве аргумента имя файла. Аналогично, для открытия входного файла создается объект класса `ifstream` - входной файловый поток, также использующий в качестве аргумента имя файла. В обоих случаях следует проверить состояние созданного объекта, чтобы убедиться в успешном открытии файла, а если это не так, операции завершатся не успешно, но корректно.

По умолчанию `ifstream` всегда открывается на чтение, а `ofstream` открывается на запись. В `ostream` и в `istream` можно использовать необязательный второй аргумент, указывающий иные режимы открытия:

```
class ios {
public:
    //...
    enum open_mode {
        in=1,           // открыть на чтение
        out=2,          // открыть как выходной
        ate=4,          // открыть и переместиться в конец файла
        app=010,        // добавить
        trunc=020,      // сократить файл до нулевой длины
        nocreate=040,   // неудача, если файл не существует
        noreplace=0100  // неудача, если файл существует
    };
    //...
};
```

Настоящие значения для `open_mode` и их смысл вероятно будут зависеть от реализации. Будьте добры, за деталями обратитесь к руководству по вашей библиотеке или экспериментируйте. Приведенные комментарии могут прояснить их назначение. Например, можно открыть файл с условием, что операция открытия не выполнится, если файл уже не существует:

```
void f()
{
    ofstream mystream(name,ios::out|ios::nocreate);

    if (ofstream.bad()) {
        //...
    }
    //...
}
```

Также можно открыть файл сразу на чтение и запись:

```
fstream dictionary("concordance", ios::in|ios::out);
```

Все операции, допустимые для `ostream` и `istream`, можно применять к `fstream`. На самом деле, класс `fstream` является производным от `iostream`, который является, в свою очередь, производным от `istream` и `ostream`. Причина, по которой информация по буферизации и форматированию для `ostream` и `istream` находится в виртуальном базовом классе `ios`, в том, чтобы заставить действовать всю эту последовательность производных классов. По этой же причине операции позиционирования в `istream` и `ostream` имеют разные имена - `seekp()` и `seekg()`. В `iostream` есть отдельные позиции для чтения и записи.

10.5.1 Закрытие потоков

Файл может быть закрыт явно, если вызвать `close()` для его потока:

```
mystream.close();
```

Но это неявно делает деструктор потока, так что явный вызов `close()` может понадобиться, если только файл нужно закрыть до достижения конца области определенности потока.

Здесь возникает вопрос, как реализация может обеспечить создание предопределенных потоков `cout`, `cin` и `cerr` до их первого использования и закрытие их только после последнего использования. Конечно, разные реализации библиотеки потоков из <iostream.h> могут по-разному решать эту задачу. В конце концов, решение - это прерогатива реализации, и оно должно быть скрыто от пользователя. Здесь приводится только один способ, примененный только в одной реализации, но он достаточно общий, чтобы гарантировать правильный порядок создания и уничтожения глобальных объектов различных типов.

Основная идея в том, чтобы определить вспомогательный класс, который по сути служит счетчиком, следящим за тем, сколько раз <iostream.h> был включен в раздельно компилировавшиеся программные файлы:

```
class Io_init {
    static int count;
    //...
public:
    Io_init();
    ~Io_init();
};

static Io_init io_init ;
```

Для каждого программного файла определен свой объект с именем `io_init`. Конструктор для объектов `io_init` использует `Io_init::count` как первый признак того, что действительная инициализация глобальных объектов потоковой библиотеки ввода-вывода сделана в точности один раз:

```
Io_init::Io_init()
{
  if (count++ == 0) {
      // инициализировать cout
      // инициализировать cerr
      // инициализировать cin
      // и т.д.
  }
}
```

Обратно, деструктор для объектов `io_init` использует `Io_count`, как последнее указание на то, что все потоки закрыты:

```
Io_init::~Io_init()
{
  if (--count == 0) {
      // очистить cout (сброс, и т.д.)
      // очистить cerr (сброс, и т.д.)
      // очистить cin
      // и т.д.
  }
 }
```

Это общий прием работы с библиотеками, требующими инициализации и удаления глобальных объектов. Впервые в C++ его применил Д. Шварц. В системах, где при выполнении все программы размещаются в основной памяти, для этого приема нет помех. Если это не так, то накладные расходы, связанные с вызовом в память каждого программного файла для выполнения функций инициализации, будут заметны. Как всегда, лучше, по возможности, избегать глобальных объектов. Для классов, в которых каждая операция значительна по объему выполняемой работы, чтобы гарантировать инициализацию, было бы разумно проверять такие первые признаки (наподобие `Io_init::count`) при каждой операции. Однако, для потоков такой подход был бы излишне расточительным.

10.5.2 Строковые потоки

Как было показано, поток может быть привязан к файлу, т.е. массиву символов, хранящемуся не в основной памяти, а, например, на диске. Точно так же поток можно привязать к массиву символов в основной памяти. Например, можно воспользоваться выходным строковым потоком `ostrstream` для форматирования сообщений, не подлежащих немедленной печати:

```
char* p = new char[message_size];
ostrstream ost(p,message_size);
do_something(arguments,ost);
display(p);
```

С помощью стандартных операций вывода функция `do_something` может писать в поток `ost`, передавать `ost` подчиняющимся ей функциям и т.п. Контроль переполнения не нужен, поскольку ost знает свой размер и при заполнении перейдет в состояние, определяемое `fail()`. Затем функция `display` может послать сообщение в "настоящий" выходной поток. Такой прием наиболее подходит в тех случаях, когда окончательная операция вывода предназначена для записи на более сложное устройство, чем традиционное, ориентированное на последовательность строк, выводное устройство. Например, текст из `ost` может быть помещен в фиксированную область на экране.

Аналогично, `istrstream` является вводным строковым потоком, читающим из последовательности символов, заканчивающейся нулем:

```
void word_per_line(char v[], int sz)
/*
  печатать "v" размером "sz" по одному слову в строке
*/
{
  istrstream ist(v,sz);  // создать istream для v
  char b2[MAX];          // длиннее самого длинного слова
  while (ist>>b2) cout <<b2 << "\n";
 }
```

Завершающий нуль считается концом файла.

Строковые потоки описаны в файле <strstream.h>.

10.5.3 Буферизация

Все операции ввода-вывода были определены без всякой связи с типом файла, но нельзя одинаково работать со всеми устройствами без учета алгоритма буферизации. Очевидно, что потоку `ostream`, привязанному к строке символов, нужен не такой буфер, как `ostream`, привязанному к файлу. Такие вопросы решаются созданием во время инициализации разных буферов для потоков разных типов. Но существует только один набор операций над этими типами буферов, поэтому в `ostream` нет функций, код которых учитывает различие буферов. Однако, функции, следящие за переполнением и обращением к пустому буферу, являются виртуальными. Это хороший пример применения виртуальных функций для единообразной работы с эквивалентными логически, но различно реализованными структурами, и они вполне справляются с требуемыми алгоритмами буферизации. Описание буфера потока в файле <iostream.h> может выглядеть следующим образом:

```
class streambuf {     // управление буфером потока
protected:
    char* base;       // начало буфера
```

```
    char* pptr;      // следующий свободный байт
    char* gptr;      // следующий заполненный байт
    char* eptr;      // один из указателей на конец буфера
    char  alloc;     // буфер, размещенный с помощью "new"
    //...
    // Опустошить буфер:
    // Вернуть EOF при ошибке, 0 - удача
  virtual int overflow(int c = EOF);

    // Заполнить буфер:
    // Вернуть EOF в случае ошибки или конца входного потока,
    // иначе вернуть очередной символ
  virtual int underflow();
  //...
  public:
    streambuf();
    streambuf(char* p, int l);
    virtual ~streambuf();

    int snextc()      // получить очередной символ
    {
      return (++gptr==pptr) ? underflow() : *gptr&0377;
    }
    int allocate();   // отвести память под буфер
    //...
};
```

Подробности реализации класса `streambuf` приведены здесь только для полноты представления. Не предполагается, что есть общедоступные реализации, использующие именно эти имена. Обратите внимание на определенные здесь указатели, управляющие буфером; с их помощью простые посимвольные операции с потоком можно определить максимально эффективно (и причем однократно) как функции-подстановки. Только функции `overflow()` и `underflow()` требует своей реализации для каждого алгоритма буферизации, например:

```
class filebuf : public streambuf {
protected:
```

```
    int  fd;          // дескриптор файла
    char opened;      // признак открытия файла
public:
    filebuf() { opened = 0; }
    filebuf(int nfd, char* p, int l)
        : streambuf(p,l) { /* ... */ }
    ~filebuf() { close(); }

    int overflow(int c=EOF);
    int underflow();

    filebuf* open(char *name, ios::open_mode om);
    int close() { /* ... */ }
    //...
};
int filebuf::underflow()    // заполнить буфер из "fd"
{
    if (!opened || allocate()==EOF) return EOF;

    int count = read(fd, base, eptr-base);
    if (count < 1) return EOF;

    gptr = base;
    pptr = base + count;
    return *gptr & 0377; // &0377 предотвращает размножение знака
}
```

За дальнейшими подробностями обратитесь к руководству по реализации класса `streambuf`.

10.6 Ввод-вывод в C

Поскольку текст программ на C и на C++ часто путают, то путают иногда и потоковый ввод-вывод C++ и функции ввода-вывода семейства `printf` для языка C. Далее, т.к. C-функции можно вызывать из программы на C++, то многие предпочитают использовать более знакомые функции ввода-вывода C.

По этой причине здесь будет дана основа функций ввода-вывода С. Обычно операции ввода-вывода на С и на С++ могут идти по очереди на уровне строк. Перемешивание их на уровне посимвольного ввода-вывода возможно для некоторых реализаций, но такая программа может быть непереносимой. Некоторые реализации потоковой библиотеки С ++ при допущении ввода-вывода на С требуют вызова статической функции-члена `ios::sync_with_stdio()`.

В общем, потоковые функции вывода имеют перед стандартной функцией С `printf()` то преимущество, что потоковые функции обладают определенной типовой надежностью и единообразно определяют вывод объектов предопределенного и пользовательского типов.

Основная функция вывода С есть

```
int printf(const char* format, ...)
```

и она выводит произвольную последовательность параметров в формате, задаваемом строкой форматирования `format`. Строка форматирования состоит из объектов двух типов: простые символы, которые просто копируются в выходной поток, и спецификации преобразований, каждая из которых преобразует и печатает очередной параметр. Каждая спецификация преобразования начинается с символа `%`, например

```
printf("there were %d members present.",no_of_members);
```

Здесь `%d` указывает, что `no_of_members` следует считать целым и печатать как соответствующую последовательность десятичных цифр. Если `no_of_members==127`, то будет напечатано

```
there were 127 members present.
```

Набор спецификаций преобразований достаточно большой и обеспечивает большую гибкость печати. За символом `%` может следовать:

- - необязательный знак минус, задающий выравнивание влево в указанном поле для преобразованного значения;

- d необязательная строка цифр, задающая ширину поля; если в преобразованном значении меньше символов, чем ширина строки, то оно дополнится до ширины поля пробелами слева (или справа, если дана спецификация выравнивания влево); если строка ширины поля начинается с нуля, то дополнение будет проводится нулями, а не пробелами;
- . необязательный символ точка служит для отделения ширины поля от последующей строки цифр;
- d необязательная строка цифр, задающая точность, которая определяет число цифр после десятичной точки для значений в спецификациях e или f, или же задает максимальное число печатаемых символов строки;
- * для задания ширины поля или точности может использоваться * вместо строки цифр. В этом случае должен быть параметр целого типа, который содержит значение ширины поля или точности;
- h необязательный символ h указывает, что последующая спецификация d, o, x или u относится к параметру типа короткое целое;
- l необязательный символ l указывает, что последующая спецификация d, o, x или u относится к параметру типа длинное целое;
- % обозначает, что нужно напечатать сам символ %; параметр не нужен;
- c символ, указывающий тип требуемого преобразования.Символы преобразования и их смысл следующие:
  - d Целый параметр выдается в десятичной записи;
  - o Целый параметр выдается в восьмеричной записи;
  - x Целый параметр выдается в шестнадцатеричной записи;
  - f Вещественный или с двойной точностью параметр выдается в десятичной записи вида `[-]ddd.ddd`, где число цифр после точки равно спецификации точности для параметра. Если точность не задана, печатается шесть цифр; если явно задана точность 0, точка и цифры после нее не печатаются;
  - e Вещественный или с двойной точностью параметр выдается в десятичной записи вида `[-]d.ddde+dd`; здесь одна цифра перед точкой, а число цифр после точки равно спецификации точности для параметра; если она не задана печатается шесть цифр;

- g Вещественный или с двойной точностью параметр печатается по той спецификации d, f или e, которая дает большую точность при меньшей ширине поля;
- c Символьный параметр печатается. Нулевые символы игнорируются;
- s Параметр считается строкой (символьный указатель), и печатаются символы из строки до нулевого символа или до достижения числа символов, равного спецификации точности; но, если точность равна 0 или не указана, печатаются все символы до нулевого;
- p Параметр считается указателем и его вид на печати зависит от реализации;
- u Беззнаковый целый параметр печатается в десятичной записи. Несуществующее поле или поле с шириной, меньшей реальной, приведет к усечению поля. Дополнение пробелами происходит, если только спецификация ширины поля больше реальной ширины.

Ниже приведен более сложный пример:

```
char* src_file_name;
int line;
char* line_format = "\n#line %d \"%s\"\n";
main()
{
  line = 13;
  src_file_name = "C++/main.c";

  printf("int a;\n");
  printf(line_format,line,src_file_name);
  printf("int b;\n");
}
```

в котором печатается

```
int a;

#line 13 "C++/main.c"
int b;
```

Использование `printf()` ненадежно в том смысле, что нет никакого контроля типов. Так, ниже приведен известный способ получения неожиданного результата - печати мусорного значения или чего похуже:

```
char x;
// ...
printf("bad input char: %s",x);
```

Однако, эти функции обеспечивают большую гибкость и знакомы программирующим на C.

Как обычно, `getchar()` позволяет знакомым способом читать символы из входного потока:

```
int i;:
while ((i=getchar())!=EOF) { // символьный ввод C
  // используем i
}
```

Обратите внимание: чтобы было законным сравнение с величиной `EOF` типа `int` при проверке на конец файла, результат `getchar()` надо помещать в переменную типа `int`, а не `char`.

За подробностями о вводе-выводе на C отсылаем к вашему руководству по C или книге Кернигана и Ритчи "Язык программирования C".

10.7 Упражнения

1. (*1.5) Читая файл вещественных чисел, составлять из пар прочитанных чисел комплексные числа, записать комплексные числа.
2. (*1.5) Определить тип `name_and_address` (тип_и_адрес). Определить для него << и >>. Написать программу копирования объектов потока `name_and_address`.
3. (*2) Разработать несколько функций для запроса и чтения данных разных типов. Предложения: целое, вещественное число, имя файла, почтовый адрес, дата, личная информация, и т.п. Попытайтесь сделать их устойчивыми к ошибкам.
4. (*1.5) Напишите программу, которая печатает: (1) строчные буквы,

(2) все буквы, (3) все буквы и цифры, (4) все символы, входящие в идентификатор в вашей версии C++, (5) все знаки пунктуации, (6) целые значения всех управляющих символов, (7) все обобщенные пробелы, (8) целые значения всех обобщенных пробелов, и, наконец, (9) все изображаемые символы.

5. (*4) Реализуйте стандартную библиотеку ввода-вывода C (<stdio.h>) с помощью стандартной библиотеки ввода-вывода C++ (<iostream.h>).
6. (*4) Реализуйте стандартную библиотеку ввода-вывода C++ (<iostream.h>) с помощью стандартной библиотеки ввода-вывода C (<stdio.h>).
7. (*4) Реализуйте библиотеки C и C++ так, чтобы их можно было использовать одновременно.
8. (*2) Реализуйте класс, для которого операция `[]` перегружена так, чтобы обеспечить произвольное чтение символов из файла.
9. (*3) Повторите упражнение 8, но добейтесь, чтобы операция `[]` была применима для чтения и для записи. Подсказка: пусть `[]` возвращает объект "дескриптор типа", для которого присваивание означает: присвоить через дескриптор файлу, а неявное приведение к типу `char` означает чтение файла по дескриптору.
10. (*2) Повторите упражнение 9, позволяя операции `[]` индексировать объекты произвольных типов, а не только символы.
11. (*3.5) Продумайте и реализуйте операцию форматного ввода. Используйте для задания формата строку спецификаций как в `printf()`. Должна быть возможность попыток применения нескольких спецификаций для одного ввода, чтобы найти требуемый формат. Класс форматного ввода должен быть производным класса `istream`.
12. (*4) Придумайте (и реализуйте) лучшие форматы ввода.
13. (**2) Определите для вывода манипулятор `based` с двумя параметрами: система счисления и целое значение, и печатайте целое в представлении, определяемом системой счисления. Например, `based(2,9)` напечатает 1001.
14. (**2) Напишите "миниатюрную" систему ввода-вывода, которая реализует классы `istream`, `ostream`, `ifstream`, `ofstream` и предоставляет функции, такие как operator<<() и operator>>() для целых, и операции, такие как `open()` и `close()` для файлов.

Используйте исключительные ситуации, а не переменные состояния, для сообщения об ошибках.

15. (**2) Напишите манипулятор, который включает и отключает эхо символа.

Проектирование и развитие

В этой главе обсуждаются подходы к разработке программного обеспечения. Обсуждение затрагивает как технические, так и социологические аспекты процесса развития программного обеспечения. Программа рассматривается как модель реальности, в которой каждый класс представляет определенное понятие. Ключевая задача проектирования состоит в определении доступной и защищенной частей интерфейса класса, исходя из которых определяются различные части программы. Определение этих интерфейсов есть итеративный процесс, обычно требующий экспериментирования. Упор делается на важной роли проектирования и организационных факторов в процессе развития программного обеспечения.

11.1 Введение

"Серебряной пули не существует."

Ф. Брукс

Создание любой нетривиальной программной системы - сложная и часто выматывающая задача. Даже для отдельного программиста собственно запись операторов программы есть только часть всей работы. Обычно анализ всей задачи, проектирование программы в целом, документация, тестирование, сопровождение и управление всем этим затмевает задачу написания и отладки отдельных частей программы. Конечно, можно все эти виды деятельности обозначить как "программирование" и затем вполне обоснованно утверждать: "Я не проектирую, я только программирую". Но как бы ни назывались отдельные виды деятельности, бывает иногда важно сосредоточиться на них по отдельности, так же как иногда бывает важно рассмотреть весь процесс в целом. Стремясь поскорее довести систему до поставки, нельзя упускать из вида ни детали, ни картину в целом, хотя довольно часто происходит именно это.

Эта лекция сосредоточена на тех частях процесса развития программы, которые не связаны с написанием и отладкой отдельных программных фрагментов. Обсуждение здесь менее точное и детальное, чем во всех

остальных частях книги, где рассматриваются конкретные черты языка или определенные приемы программирования. Это неизбежно, поскольку нет готовых рецептов создания хороших программ. Детальные рецепты "как" могут существовать только для определенных, хорошо разработанных областей применения, но не для достаточно широких областей приложения. В программировании не существует заменителей разума, опыта и вкуса. Следовательно, в этой лекции вы найдете только общие рекомендации, альтернативные подходы и осторожные выводы.

Сложность данной тематики связана с абстрактной природой программ и тем фактом, что приемы, применимые для небольших проектов (скажем, программа в 10000 строк, созданная одним или двумя людьми), не распространяются на средние или большие проекты. По этой причине иногда мы приводим примеры из менее абстрактных инженерных дисциплин, а не только из программирования. Не преминем напомнить, что "доказательство по аналогии" является мошенничеством, и аналогии служат здесь только в качестве примера. Понятия проектирования, формулируемые с помощью определенных конструкций C++, и поясняемые примерами, мы будем обсуждать в лекциях 12 и 13. Предложенные в этой лекции рекомендации, отражаются как в самом языке C++, так и в решении конкретных программных задач по всей книге.

Снова напомним, что в силу чрезвычайного разнообразия областей применения, программистов и среды, в которой развивается программная система, нельзя ожидать, что каждый вывод, сделанный здесь, будет прямо применим для вашей задачи. Эти выводы применимы во многих самых разных случаях, но их нельзя считать универсальными законами. Смотрите на них со здоровой долей скептицизма.

Язык C++ можно просто использовать как лучший вариант C. Однако, поступая так, мы не используем наиболее мощные возможности C++ и определенные приемы программирования на нем, так что реализуем лишь малую долю потенциальных достоинств C++. В этой лекции излагается такой подход к проектированию, который позволяет полностью использовать возможности абстрактных данных и средства объектного программирования C++. Такой подход обычно называют

объектно-ориентированным проектированием. В лекции 12 обсуждаются основные приемы программирования на C++, там же содержится предостережение от сомнительных идей, что есть только один "правильный" способ использования C++, и что для получения максимального выигрыша следует всякое средство C++ применять в любой программе (§ 12.1).

Укажем некоторые основные принципы, рассматриваемые в этой лекции:

- из всех вопросов, связанных с процессом развития программного обеспечения, самый важный - четко сознавать, что собственно вы пытаетесь создать.
- Успешный процесс развития программного обеспечения - это длительный процесс.
- Системы, которые мы создаем, стремятся к пределу сложности по отношению как к самим создателям, так и используемым средствам.
- Эксперимент является необходимой частью проекта для разработки всех нетривиальных программных систем.
- Проектирование и программирование - это итеративные процессы.
- Различные стадии проекта программного обеспечения, такие как: проектирование, программирование и тестирование - невозможно строго разделить.
- Проектирование и программирование нельзя рассматривать в отрыве от вопросов управления этими видами деятельности.

Недооценить любой из этих принципов очень легко, но обычно накладно. В то же время трудно воплотить эти абстрактные идеи на практике. Здесь необходим определенный опыт. Подобно построению лодки, езде на велосипеде или программированию, проектирование - это искусство, которым нельзя овладеть только с помощью теоретических занятий.

Может быть, все эти емкие принципы можно сжать в один: проектирование и программирование - виды человеческой деятельности; забудь про это - и все пропало.

Слишком часто мы забываем про это и рассматриваем процесс развития программного обеспечения просто как "последовательность хорошо определенных шагов, на каждом из которых по заданным правилам производятся некоторые действия над входными данными, чтобы получить требуемый результат". Сам стиль предыдущего предложения выдает присутствие человеческой природы!

Эта лекция относится к проектам, которые можно считать честолюбивыми, если учитывать ресурсы и опыт людей, создающих систему. Похоже, это в природе как индивидуума, так и организации - браться за проекты на пределе своих возможностей. Если задача не содержит определенный вызов, нет смысла уделять особое внимание ее проектированию. Такие задачи решаются в рамках уже устоявшейся структуры, которую не следует разрушать. Только если замахиваются на что-то амбициозное, появляется потребность в новых, более мощных средствах и приемах. Кроме того, существует тенденция у тех, кто "знает как делать", перепоручать проект новичкам, которые не имеют таких знаний.

Не существует "единственного правильного способа" для проектирования и создания всей системы. Я бы считал веру в "единственный правильный способ" детской болезнью, если бы этой болезнью слишком часто не заболевали и опытные программисты. Напомним еще раз: только по той причине, что прием успешно использовался в течение года для одного проекта, не следует, что он без всяких изменений окажется столь же полезен для другого человека или другой задачи. Всегда важно не иметь предубеждений.

Убеждение в том, что нет единственно верного решения, пронизывает весь проект языка C++, и, в основном, по этой причине в первом издании книги не было раздела, посвященного проектированию: я не хотел, чтобы его рассматривали как "манифест" моих личных симпатий. По этой же причине здесь, как и в лекциях 12 и 13, нет четко определенного взгляда на процесс развития программного обеспечения, скорее здесь просто дается обсуждение определенного круга, часто возникающих, вопросов и предлагаются некоторые решения, оказавшиеся полезными в определенных условиях.

За этим введением следует краткое обсуждение целей и средств

развития программного обеспечения в § 11.2, а дальше лекция распадается на две основных части:

- § 11.3 содержит описание процесса развития программного обеспечения.
- § 11.4 содержит некоторые практические рекомендации по организации этого процесса.

Взаимосвязь между проектированием и языком программирования обсуждается в лекции 12, а лекция 13 посвящена вопросам проектирования библиотек для C++.

Очевидно, большая часть рассуждений относится к программным проектам большого объема. Читатели, которые не участвуют в таких разработках, могут сидеть спокойно и радоваться, что все эти ужасы их миновали, или же они могут выбрать вопросы, касающиеся только их интересов. Нет нижней границы размера программы, начиная с которой имеет смысл заняться проектированием прежде, чем начать писать программу. Однако все-таки есть нижняя граница, начиная с которой можно использовать какие-либо методы проектирования. Вопросы, связанные с размером, обсуждаются в § 11.4.2.

Труднее всего в программных проектах бороться с их сложностью. Есть только один общий способ борьбы со сложностью: разделяй и властвуй. Если задачу удалось разделить на две подзадачи, которые можно решать в отдельности, то можно считать ее решенной за счет разделения более, чем наполовину. Этот простой принцип применим для удивительно большого числа ситуаций. В частности, использование модулей или классов при разработке программных систем позволяет разбить программу на две части: часть реализации и часть, открытую пользователю - которые связаны между собой (в идеале) вполне определенным интерфейсом. Это основной, внутренне присущий программированию, принцип борьбы со сложностью.

Подобно этому и процесс проектирования программы можно разбить на отдельные виды деятельности с четко определенным (в идеале) взаимодействием между людьми, участвующими в них. Это основной, внутренне присущий проектированию, принцип борьбы со сложностью

и подход к управлению людьми,занятыми в проекте.

В обоих случаях выделение частей и определение интерфейса между частями - это то место, где требуется максимум опыта и чутья. Такое выделение не является чисто механическим процессом, обычно оно требует проницательности, которая может появиться только в результате досконального понимания системы на различных уровнях абстракции (см. § 11.3.3, § 12.2.1 и § 13.3). Близорукий взгляд на программу или на процесс разработки программного обеспечения часто приводит к дефектной системе. Отметим, что как программы, так и программистов разделить просто. Труднее достигнуть эффективного взаимодействия между участниками по обе стороны границы, не нарушая ее и не делая взаимодействие слишком жестким.

Здесь предложен определенный подход к проектированию, а не полное формальное описание метода проектирования. Такое описание выходит за предметную область книги. Подход, предложенный здесь, можно применять с различной степенью формализации, и он может служить базой для различных формальных спецификаций. В то же время нельзя считать эту лекцию рефератом, и здесь не делается попытка рассмотреть каждую тему, относящуюся к процессу разработки программ или изложить каждую точку зрения. Это тоже выходит за предметную область книги. Реферат по этой тематике можно найти в [2]. В этой книге используется достаточно общая и традиционная терминология. Самые "интересные" термины, как: проектирование, прототип, программист - имеют в литературе несколько определений, часто противоречащих друг другу, поэтому предостерегаем вас от того, чтобы, исходя из принятых в вашем окружении определений терминов, вы не вынесли из книги то, на что автор совершенно не рассчитывал.

11.2 Цели и средства

Цель программирования - создать продукт, удовлетворяющий пользователя. Важнейшим средством для достижении этой цели является создание программы с ясной внутренней структурой и воспитание коллектива программистов и разработчиков, имеющих достаточный опыт и мотивацию, чтобы быстро и эффективно реагировать на все изменения.

Почему это так? Ведь внутренняя структура программы и процесс, с помощью которого она получена, в идеале никак не касаются конечного пользователя. Более того, если конечный пользователь почему-то интересуется тем, как написана программа, то что-то с этой программой не так. Почему, несмотря на это, так важны структура программы и люди, ее создавшие? В конце концов конечный пользователь ничего об этом не должен знать.

Ясная внутренняя структура программы облегчает:

- тестирование,
- переносимость,
- сопровождение,
- расширение,
- реорганизацию и
- понимание.

Главное здесь в том, что любая удачная большая программа имеет долгую жизнь, в течение которой над ней работают поколения программистов и разработчиков, она переносится на новую машину, приспосабливается к непредусмотренным требованиям и несколько раз перестраивается. Во все время жизни необходимо в приемлемое время и с допустимым числом ошибок выдавать версии программы. Не планировать все это - все равно, что запланировать неудачу.

Отметим, что, хотя в идеальном случае пользователи не должны знать внутреннюю структуру системы, на практике они обычно хотят ее знать. Например, пользователь может желать познакомиться в деталях с разработкой системы с целью научиться контролировать возможности и надежность системы на случай переделок и расширений. Если рассматриваемый программный продукт есть не полная система, а набор библиотек для получения программных систем, то пользователь захочет узнать побольше "деталей", чтобы они служили источником идей и помогали лучше использовать библиотеку.

Нужно уметь очень точно определить объем проектирования программы. Недостаточный объем приводит к бесконечному срезанию острых углов ("побыстрее передадим систему, а ошибку устраним в следующей версии"). Избыточный объем приводит к усложненному

описанию системы, в котором существенное теряется в формальностях, в результате чего при реорганизации программы получение работающей версии затягивается ("новая структура намного лучше старой, пользователь согласен ждать ради нее"). К тому же возникают такие потребности в ресурсах, которые непозволительны для большинства потенциальных пользователей. Выбор объема проектирования - самый трудный момент в разработке, именно здесь проявляется талант и опыт. Выбор трудно сделать и для одного программиста или разработчика, но он еще труднее для больших задач, где занято много людей разного уровня квалификации.

Организация должна создавать программный продукт и сопровождать его, несмотря на изменения в штате, в направлении работы или в управляющей структуре. Распространенный способ решения этих проблем заключался в попытке сведения процесса создания системы к нескольким относительно простым задачам, укладывающимся в жесткую структуру. Например, создать группу легко обучаемых (дешевых) и взаимозаменяемых программистов низкого уровня ("кодировщиков") и группу не таких дешевых, но взаимозаменяемых (а значит также не уникальных) разработчиков. Считается, что кодировщики не принимают решений по проектированию, а разработчики не утруждают себя "грязными" подробностями кодирования. Обычно такой подход приводит к неудаче, а где он срабатывает, получается слишком громоздкая система с плохими характеристиками.

Недостатки такого подхода состоят в следующем:

- слабое взаимодействие между программистами и разработчиками приводит к неэффективности, промедлению, упущенным возможностям и повторению ошибок из-за плохого учета и отсутствия обмена опытом;
- сужение области творчества разработчиков приводит к слабому профессиональному росту, безынициативности, небрежности и большой текучести кадров.

По сути, подобные системы - это бесполезная трата редких человеческих талантов. Создание структуры, в рамках которой люди могут найти применение разным талантам, овладеть новым родом

деятельности и участвовать в творческой работе - это не только благородное дело, но и практичное, коммерчески выгодное предприятие.

С другой стороны, нельзя создать систему, представить документацию по ней и бесконечно ее сопровождать без некоторой жесткой организационной структуры. Для чисто новаторского проекта хорошо начать с того, что просто найти лучших специалистов и позволить им решать задачу в соответствии с их идеями. Но по мере развития проекта требуется все больше планирования, специализации и строго определенного взаимодействия между занятыми в нем людьми. Под строго определенным понимается не математическая или автоматически верифицируемая запись (хотя это безусловно хорошо там, где возможно и применимо), а скорее набор указаний по записи, именованию, документации, тестированию и т.п. Но и здесь необходимо чувство меры. Слишком жесткая структура может мешать росту и затруднять совершенствование. Здесь подвергается проверке талант и опыт менеджера. Для отдельного работника аналогичная проблема сводится к определению, где нужно проявить смекалку, а где действовать по рецептам.

Можно рекомендовать планировать не на период до выдачи следующей версии системы, а на более долгий срок. Строить планы только до выпуска очередной версии - значит планировать неудачу. Нужно иметь организацию и стратегию развития программного обеспечения, которые нацелены на создание и поддержание многих версий разных систем, т.е. нужно многократное планирование успеха.

Цель проектирования в выработке ясной и относительно простой внутренней структуры программы, называемой иногда архитектурой, иными словами каркаса, в который укладываются отдельные программные фрагменты, и который помогает написанию этих фрагментов.

Проект - конечный результат процесса проектирования (если только бывает конечный продукт у итеративного процесса). Он является средоточием взаимодействий между разработчиком и программистом и между программистами. Здесь необходимо соблюсти чувство меры. Если я, как отдельный программист, проектирую небольшую программу,

которую собираюсь написать завтра, то точность и полнота описания проекта может свестись к нескольким каракулям на обратной стороне конверта. На другом полюсе находится система, над которой работают сотни программистов и разработчиков, и здесь могут потребоваться тома тщательно составленных спецификаций проекта на формальном или полуформальном языке. Определение нужной степени точности, детализации и формальности проектирования является уже само по себе нетривиальной технической и административной задачей.

Далее будет предполагаться, что проект системы записывается как ряд определений классов (в которых частные описания опущены как лишние детали) и взаимоотношений между ними. Это упрощение, т.к. конкретный проект может учитывать: вопросы параллельности, использование глобального пространства имен, использование глобальных функций и данных, построение программы для минимизации перетрансляции, устойчивость, многомашинный режим и т.п. Но при обсуждении на данном уровне детализации без упрощения не обойтись, а классы в контексте C++ являются ключевым понятием проектирования. Некоторые из указанных вопросов будут обсуждаться ниже, а те, которые прямо затрагивают проектирование библиотек C++, будут рассмотрены в лекции 13. Более подробное обсуждение и примеры определенных методов объектно-ориентированного проектирования содержатся в [2].

Мы сознательно не проводили четкого разделения анализа и проектирования, поскольку обсуждение их различий выходит за рамки этой книги, и оно зависит от применяемых методов проектирования. Главное в том, чтобы выбрать метод анализа, подходящий для метода проектирования, и выбрать метод проектирования, подходящий для стиля программирования и используемого языка.

11.3 Процесс развития

Процесс развития программного обеспечения - это итеративный и расширяющийся процесс. По мере развития каждая стадия повторяется многократно, и при всяком возврате на некоторую стадию процесса уточняется конечный продукт, получаемый на этой стадии. В общем случае процесс не имеет ни начала, ни конца, поскольку, проектируя и реализуя систему, вы начинаете, используя как базу другие проекты,

библиотеки и прикладные системы, в конце работы после вас остается описание проекта и программа, которые другие могут уточнять, модифицировать, расширять и переносить. Естественно конкретный проект имеет определенное начало и конец, и важно (хотя часто удивительно трудно) четко и строго ограничить время и область действия проекта. Но заявление, что вы начинаете с "чистого листа", может привести к серьезным проблемам для вас, также как и позиция, что после передачи окончательной версии - хоть потоп, вызовет серьезные проблемы для ваших последователей (или для вас в новой роли).

Из этого вытекает, что следующие разделы можно читать в любом порядке, поскольку вопросы проектирования и реализации могут в реальном проекте переплетаться почти произвольно. Именно, "проект" почти всегда подвергается перепроектированию на основе предыдущего проекта, определенного опыта реализации, ограничений, накладываемых сроками, мастерством работников, вопросами совместимости и т.п. Здесь основная трудность для менеджера или разработчика или программиста в том, чтобы создать такой порядок в этом процессе, который не препятствует усовершенствованиям и не запрещает повторные проходы, необходимые для успешного развития.

У процесса развития три стадии:

- Анализ: определение области задачи.
- Проектирование: создание общей структуры системы.
- Реализация: программирование и тестирование.

Не забудьте об итеративной природе этих процессов (неспроста стадии не были пронумерованы), и заметьте, что никакие важные аспекты процесса развития программы не выделяются в отдельные стадии, поскольку они должны допускать:

- Экспериментирование.
- Тестирование.
- Анализ проектирования и реализации.
- Документирование.
- Сопровождение.

Сопровождение программного обеспечения рассматривается просто как еще несколько проходов по стадиям процесса развития (см. также § 11.3.6).

Очень важно, чтобы анализ, проектирование и реализация не были слишком оторваны друг от друга, и чтобы люди, принимающие в них участие, были одного уровня квалификации для налаживания эффективных контактов.

В больших проектах слишком часто бывает иначе. В идеале, в процессе развития проекта работники должны сами переходить с одной стадии на другую: лучший способ передачи тонкой информации - это использовать голову работника. К сожалению, в организациях часто устанавливают барьеры для таких переходов, например, у разработчика может быть более высокий статус и (или) более высокий оклад, чем у "простого" программиста. Не принято, чтобы сотрудники ходили по отделам с целью набраться опыта и знаний, но пусть, по крайней мере, будут регулярными собеседования сотрудников, занятых на разных стадиях проекта.

Для средних и малых проектов обычно не делают различия между анализом и проектированием - эти стадии сливаются в одну. Для малых проектов также не разделяют проектирование и программирование. Конечно, тем самым решается проблема взаимодействия. Для данного проекта важно найти подходящую степень формализации и выдержать нужную степень разделения между стадиями (§ 11.4.2). Нет единственно верного способа для этого.

Приведенная здесь модель процесса развития программного обеспечения радикально отличается от традиционной модели "каскад" (waterfall). В последней процесс развития протекает линейно от стадии анализа до стадии тестирования. Основной недостаток модели каскад тот, что в ней информация движется только в одном направлении. Если выявлена проблема "ниже по течению", то возникает сильное методологическое и организационное давление, чтобы решить проблему на данном уровне, не затрагивая предыдущих стадий процесса. Отсутствие повторных проходов приводит к дефектному проекту, а в результате локального устранения проблем получается искаженная реализация. В тех неизбежных случаях, когда информация

должна быть передана назад к источнику ее получения и вызвать изменения в проекте, мы получим лишь слабое "колыхание" на всех уровнях системы, стремящейся подавить внесенное изменение, а значит система плохо приспособлена к изменениям. Аргумент в пользу "никаких изменений" или "только локальные изменения" часто сводится к тому, что один отдел не хочет перекладывать большую работу на другой отдел "ради их же блага".

Часто бывает так, что ко времени, когда ошибка уже найдена, исписано столько бумаги относительно ошибочного решения, что усилия, нужные на исправление документации, затмевают усилия для исправления самой программы. Таким образом, бумажная работа может стать главной проблемой процесса создания системы. Конечно, такие проблемы могут быть и возникают в процессе развития больших систем. В конце концов, определенная работа с бумагами необходима. Но выбор линейной модели развития (каскад) многократно увеличивает вероятность, что эта проблема выйдет из-под контроля.

Недостаток модели каскад в отсутствии повторных проходов и неспособности реагировать на изменения. Опасность предлагаемой здесь итеративной модели состоит в искушении заменить размышление и реальное развитие на последовательность бесконечных изменений. Тот и другой недостатки легче указать, чем устранить, и для того, кто организует работу, легко принять простую активность за реальный прогресс.

Вы можете уделять пристальное внимание деталям, использовать разумные приемы управления, развитую технологию, но ничто не спасет вас, если нет ясного понимания того, что вы пытаетесь создать. Больше всего проектов проваливалось именно из-за отсутствия хорошо сформулированных реалистичных целей, а не по какой-либо иной причине. Что бы вы не делали и чем бы не занимались, надо ясно представлять имеющиеся у вас средства, ставить достижимые цели и ориентиры и не искать технических решений социологических проблем. С другой стороны, надо применять только адекватную технологию, даже если она потребует затрат,- люди работают лучше, имея адекватные средства и приемлемую среду. Не заблуждайтесь, думая, что легко выполнить эти рекомендации.

11.3.1 Цикл развития

Процесс развития системы - это итеративная деятельность. Основной цикл сводится к повторяемым в следующей последовательности шагам:

1. Создать общее описание проекта.
2. Выделить стандартные компоненты.
    - [a] Подогнать компоненты под данный проект.
3. Создать новые стандартные компоненты.
    - [a] Подогнать компоненты под данный проект.
4. Составить уточненное описание проекта.

В качестве примера рассмотрим автомобильный завод. Проект должен начинаться с самого общего описания новой машины. Этот первый шаг базируется на некотором анализе и описании машины в самых общих терминах, которые скорее относятся к предполагаемому использованию, чем к характеристикам желаемых возможностей машины. Часто самой трудной частью проекта бывает выбор желаемых возможностей, или, точнее, определение относительно простого критерия выбора желаемых возможностей. Удача здесь, как правило, является результатом работы отдельного проницательного человека и часто называется предвидением. Слишком типично как раз отсутствие ясных целей, что приводит к неуверенно развивающимся или просто проваливающимся проектам.

Итак, допустим необходимо создать машину среднего размера с четырьмя дверцами и достаточно мощным мотором. Очевидно, что на первом этапе проекта не следует начинать проектирование машины (и всех ее компонентов) с нуля. Хотя программист или разработчик программного обеспечения в подобных обстоятельствах поступит именно так.

На первом этапе надо выяснить, какие компоненты доступны на вашем собственном складе и какие можно получить от надежных поставщиков. Найденные таким образом компоненты не обязательно в точности подойдут для новой машины. Всегда требуется подгонка компонентов. Может быть даже потребуется изменить характеристики "следующей версии" выбранных компонентов, чтобы сделать их пригодными для

проекта. Например, может существовать вполне пригодный мотор, вырабатывающий немного меньшую мощность.Тогда или вы, или поставщик мотора должны предложить, не изменяя общего описания проекта, в качестве компенсации дополнительный зарядный генератор. Заметим, что сделать это,"не изменяя общего описания проекта", маловероятно, если только само описание не приспособлено к определенной подгонке. Обычно подобная подгонка требует кооперации между вами и поставщиком моторов. Сходные вопросы возникают и у программиста или разработчика программного обеспечения. Здесь подгонку обычно облегчает эффективное использование производных классов. Но не рассчитывайте провести произвольные расширения в проекте без определенного предвидения или кооперации с создателем таких классов.

Когда исчерпается набор подходящих стандартных компонентов, проектировщик машины не спешит заняться проектированием новых оптимальных компонентов для своей машины. Это было бы слишком расточительно. Допустим, что не нашлось подходящего блока кондиционирования воздуха, зато есть свободное пространство, имеющее форму буквы L, в моторном отсеке. Возможно решение разработать блок кондиционирования указанной формы. Но вероятность того, что блок подобной странной формы будет использоваться в машинах другого типа (даже после значительной подгонки), крайне низка. Это означает, что наш проектировщик машины не сможет разделить затраты на производство такого блока с создателями машин другого типа, а значит время жизни этого блока коротко. Поэтому стоит спроектировать блок, который найдет более широкое применение, т.е. разработать разумный проект блока, более приспособленный для подгонки, чем наше L-образное чудище. Возможно, это потребует больших усилий, и даже придется для приспособления более универсального блока изменить общее описание проекта машины. Поскольку новый блок разрабатывался для более общего применения, чем наше L-образное чудище, предположительно, для него потребуется некоторая подгонка, чтобы полностью удовлетворить наши пересмотренные запросы.

Подобная же альтернатива возникает и у программиста или разработчика программного обеспечения: вместо того, чтобы создать программу, привязанную к конкретному проекту, разработчик может

спроектировать новую достаточно универсальную программу, которая будет иметь хорошие шансы стать стандартной в определенной области.

Наконец, когда мы прошлись по всем стандартным компонентам, составляется "окончательное" общее описание проекта. Несколько специально разработанных средств указываются как возможные. Вероятно, в следующем году придется для новой модели повторить наши шаги, и как раз эти специальные средства придется переделать или выбросить. Как ни печально, но опыт традиционно проектировавшихся программ показывает, что лишь несколько частей системы можно выделить в отдельные компоненты и лишь несколько из них пригодны вне данного проекта.

Мы не пытаемся утверждать, что все разработчики машин действуют столь разумно, как в приведенном примере, а разработчики программ совершают все указанные ошибки. Утверждается, что указанная методика разработки машин применима и для программного обеспечения. Так, в этой и следующей лекциях даны приемы использования ее для C++. Тем не менее можно сказать, что сама природа программирования способствует совершению указанных ошибок (§ 12.2.1 и § 12.2.5). В разделе 11.4.3 опровергается профессиональное предубеждение против использования описанной здесь модели проектирования.

Заметим, что модель развития программного обеспечения хорошо применима только в расчете на большие сроки. Если ваш горизонт сужается до времени выдачи очередной версии, нет смысла создавать и поддерживать функционирование стандартных компонентов. Это просто приведет к излишним накладным расходам. Наша модель рассчитана на организации со временем жизни, за которое проходит несколько проектов, и с размерами, которые позволяют нести дополнительные расходы и на средства проектирования, программирования, и на сопровождение проектов, и на повышение квалификации разработчиков, программистов и менеджеров. Фактически это эскиз некоторой фабрики по производству программ. Как ни удивительно, она только масштабом отличается от действий лучших программистов, которые для повышения своей производительности в течении лет накапливали запас приемов и методов проектирования, создавали инструменты и библиотеки.

Похоже, что большинство организаций просто не умеет воспользоваться достижениями лучших сотрудников, как из-за отсутствия предвидения, так и по неспособности применить эти достижения в достаточно широком объеме.

Все-таки неразумно требовать, чтобы "стандартные компоненты" были стандартными универсально. Существует лишь малое число международных стандартных библиотек, а в своем большинстве компоненты окажутся стандартными только в пределах страны, отрасли, компании, производственной цепочки, отдела или области приложения и т.д. Просто мир слишком велик, чтобы универсальный стандарт всех компонентов и средств был реальной или желанной целью проекта.

11.3.2 Цели проектирования

Каковы самые общие цели проектирования? Конечно, простота, но в чем критерий простоты? Поскольку мы считаем, что проект должен развиваться во времени, т.е. система будет расширяться, переноситься, настраиваться и, вообще, изменяться массой способов, которые невозможно предусмотреть, необходимо стремиться к такой системе проектирования и реализации, которая была бы простой с учетом, что она будет меняться многими способами. На самом деле, практично допустить, что сами требования к системе будут меняться неоднократно за период от начального проекта до выдачи первой версии системы.

Вывод таков: система должна проектироваться максимально простой при условии, что она будет подвергаться серии изменений. Мы должны проектировать в расчете на изменения, т.е. стремиться к

- гибкости,
- расширяемости и
- переносимости

Лучшее решение - выделить части системы, которые вероятнее всего будут меняться, в самостоятельные единицы, и предоставить программисту или разработчику гибкие возможности для модификаций таких единиц. Это можно сделать, если выделить ключевые для данной

задачи понятия и предоставить класс, отвечающий за всю информацию, связанную с отдельным понятием (и только с ним). Тогда изменение будет затрагивать только определенный класс. Естественно, такой идеальный способ гораздо легче описать, чем воплотить.

Рассмотрим пример: в задаче моделирования метеорологических объектов нужно представить дождевое облако. Как это сделать? У нас нет общего метода изображения облака, поскольку его вид зависит от внутреннего состояния облака, а оно может быть задано только самим облаком.

Первое решение: пусть облако изображает себя само. Оно подходит для многих ограниченных приложений. Но оно не является достаточно общим, поскольку существует много способов представления облака: детальная картина, набросок очертаний, пиктограмма, карта и т.п. Другими словами, вид облака определяется как им самим, так и его окружением.

Второе решение заключается в том, чтобы предоставить самому облаку для его изображения сведения о его окружении. Оно годится для большего числа случаев. Однако и это не общее решение. Если мы предоставляем облаку сведения об его окружении, то нарушаем основной постулат, который требует, чтобы класс отвечал только за одно понятие, и каждое понятие воплощалось определенным классом. Может оказаться невозможным предложить согласованное определение "окружения облака", поскольку, вообще говоря, как выглядит облако зависит от самого облака и наблюдателя. Чем представляется облако мне, сильно зависит от того, как я смотрю на него: невооруженным глазом, с помощью поляризационного фильтра, с помощью метеорадара и т.д. Помимо наблюдателя и облака следует учитывать и "общий фон", например, относительное положение солнца. К дальнейшему усложнению картины приводит добавление новых объектов типа других облаков, самолетов. Чтобы сделать задачу разработчика практически неразрешимой, можно добавить возможность одновременного существования нескольких наблюдателей.

Третье решение состоит в том, чтобы облако, а также и другие объекты, например, самолеты или солнце, сами описывали себя по отношению к наблюдателю. Такой подход обладает достаточной общностью, чтобы

удовлетворить большинство запросов. Однако, он может привести к значительному усложнению и большим накладным расходам при выполнении. Как, например, добиться того, чтобы наблюдатель понимал описания, произведенные облаком или другими объектами?

Даже эта модель будет, по всей видимости, не достаточной для таких предельных случаев, как графика с высокой степенью разрешимости. Я думаю, что для получения очень детальной картины нужен другой уровень абстракции.

Дождевые облака - это не тот объект, который часто встретишь в программах, но объекты, участвующие в различных операциях ввода и вывода, встречаются часто. Поэтому можно считать пример с облаком пригодным для программирования вообще и для разработки библиотек в частности. Логически схожий пример в C++ представляют манипуляторы, которые используются для форматирования вывода в потоковом вводе-выводе (§ 10.4.2). Заметим, что третье решение не есть "верное решение", это просто более общее решение. Разработчик должен сбалансировать различные требования системы, чтобы найти уровень общности и абстракции, пригодный для данной задачи в данной области. Золотое правило: для программы с долгим сроком жизни правильным будет самый общий уровень абстракции, который вам еще понятен и который вы можете себе позволить, но не обязательно абсолютно общий. Обобщение, выходящее за пределы данного проекта и понятия людей, в нем участвующих, может принести вред, т.е. привести к задержкам, неприемлемым характеристикам, неуправляемым проектам и просто к провалу.

Чтобы использование указанных методов было экономично и поддавалось управлению, проектирование и управление должно учитывать повторное использование, о чем говорится в § 11.4.1 и не следует совсем забывать об эффективности (см. § 11.3.7).

11.3.3 Шаги проектирования

Рассмотрим проектирование отдельного класса. Обычно это не лучший метод. Понятия не существуют изолированно, наоборот, понятие

определяется в связи с другими понятиями. Аналогично и класс не существует изолированно, а определяется совместно с множеством связанных между собой классов. Это множество часто называют библиотекой классов или компонентом. Иногда все классы компонента образуют единую иерархию, иногда это не так (см. § 12.3).

Множество классов компонента бывают объединены некоторым логическим условием, иногда это - общий стиль программирования или описания, иногда - предоставляемый сервис. Компонент является единицей проектирования, документации, права собственности и, часто, повторного использования.

Это не означает, что если вы используете один класс компонента, то должны разбираться во всех и уметь применять все классы компонента или должны подгружать к вашей программе модули всех классов компонента. В точности наоборот, обычно стремятся обеспечить, чтобы использование класса вело к минимуму накладных расходов: как машинных ресурсов, так и человеческих усилий. Но для использования любого класса компонента нужно понимать логическое условие, которое его определяет (можно надеяться, что оно предельно ясно изложено в документации), понимать соглашения и стиль, примененный в процессе проектирования и описания компонента, и доступный сервис (если он есть).

Итак, перейдем к способам проектирования компонента. Поскольку часто это непростая задача, имеет смысл разбить ее на шаги и, сконцентрировавшись на подзадачах, дать полное и последовательное описание. Обычно нет единственно правильного способа разбиения. Тем не менее, ниже приводится описание последовательности шагов, которая пригодилась в нескольких случаях:

1. Определить понятие / класс и установить основные связи между ними.
2. Уточнить определения классов, указав набор операций для каждого.
    - [a] Провести классификацию операций. В частности, уточнить необходимость построения, копирования и уничтожения.
    - [b] Убедиться в минимальности, полноте и удобстве.

3. Уточнить определения классов, указав их зависимость от других классов.
    - [a] Наследование.
    - [b] Использование зависимостей.
4. Определить интерфейсы классов.
    - [a] Поделить функции на общие и защищенные.
    - [b] Определить точный тип операций класса.

Отметим, что это шаги итеративного процесса. Обычно для получения проекта, который можно уверенно использовать для первичной реализации или повторной реализации, нужно несколько раз проделать последовательность шагов. Одним из преимуществ глубокого анализа и предложенной здесь абстракции данных оказывается относительная легкость, с которой можно перестроить взаимоотношения классов даже после программирования каждого класса. Хотя это никогда не бывает просто.

Далее следует приступить к реализации классов, а затем вернуться, чтобы оценить проект, исходя из опыта реализации. Рассмотрим эти шаги в отдельности.

11.3.3.1 Шаг 1: определение классов

Определите понятия/классы и установите основные связи между ними. Главное в хорошем проекте - прямо отразить какое-либо понятие "реальности", т.е. уловить понятие из области приложения классов, представить взаимосвязь между классами строго определенным способом, например, с помощью наследования, и повторить эти действия на разных уровнях абстракции. Но как мы можем уловить эти понятия? Как на практике решить, какие нам нужны классы?

Лучше поискать ответ в самой области приложения, чем рыться в программистском хранилище абстракций и понятий. Обратитесь к тому, кто стал экспертом по работе в некогда сделанной системе, а также к тому, кто стал критиком системы, пришедшей ей на смену. Запомните выражения того и другого.

Часто говорят, что существительные играют роль классов и объектов,

используемых в программе, это действительно так. Но это только начало. Далее, глаголы могут представлять операции над объектами или обычные (глобальные) функции, вырабатывающие новые значения, исходя из своих параметров, или даже классы. В качестве примера можно рассматривать функциональные объекты, описанные в § 10.4.2. Такие глаголы, как "повторить" или "совершить" (commit) могут быть представлены итеративным объектом или объектом, представляющим операцию выполнения программы в базах данных.

Даже прилагательные можно успешно представлять с помощью классов, например, такие, как "хранимый", "параллельный", "регистровый", "ограниченный". Это могут быть классы, которые помогут разработчику или программисту, задав виртуальные базовые классы, специфицировать и выбрать нужные свойства для классов, проектируемых позднее.

Лучшее средство для поиска этих понятий / классов - грифельная доска, а лучший метод первого уточнения - это беседа со специалистами в области приложения или просто с друзьями. Обсуждение необходимо, чтобы создать начальный жизнеспособный словарь терминов и понятийную структуру. Мало кто может сделать это в одиночку. Обратитесь к [1], чтобы узнать о методах подобных уточнений.

Не все классы соответствуют понятиям из области приложения. Некоторые могут представлять ресурсы системы или абстракции периода реализации (см. § 12.2.1).

Взаимоотношения, о которых мы говорим, естественно устанавливаются в области приложения или (в случае повторных проходов по шагам проектирования) возникают из последующей работы над структурой классов. Они отражают наше понимание основ области приложения. Часто они являются классификацией основных понятий. Пример такого отношения: машина с выдвижной лестницей есть грузовик, есть пожарная машина, есть движущееся средство.

В § 11.3.3.2 и § 11.3.3.5 предлагается некоторая точка зрения на классы и иерархию классов, если необходимо улучшить их структуру.

11.3.3.2 Шаг 2: определение набора операций

Уточните определения классов, указав набор операций для каждого. В действительности нельзя разделить процессы определения классов и выяснения того, какие операции для них нужны. Однако, на практике они различаются, поскольку при определении классов внимание концентрируется на основных понятиях, не останавливаясь на программистских вопросах их реализации, тогда как при определении операций прежде всего сосредотачивается на том, чтобы задать полный и удобный набор операций. Часто бывает слишком трудно совместить оба подхода, в особенности, учитывая, что связанные классы надо проектировать одновременно.

Возможно несколько подходов к процессу определения набора операций.

Предлагаем следующую стратегию:

1. Рассмотрите, каким образом объект класса будет создаваться, копироваться (если нужно) и уничтожаться.
2. Определите минимальный набор операций, который необходим для понятия, представленного классом.
3. Рассмотрите операции, которые могут быть добавлены для удобства записи, и включите только несколько действительно важных.
4. Рассмотрите, какие операции можно считать тривиальными, т.е. такими, для которых класс выступает в роли интерфейса для реализации производного класса.
5. Рассмотрите, какой общности именования и функциональности можно достигнуть для всех классов компонента.

Очевидно, что это - стратегия минимализма. Гораздо проще добавлять любую функцию, приносящую ощутимую пользу, и сделать все операции виртуальными. Но, чем больше функций, тем больше вероятность, что они не будут использоваться, наложат определенные ограничения на реализацию и затруднят эволюцию системы. Так, функции, которые могут непосредственно читать и писать в переменную состояния объекта из класса, вынуждают использовать единственный способ реализации и значительно сокращают

возможности перепроектирования. Такие функции снижают уровень абстракции от понятия до его конкретной реализации. К тому же добавление функций добавляет работы программисту и даже разработчику, когда он вернется к проектированию. Гораздо легче включить в интерфейс еще одну функцию, как только установлена потребность в ней, чем удалить ее оттуда, когда уже она стала привычной.

Причина, по которой мы требуем явного принятия решения о виртуальности данной функции, не оставляя его на стадию реализации, в том, что, объявив функцию виртуальной, мы существенно повлияем на использование ее класса и на взаимоотношения этого класса с другими. Объекты из класса, имеющего хотя бы одну виртуальную функцию, требуют нетривиального распределения памяти, если сравнить их с объектами из таких языков как C или Фортран. Класс с хотя бы одной виртуальной функцией по сути выступает в роли интерфейса по отношению к классам, которые "еще могут быть определены", а виртуальная функция предполагает зависимость от классов, которые "еще могу быть определены" (см. § 12.2.3)

Отметим, что стратегия минимализма требует, пожалуй, больших усилий со стороны разработчика.

При определении набора операций больше внимания следует уделять тому, что надо сделать, а не тому, как это делать.

Иногда полезно классифицировать операции класса по тому, как они работают с внутренним состоянием объектов:

- Базовые операции: конструкторы, деструкторы, операции копирования.
- Селекторы: операции, не изменяющие состояния объекта.
- Модификаторы: операции, изменяющие состояние объекта.
- Операции преобразований, т.е. операции, порождающие объект другого типа, исходя из значения (состояния) объекта, к которому они применяются.
- Повторители: операции, которые открывают доступ к объектам класса или используют последовательность объектов.

Это не есть разбиение на ортогональные группы операций. Например, повторитель может быть спроектирован как селектор или модификатор. Выделение этих групп просто предназначено помочь в процессе проектирования интерфейса класса. Конечно, допустима и другая классификация. Проведение такой классификации особенно полезно для поддержания непротиворечивости между классами в рамках одного компонента.

В языке C++ есть конструкция, помогающая заданию селекторов и модификаторов в виде функции-члена со спецификацией const и без нее. Кроме того, есть средства, позволяющие явно задать конструкторы, деструкторы и функции преобразования. Операция копирования реализуется с помощью операций присваивания и конструкторов копирования.

11.3.3.3 Шаг 3: указание зависимостей

Уточните определение классов, указав их зависимости от других классов. Различные виды зависимостей обсуждаются в § 12.2. Основными по отношению к проектированию следует считать отношения наследования и использования. Оба предполагают понимание того, что значит для класса отвечать за определенное свойство системы. Отвечать за что-либо не означает, что класс должен содержать в себе всю информацию, или, что его функции-члены должны сами проводить все необходимые операции. Как раз наоборот, каждый класс, имеющий определенный уровень ответственности, организует работу, перепоручая ее в виде подзадач другим классам, которые имеют меньший уровень ответственности. Но надо предостеречь, что злоупотребление этим приемом приводит к неэффективным и плохо понимаемым проектам, поскольку происходит размножение классов и объектов до такой степени, что вместо реальной работы производится только серия запросов на ее выполнение. То, что можно сделать в данном месте, следует сделать.

Необходимость учесть отношения наследования и использования на этапе проектирования (а не только в процессе реализации) прямо вытекает из того, что классы представляют определенные понятия. Отсюда также следует, что именно компонент (т.е. множество

связанных классов), а не отдельный класс, являются единицей проектирования.

11.3.3.4 Шаг 4: определение интерфейсов

Определите интерфейсы классов. На этой стадии проектирования не нужно рассматривать приватные функции. Вопросы реализации, возникающие на стадии проектирования, лучше всего обсуждать на шаге 3 при рассмотрении различных зависимостей. Более того, существует золотое правило: если класс не допускает по крайней мере двух существенно отличающихся реализаций, то что-то явно не в порядке с этим классом, это просто замаскированная реализация, а не представление абстрактного понятия. Во многих случаях для ответа на вопрос: "Достаточно ли интерфейс класса независим от реализации?"- надо указать, возможна ли для класса схема ленивых вычислений.

Отметим, что общие базовые классы и друзья (friend) являются частью общего интерфейса класса (см. § 5.4.1 и § 12.4). Полезным упражнением может быть определение раздельного интерфейса для классов-наследников и всех остальных классов с помощью разбиения интерфейса на общую и закрытые части.

Именно на этом шаге следует продумать и описать точные определения типов аргументов. В идеале желательно иметь максимальное число интерфейсов со статическими типами, относящимися к области приложения (см. § 12.1.3 и § 12.4).

При определении интерфейсов следует обратить внимание на те классы, где набор операций представлен более, чем на одном уровне абстракции. Например, в классе `file` у некоторых функций-членов аргументы имеют тип `file_descriptor` (дескриптор_файла), а у других аргументы - строка символов, которая обозначает имя файла. Операции с `file_descriptor` работают на другом уровне (меньшем) абстракции, чем операции с именем файла, так что даже странно, что они относятся к одному классу. Возможно, было бы лучше иметь два класса: один представляет понятие дескриптора файла, а другой - понятие имени файла. Обычно все операции класса должны представлять понятия одного уровня абстракции. Если это не так, то

стоит подумать о реорганизации и его, и связанных с ним классов.

11.3.3.5 Перестройка иерархии классов

Шаги 1 и 3 требуют исследования классов и их иерархии, чтобы убедиться, что они адекватно отвечают нашим требованиям. Обычно это не так, и приходится проводить перестройку для улучшения структуры, проекта или реализации.

Самая типичная перестройка иерархии классов состоит в выделении общей части двух классов в новый класс или в разбиении класса на два новых. В обоих случаях в результате получится три класса: базовый класс и два производных. Когда следует проводить такую перестройку? Каковы общие показания, что такая перестройка будет полезной?

К сожалению нет простого и универсального ответа на эти вопросы. Это и не удивительно, поскольку то, что предлагается, не является мелочью при реализации, а изменяет основные понятия системы. Важной и нетривиальной задачей является поиск общности среди классов и выделение общей части. Нет точного определения общности, но следует обращать внимание на общность для понятий системы, а не просто для удобства реализации. Указаниями, что два класса имеют нечто общее, что возможно выделить в общий базовый класс, служат схожие способы использования, сходство наборов операций, сходство реализаций и просто тот факт, что часто в процессе обсуждения проекта оба класса появляются одновременно. С другой стороны, если есть несколько наборов операций класса с различными способами использования, если эти наборы обеспечивают доступ к раздельным подмножествам объектов реализации, и, если класс возникает в процессе обсуждения несвязанных тем, то этот класс является явным кандидатом для разбиения на части.

В силу тесной связи между понятиями и классами проблемы перестройки иерархии классов высвечиваются на поверхности проблем именования классов и использования имен классов в процессе обсуждения проекта. Если имена классов и их упорядоченность, задаваемая иерархией классов, кажутся неудобными при обсуждении проекта, значит, по всей видимости, есть возможность улучшения

иерархии. Заметим, что подразумевается, что анализ иерархии классов лучше проводить не в одиночку. Если вы оказались в таком положении, когда не с кем обсудить проект, хорошим выходом будет попытаться составить учебное описание системы, используя имена классов.

11.3.3.6 Использование моделей

Когда пишешь статью, пытаешься найти подходящую для темы модель. Нужно не бросаться сразу печатать текст, а поискать статьи на сходные темы, вдруг найдется такая, которая может послужить отправной точкой. Если ею окажется моя собственная статья, то можно будет использовать даже куски из нее, изменяя по мере надобности другие части, и вводить новую информацию только там, где требует логика предмета. Таким образом, исходя из первого издания, написана эта книга. Предельный случай такого подхода - это написание открытки-формуляра, где просто нужно указать имя и, возможно, добавить пару строк для придания "личного" отношения. По сути такие открытки пишутся с указанием отличия от стандарта.

Во всех видах творческой деятельности использование существующих систем в качестве моделей для новых проектов является скорее правилом, а не исключением. Всегда, когда это возможно, проектирование и программирование должны основываться на предыдущих работах. Это сокращает степени свободы для разработчика и позволяет сосредоточить внимание на меньшем числе вопросов в заданное время. Начать большой проект "практически с нуля" - это может возбуждать, но правильнее будет употребить термин "опьянение", которое приведет к "пьяному блужданию" в множестве вариантов. Построение модели не накладывает каких-либо ограничений и не означает покорного следования ей, это просто освобождает разработчика от некоторых вопросов.

Заметим, что на самом деле использование моделей неизбежно, поскольку каждый проект синтезируется из опыта его разработчиков. Лучше, когда использование модели является явно сформулированным решением, тогда все допущения делаются явно, определяется общий словарь терминов, появляется начальный каркас проекта и увеличивается вероятность того, что у разработчиков есть общий

подход.

Естественно, что выбор начальной модели является важным решением, и обычно оно принимается только после поиска потенциальных моделей и тщательной оценки вариантов. Более того, во многих случаях модель подходит только при условии понимания того, что потребуются значительные изменения для воплощения ее идей в иной области приложения. Но проектирование программного обеспечения - тяжелый труд, и надо использовать любую помощь. Не следует отказываться от использования моделей из-за неоправданного пренебрежения к имитации. Имитация - не что иное, как форма искреннего восхищения, а, с учетом права собственности и авторского права, использование моделей и предшествующих работ в качестве источника вдохновения - допустимый способ для всех новаторских работ во всех видах деятельности. То, что было позволено Шекспиру, подходит и для нас. Некоторые обозначают использование моделей в процессе проектирования как "проектирование повторного использования".

11.3.4 Эксперимент и анализ

В начале честолюбивого проекта нам неизвестен лучший способ построения системы. Часто бывает так, что мы даже не знаем точно, что должна делать система, поскольку конкретные факты прояснятся только в процессе построения, тестирования и эксплуатации системы. Как задолго до создания законченной системы получить сведения, необходимые для понимания того, какие решения при проектировании окажутся существенными, и к каким последствиям они приведут?

Нужно проводить эксперименты. Конечно, нужен анализ проекта и его реализации, как только появляется пища для него. Преимущественно обсуждение вертится вокруг альтернатив при проектировании и реализации. За исключением редких случаев проектирование есть социальная активность, которая ведет по пути презентации и обсуждений. Часто самым важным средством проектирования оказывается простая грифельная доска; без нее идеи проекта, находящиеся в зародыше, не могут развиться и стать общим достоянием в среде разработчиков и программистов.

Похоже, что самый популярный способ проведения эксперимента сводится к построению прототипа, т.е. уменьшенной версии системы. Прототип не обязан удовлетворять характеристикам реальных систем, обычно в изобилии есть машинные ресурсы и программная поддержка, и в таких условиях программисты и разработчики становятся непривычно опытными, хорошо образованными и активными. Появляется цель - сделать работающий прототип как можно скорее, чтобы начать исследование вариантов проекта и способов реализации.

Такой подход, если применять его разумно, может привести к успеху. Но он также может служить оправданием неудачно сделанных систем. Дело в том, что уделяя особое внимание прототипу, можно прийти к смещению усилий от "исследование вариантов проекта" к "получение как можно скорее рабочей версии системы". Тогда быстро угаснет интерес к внутренней структуре прототипа ("ведь это только прототип"), а работа по проектированию будет вытесняться манипулированием с реализацией прототипа. Просчет заключается в том, что такая реализация может легко привести к системе, которая имеет вид "почти законченной", а по сути является пожирателем ресурсов и кошмаром для тех, кто ее сопровождает. В этом случае на прототип тратятся время и энергия, которые лучше приберечь для реальной системы. Для разработчиков и менеджеров есть искушение переделать прототип в конечный программный продукт, а "искусство настройки системы" отложить до выпуска следующей версии. Если идти таким путем, то прототипы отрицают все основы проектирования.

Сходная проблема возникает, если исследователи привязываются к тем средствам, которые они создали при построении прототипа, и забывают, что они могут оказаться непригодными для рабочей системы, и что свобода от ограничений и формальностей, к которой они привыкли, работая в небольшой группе, может оказаться невозможной в большом коллективе, бьющимся над устранением длинной цепи препятствий.

И в то же время создание прототипов может сыграть важную роль. Рассмотрим, например, проектирование пользовательского интерфейса. Для этой задачи внутренняя структура той части системы, которая прямо не общается с пользователем, обычно не важна, и использование прототипов - это единственный способ узнать, какова будет реакция

пользователя при работе с системой. Другим примером служат прототипы, прямо предназначенные для изучения внутренней структуры системы. Здесь уже интерфейс с пользователем может быть примитивным, возможна работа с моделью пользователей.

Использование прототипов - это способ экспериментирования. Ожидаемый результат - это более глубокое понимание целей, а не сам прототип. Возможно, сущность прототипа заключается в том, что он является настолько неполным, что может служить лишь средством для эксперимента, и его нельзя превратить в конечный продукт без больших затрат на перепроектирование и на другую реализацию. Оставляя прототип "неполным", мы тем самым переключаем внимание на эксперимент и уменьшаем опасность превращения прототипа в законченный продукт. Это также почти избавляет от искушения взять за основу проекта системы проект прототипа, при этом забывая или игнорируя те ограничения, которые внутренне присущи прототипу. После эксперимента прототип надо просто выбросить.

Не следует забывать о других способах проведения эксперимента, которые могут служить во многих случаях альтернативой созданию прототипа, и там, где они применимы, их использование предпочтительнее, поскольку они обладают большей точностью и требуют меньших затрат времени разработчика и ресурсов системы. Примерами могут служить математические модели и различные формы моделирования. По сути, существует бесконечная возрастающая последовательность, начиная от математических моделей, ко все более и более детальным способам моделирования, затем к прототипам, к частичным реализациям системы, вплоть до полной системы.

Это подводит к идее построения системы, исходя из начального проекта и реализации, и двигаясь путем повторного прохождения этапов проектирования и реализации. Это идеальная стратегия, но она предъявляет высокие требования к средствам проектирования и реализации, и в ней содержится определенный риск того, что программный объем, реализующий решения, принятые при начальном проектировании, в процессе развития вырастет до такой величины, что существенное улучшение проекта будет просто невозможно.

Похоже, что по крайней мере теперь такую стратегию применяют или в

проектах от малого до среднего размеров, т.е. там, где маловероятны переделки общего проекта, или же для перепроектирования и иной реализации после выдачи первоначальной версии системы, где указанная стратегия становится неизбежной.

Помимо экспериментов, предназначенных для оценки решений, принимаемых на этапе проектирования, источником получения полезной информации может быть анализ собственно проектирования и (или) реализации. Например, может оказаться полезным изучение различных зависимостей между классами (см. § 12.2), не следует забывать и о таких традиционных вспомогательных средствах реализации, как граф вызовов функций, оценка производительности и т.п.

Заметим, что спецификация (результат анализа системы) и проект могут содержать ошибки, как и реализация, и возможно, они даже больше подвержены ошибкам, т.к. являются менее точными, не могут быть проверены на практике и обычно не окружены такими развитыми средствами, как те, что служат для анализа и проверки реализации. Введение большей формализации в язык или запись, с помощью которой изложен проект, в какой-то степени облегчает использования этих средств для проектирования. Но, как сказано в § 12.1.1, это нельзя делать за счет ухудшения языка, используемого для реализации. К тому же формальная запись может сама стать источником трудностей и проблем. Это происходит, когда выбранная степень формализации плохо подходит для конкретных задач, когда строгость формализации превосходит математическую основу системы и квалификацию разработчиков и программистов, и когда формальное описание системы начинает расходиться с реальной системой, для которой оно предназначалось.

Заключение о необходимости опыта и о том, что проектирование неизбежно сопровождается ошибками и плохо поддержано программными средствами, служит основным доводом в пользу итеративной модели проектирования и реализации. Альтернатива - это линейная модель процесса развития, начиная с анализа и кончая тестированием, но она существенно дефектна, поскольку не допускает повторных проходов, исходя из опыта, полученного на различных этапах развития системы.

11.3.5 Тестирование

Программа, которая не прошла тестирование, не работает. Идеал, чтобы после проектирования и (или) верификации программа заработала с первого раза, недостижим для всех, за исключением самых тривиальных программ. Следует стремиться к идеалу, но не заблуждаться, что тестирование простое дело.

"Как проводить тестирование?" - на этот вопрос нельзя ответить в общем случае. Однако, вопрос "Когда начинать тестирование?" имеет такой ответ - на самом раннем этапе, где это возможно. Стратегия тестирования должна быть разработана как часть проекта и включена в реализацию, или, по крайней мере, разрабатываться параллельно с ними. Как только появляется работающая система, надо начинать тестирование. Откладывание тестирования до "проведения полной реализации" - верный способ выйти из графика или передать версию с ошибками.

Всюду, где это возможно, проектирование должно вестись так, чтобы тестировать систему было достаточно просто. В частности, имеет смысл средства тестирования прямо встраивать в систему. Иногда это не делается из-за боязни слишком объемных проверок на стадии выполнения, или из-за опасений, что избыточность, необходимая для полного тестирования, излишне усложнит структуры данных. Обычно такие опасения неоправданы, поскольку собственно программы проверки и дополнительные конструкции, необходимые для них, можно при необходимости удалить из системы перед ее поставкой пользователю. Иногда могут пригодится утверждения о свойствах программы (см. § 12.2.7).

Более важным, чем набор тестов, является подход, когда структура системы такова, что есть реальные шансы убедить себя и пользователей, что ошибки можно исключить с помощью определенного набора статических проверок, статического анализа и тестирования. Если разработана стратегия построения системы, устойчивой к ошибкам (см. § 9.8), стратегия тестирования обычно разрабатывается как вспомогательная.

Если вопросы тестирования полностью игнорируются на этапе проектирования, возникнут проблемы с тестированием, временем поставки и сопровождением системы. Лучше всего начать работать над стратегией тестирования с интерфейсов классов и их взаимозависимостей (как предлагается в § 12.2 и § 12.4).

Трудно определить необходимый объем тестирования. Однако, очевидно, что проблему представляет недостаток тестирования, а не его избыток. Сколько именно ресурсов в сравнении с проектированием и реализацией следует отвести для тестирования зависит от природы системы и методов ее построения. Однако, можно предложить следующее правило: отводить больше ресурсов времени и человеческих усилий на тестирование системы, чем на получения ее первой реализации.

11.3.6 Сопровождение

"Сопровождение программного обеспечения" - неудачный термин. Слово "сопровождение" предлагает неверную аналогию с аппаратурой. Программы не требуют смазки, не имеют движущихся частей, которые изнашиваются так, что требуют замены, у них нет трещин, в которые попадает вода, вызывая ржавчину. Программы можно воспроизводить в точности и передавать в течении минуты на длинные расстояния. Короче, программы это совсем не то, что аппаратура. (В оригинале: "Software is not hardware").

Деятельность, которая обозначается, как сопровождение программ, на самом деле, состоит из перепроектирования и повторной реализации, а значит входит в обычный цикл развития программного обеспечения. Если в проекте учтены вопросы расширяемости, гибкости и переносимости, то обычные задачи сопровождения решаются естественным образом.

Подобно тестированию задачи сопровождения не должны решаться вне основного направления развития проекта и их не следует откладывать на потом.

11.3.7 Эффективность

Д. Кнуту принадлежит утверждение "Непродуманная оптимизация - корень всех бед". Некоторые слишком хорошо убедились в справедливости этого и считают вредными все заботы об оптимизации. На самом деле вопросы эффективности надо все время иметь в виду во время проектирования и реализации. Это не означает, что разработчик должен заниматься задачами локальной оптимизации, только задача оптимизации на самом глобальном уровне должна его волновать.

Лучший способ добиться эффективности - это создать ясный и простой проект. Только такой проект может остаться относительно устойчивым на весь период развития и послужить основой для настройки системы с целью повышения производительности. Здесь важно избежать "гаргантюализма", который является проклятием больших проектов. Слишком часто люди добавляют определенные возможности системы "на всякий случай" (см. § 11.3.3.2 и § 11.4.3), удваивая, учетверяя размер выполняемой программы ради завитушек. Еще хуже то, что такие усложненные системы трудно поддаются анализу, и поэтому трудно отличить избыточные накладные расходы от необходимых и провести анализ и оптимизации на общем уровне. Оптимизация должна быть результатом анализа и оценки производительности системы, а не произвольным манипулированием с программным кодом, причем это особенно справедливо для больших систем, где интуиция разработчика или программиста не может служить надежным указателем в вопросах эффективности.

Важно избегать по сути неэффективных конструкций, а также таких конструкций, которые можно довести до приемлемого уровня выполнения, только затратив массу времени и усилий. По этой же причине важно свести к минимуму использование непереносимых по своей сути конструкций и средств, поскольку их наличие препятствует работе системы на других машинах (менее мощных, менее дорогих).

11.4 Управление проектом

Если только это имеет какой-то смысл, большинство людей делает то, что их поощряют делать. Так, в контексте программного проекта, если

менеджер поощряет определенные способы действий и наказывает за другие, редкие программисты или разработчики рискнут своим положением, встречая сопротивления или безразличия администрации, чтобы делать так, как они полагают нужным.

Организация, в которой считают своих программистов недоумками, очень скоро получит программистов, которые будут рады и способны действовать только как недоумки.

Отсюда следует, что менеджер должен поощрять такие структуры, которые соответствуют сформулированным целям проекта и реализации. Однако на практике слишком часто бывает иначе. Существенное изменение стиля программирования достижимо только при соответствующем изменении в стиле проектирования, кроме того, обычно и то и другое требует изменения в стиле управления. Мыслительная и организационная инерция слишком просто сводят все к локальным изменениям, хотя только глобальные изменения могут принести успех. Прекрасной иллюстрацией служит переход на язык с объектно-ориентированным программированием, например на C++, когда он не влечет за собой соответствующих изменений в методах проектирования, чтобы воспользоваться новыми возможностями языка (см. § 12.1), и, наоборот, когда переход на "объектно-ориентированное проектирование" не сопровождается переходом на язык реализации, который поддерживает этот стиль.

11.4.1 Повторное использование

Часто основной причиной перехода на новый язык или новый метод проектирования называют то, что это облегчает повторное использование программ или проекта. Однако, во многих организациях поощряют сотрудника или группу, когда они предпочитают изобретать колесо. Например, если производительность программиста измеряется числом строк программы, то будет ли он писать маленькие программы, работающие со стандартными библиотеками, за счет своего дохода и, может быть, положения? А менеджер, если он оплачивается пропорционально числу людей в его группе, будет ли он использовать программы, сделанные другими коллективами, если он может просто нанять еще пару программистов в свою группу? Компания может

получить правительственный контракт, в котором ее доход составляет фиксированный процент от расходов на проект, будет ли она сокращать свой доход за счет использования наиболее эффективных средств? Трудно обеспечить вознаграждение за повторное использование, но если администрация не найдет способов поощрения и вознаграждения, то его просто не будет.

Повторное использование является прежде всего социальным фактором. Повторное использование программы возможно при условии, что

1. она работает; нельзя использовать повторно, если это невозможно и в первый раз;
2. она понятна; здесь имеет значение структура программы, наличие комментариев, документации, руководства;
3. она может работать вместе с программами, которые не создавались специально с таким условием;
4. можно рассчитывать на ее сопровождение (или придется делать это самому, что обычно не хочется);
5. это выгодно (хотя можно и разделить расходы по разработке и сопровождению с другими пользователями) и, наконец;
6. ее можно найти.

К этому можно еще добавить, что компонент не является повторно используемым, пока кто-то действительно не сделал это. Обычно задача приспособления компонента к существующему окружению приводит к уточнению набора операций, обобщению его поведения, и повышению его способности адаптации к другим программам. Пока все это не проделано хотя бы один раз, неожиданные острые углы находятся даже у компонентов, которые тщательно проектировались и реализовывались.

Личный опыт подсказывает, что условия для повторного использования возникают только в том случае, когда находится конкретный человек, занятый этим вопросом. В маленьких группах это обычно бывает тот, кто случайно или запланированно оказывается хранителем общих библиотек или документации. В больших организациях это бывает группа или отдел, которые получают привилегию собирать, документировать, популяризировать и сопровождать программное обеспечение, используемое различными группами.

Нельзя недооценивать такие группы "стандартных компонентов".

Укажем, что в первом приближении, система отражает организацию, которая ее создала. Если в организации нет средств поощрения и вознаграждения кооперации и разделения труда, то и на практике они будут исключением. Группа стандартных компонентов должна активно предлагать свои компоненты. Обычная традиционная документация важна, но ее недостаточно. Помимо этого указанная группа должна предоставлять руководства и другую информацию, которая позволит потенциальному пользователю отыскать компонент и понять как он может ему помочь. Значит эта группа должна предпринимать действия, которые обычно связываются с системой образования и маркетинга. Члены группы компонентов должны всегда, когда это возможно, работать в тесном сотрудничестве с разработчиками из областей приложения. Только тогда они будут в курсе запросов пользователей и сумеют почуять возможности использования стандартного компонента в различных областях. Это является аргументом за использование такой группы в роли консультанта и в пользу внутренних поставок программ, чтобы информация из группы компонентов могла свободно распространяться.

Заметим, что не все программы должны быть рассчитаны на повторное использование, иными словами, повторное использование не является универсальным свойством. Сказать, что некоторый компонент может быть повторно использован, означает, что в рамках определенной структуры его повторное использование не потребует значительных усилий. Но в большинстве случаев перенос в другую структуру может потребовать большой работы. В этом смысле повторное использование сильно напоминает переносимость. Важно понимать, что повторное использование является результатом проектирования, ставившего такую цель, модификации компонентов на основе опыта и специальных усилий, предпринятых для поиска среди существующих компонентов кандидатов на повторное использование. Неосознанное использование средств языка или приемов программирования не может чудесным образом гарантировать повторное использование. Такие средства языка C++, как классы, виртуальные функции и шаблоны типа, способствуют проектированию, облегчающему повторное использование (значит делают его более вероятным), но сами по себе эти средства не гарантируют повторное использование.

11.4.2 Размер

Человек и организация склонны излишне радоваться тому, что они "действуют по правильной методе". В институтской среде это часто звучит как "развитие согласно строгим предписаниям". В обоих случаях здравый смысл становится первой жертвой страстного и часто искреннего желания внести улучшения. К несчастью, если здравого смысла не хватает, то ущерб, нанесенный неразумными действиями, может быть неограниченным.

Вернемся к этапам процесса развития, перечисленным в § 11.3, и к шагам проектирования, указанным в § 11.3.3. Относительно просто переработать эти этапы в точный метод проектирования, когда шаг точно определен, имеет хорошо определенные входные и выходные данные и полуформальную запись для задания входных и выходных данных. Можно составить протокол, которому должно подчиняться проектирование, создать средства, предоставляющие определенные удобства для записи и организации процесса. Далее, исследуя классификацию зависимостей, приведенную в § 12.2, можно постановить, что определенные зависимости являются хорошими, а другие следует считать плохими, и предоставить средства анализа, которые обеспечат проведение таких оценок во всех стадиях проекта. Чтобы завершить такую "стандартизацию" процесса создания программ, можно было бы ввести стандарты на документацию (в том числе правила на правописание и грамматику и соглашения о формате документации), а так же стандарты на общий вид программ (в том числе указания какие средства языка следует использовать, а какие нет, перечисление допустимых библиотек и тех, которые не нужно использовать, соглашения об именовании функций, типов, переменных, правила расположения текста программы и т.д.).

Все это может способствовать успеху проекта. По крайней мере, было бы явной глупостью, браться за проект системы, которая предположительно будет иметь порядка десяти миллионов строк текста, над которой будут работать сотни человек, и которую будут сопровождать тысячи человек в течении десятилетий, не имея достаточно хорошо определенного и строгого плана по всем перечисленным выше позициям.

К счастью, большинство систем не относится к этой категории. Тем не менее, если решено, что данный метод проектирования или следование указанным образцам в программировании и документации являются "правильными", то начинает оказываться давление, чтобы применять их повсеместно. В небольших проектах это приводит к нелепым ограничениям и большим накладным расходам. В частности, это может привести к тому, что мерой развития и успеха становится не продуктивная работа, а пересылка бумажек и заполнение различных бланков. Если это случится, то в таком проекте настоящих программистов и разработчиков вытеснят бюрократы.

Когда происходит такое нелепое злоупотребление методами проектирования (по всей видимости совершенно разумными), то неудача проекта становится оправданием отказа от практически всякой формализации процесса разработки программного обеспечения. Это, в свою очередь, ведет к такой путанице и таким провалам, которые как раз и должен был предотвратить надлежащий метод проектирования.

Основная проблема состоит в определении степени формализации, пригодной для процесса развития конкретного проекта. Не рассчитывайте легко найти ее решение. По сути для малого проекта каждый метод может сработать. Еще хуже то, что похоже практически каждый метод, даже если он плохо продуман и жесток по отношению к исполнителям, может сработать для большого проекта, если вы готовы затратить уйму времени и денег.

В процессе развития программного обеспечения главная задача - сохранить целостность проекта. Трудность этой задачи зависит нелинейно от размера проекта. Сформулировать и сохранить основные установки в большом проекте может только один человек или маленькая группа. Большинство людей тратит столько времени на решение подзадач, технические детали, повседневную административную работу, что общие цели проекта легко забывает или заменяет их на более локальные и близкие цели. Верный путь к неудаче, когда нет человека или группы с прямым заданием следить за целостностью проекта. Верный путь к неудаче, когда у такого человека или группы нет средств воздействовать на проект в целом.

Отсутствие согласованных дальних целей намного более опасно для

проекта и организации, чем отсутствие какого-либо одного конкретного свойства. Небольшая группа людей должна сформулировать такие общие цели, постоянно держать их в уме, составить документы, содержащие самое общее описание проекта, составить пояснения к основным понятиям, и вообще, помогать всем остальным помнить о назначении проекта.

11.4.3 Человеческий фактор

Описанный здесь метод проектирования рассчитан на искусных разработчиков и программистов, поэтому от их подбора зависит успех организации.

Менеджеры часто забывают, что организация состоит из индивидуумов. Распространено мнение, что программисты равны и взаимозаменяемы. Это заблуждение может погубить организацию за счет вытеснения многих самых активных сотрудников и принуждения остальных работать над задачами значительно ниже их уровня. Индивидуумы взаимозаменяемы только, если им не дают применить свой талант, который поднимает их над общим минимальным уровнем, необходимым для решения данной задачи. Поэтому миф о взаимозаменяемости бесчеловечен и по сути своей расточителен.

Многие системы оценок производительности программиста поощряют расточительность и не могут учесть существенный личный вклад человека. Самым очевидным примером служит широко распространенная практика оценивать успех в количестве запрограммированных строк, выданных страниц документации, пройденных тестов и т.п. Такие цифры эффектно выглядят на диаграммах, но имеют самое отдаленное отношение к действительности. Например, если производительность измерять числом запрограммированных строк, то удачное повторное использование ухудшит оценку труда программиста. Обычно тот же эффект будет иметь удачное применение лучших приемов в процессе перепроектирования большой части системы.

Качество результата измерить значительно труднее, чем количество, и вознаграждать исполнителя или группу следует за качество их труда, а

не на основе грубых количественных оценок. К сожалению, насколько известно, практическая разработка способов оценки качества еще не началась. К тому же оценки, которые неполно описывают состояние проекта, могут исказить процесс его развития. Люди приспосабливаются, чтобы уложиться в отведенный срок и перестраивают свою работу в соответствии с оценками производительности, в результате страдает общая целостность системы и ее производительность. Например, если отведен срок для выявления определенного числа ошибок, то для того, чтобы уложиться в него, активно используют проверки на стадии выполнения, что ухудшает производительность системы. Обратно, если учитываются только характеристики системы на стадии выполнения, то число невыявленных ошибок будет расти при условии недостатка времени у исполнителей. Отсутствие хороших и разумных оценок качества повышает требования к технической квалификации менеджеров, иначе будет постоянная тенденция поощрять произвольную активность, а не реальный прогресс. Не надо забывать, что менеджеры тоже люди, и они должны по крайней мере настолько разбираться в новых технологиях, как и те, кем они управляют.

Здесь, как и в других аспектах процесса развития программного обеспечения, следует рассматривать большие временные сроки. По сути невозможно указать производительность человека на основе его работы за год. Однако, многие сотрудники имеют карточку своих достижений за большой период, и она может послужить надежным указанием для предсказания их производительности. Если не принимать во внимание такие карточки, что и делается, когда сотрудников считают взаимозаменяемыми спицами в колесе организации, то у менеджера остаются только вводящие в заблуждения количественные оценки.

Если мы рассматриваем только достаточно большие временные сроки и отказываемся от методов управления, рассчитанных на "взаимозаменяемых недоумков", то надо признать, что индивидууму (как разработчику или программисту, так и менеджеру) нужен большой срок, чтобы дорасти до более интересной и важной работы. Такой подход не одобряет как "скакание" с места на место, так и передачу работы другому из-за карьерных соображений. Целью должен быть низкий оборот ключевых специалистов и ключевых менеджеров. Никакой менеджер не добьется успеха без подходящих технических знаний и

взаимопонимания с основными разработчиками и программистами. В то же время, в конечном счете никакая группа разработчиков или программистов не добьется успеха без поддержки компетентных менеджеров и без понимания хотя бы основных нетехнических вопросов, касающихся окружения, в котором они работают.

Когда требуется предложить нечто новое, на передний план выходят основные специалисты - аналитики, разработчики, программисты. Именно они должны решить трудную и критическую задачу внедрения новой технологии. Это те люди, которые должны овладеть новыми методами и во многих случаях забыть старые привычки. Это не так легко. Ведь эти люди сделали большой личный вклад в создание старых методов и свою репутацию как специалиста обосновывают успехами, полученными с помощью старых методов. Так же обстоит дело и с многими менеджерами.

Естественно у таких людей есть страх перед изменениями. Он может привести к преувеличению проблем, возникающих при изменениях, и к нежеланию признать проблемы, вызванные старыми методами. Естественно, с другой стороны люди, выступающие за изменения, могут переоценивать выгоды, которые принесут изменения, и недооценивать возникающие здесь проблемы. Эти две группы людей должны общаться, они должны научиться говорить на одном языке и должны помочь друг другу разработать подходящую схему перехода. Альтернативой будет организационный паралич и уход самых способных людей из обоих групп. Тем и другим следует знать, что самые удачливые из "старых ворчунов" могли быть "молодыми львами" в прошлом году, и если человеку дали возможность научиться без всяких издевательств, то он может стать самым стойким и разумным сторонником перемен. Он будет обладать неоценимыми свойствами здорового скептицизма, знания пользователей и понимания организационных препятствий. Сторонники немедленных и радикальных изменений должны осознать, что гораздо чаще нужен переход, предполагающий постепенное внедрение новых методов. С другой стороны, те, кто не желает перемен, должны поискать для себя такие области, где это возможно, чем вести ожесточенные, арьергардные бои в той области, где новые требования уже задали совершенно иные условия для успешного проекта.

11.5 Свод правил

В этой лекции мы затронули много тем, но как правило не давали настоятельных и конкретных рекомендаций по проектированию. Это соответствует моему убеждению, что нет "единственно верного решения". Принципы и приемы следует применять тем способом, который лучше подходит для конкретных задач. Для этого нужен вкус, опыт и разум. Все-таки можно указать некоторый свод правил, который разработчик может использовать в качестве ориентиров, пока не наберется достаточно опыта, чтобы выработать лучшие. Ниже приведен свод таких правил.

Эти правила можно использовать в качестве отправной точки в процессе выработки основных направлений для проекта или организации или в качестве проверочного списка. Подчеркну еще раз, что они не являются универсальными правилами и не могут заменить размышления.

- Узнайте, что вам предстоит создать.
- Ставьте определенные и осязаемые цели.
- Не пытайтесь с помощью технических приемов решить социальные проблемы.
- Рассчитывайте на большой срок
  - в проектировании, и
  - управлении людьми.
- Используйте существующие системы в качестве моделей, источника вдохновения и отправной точки.
- Проектируйте в расчете на изменения:
  - гибкость,
  - расширяемость,
  - переносимость, и
  - повторное использование.
- Документируйте, предлагайте и поддерживайте повторно используемые компоненты.
- Поощряйте и вознаграждайте повторное использование
  - проектов,
  - библиотек, и
  - классов.

- Сосредоточьтесь на проектировании компоненты.
  - Используйте классы для представления понятий.
  - Определяйте интерфейсы так, чтобы сделать открытым минимальный объем информации, требуемой для интерфейса.
  - Проводите строгую типизацию интерфейсов всегда, когда это возможно.
  - Используйте в интерфейсах типы из области приложения всегда, когда это возможно.
- Многократно исследуйте и уточняйте как проект, так и реализацию.
- Используйте лучшие доступные средства для проверки и анализа
  - проекта, и
  - реализации.
- Экспериментируйте, анализируйте и проводите тестирование на самом раннем возможном этапе.
- Стремитесь к простоте, максимальной простоте, но не сверх того.
- Не разрастайтесь, не добавляйте возможности "на всякий случай".
- Не забывайте об эффективности.
- Сохраняйте уровень формализации соответствующим размеру проекта.
- Не забывайте, что разработчики, программисты и даже менеджеры остаются людьми.

Еще некоторые правила можно найти в § 12.5

11.6 Список литературы с комментариями

В этой лекции мы только поверхностно затронули вопросы проектирования и управления программными проектами. По этой причине ниже предлагается список литературы с комментариями. Значительно более обширный список литературы с комментариями можно найти в [2].

1. Bruce Anderson and Sanjiv Gossain: An Iterative Design Model for Reusable Object-Oriented Software. Proc. OOPSLA'90. Ottawa, Canada. pp. 12-27. Описание модели итеративного проектирования и повторного проектирования с некоторыми

примерами и обсуждением результатов.

2. Grady Booch: Object Oriented Design. Benjamin Cummings. 1991. В этой книге есть детальное описание проектирования, определенный метод проектирования с графической формой записи и несколько больших примеров проекта, записанных на различных языках. Это превосходная книга, которая во многом повлияла на эту лекцию. В ней более глубоко рассматриваются многие из затронутых здесь вопросов.
3. Fred Brooks: The Mythical Man Month. Addison Wesley. 1982. Каждый должен перечитывать эту книгу раз в пару лет. Предостережение от высокомерия. Она несколько устарела в технических вопросах, но совершенно не устарела во всем, что касается отдельного работника, организации и вопросов размера.
4. Fred Brooks: No Silver Bullet. IEEE Computer, Vol.20 No.4. April 1987. Сводка различных подходов к процессу развития больших программных систем с очень полезным предостережением от веры в магические рецепты ("золотая пуля").
5. De Marco and Lister: Peopleware. Dorset House Publishing Co. 1987. Одна из немногих книг, посвященных роли человеческого фактора в производстве программного обеспечения. Необходима для каждого менеджера. Достаточно успокаивающая для чтения перед сном. Лекарство от многих глупостей.
6. Ron Kerr: A Materialistic View of the Software "Engineering" Analogy. in SIGPLAN Notices, March 1987. pp 123-125. Использование аналогии в этой и следующей лекциях во многом обязано наблюдениям из указанной статьи, а так же беседам с Р. Керром, которые этому предшествовали.
7. Barbara Liskov: Data Abstraction and Hierarchy. Proc. OOPSLA'87 (Addendum). Orlando, Florida. pp 17-34. Исследуется как использование наследования может повредить концепции абстрактных данных. Укажем, что в C++ есть специальные языковые средства, помогающие избежать большинство указанных проблем (§ 12.2.5).
8. C. N. Parkinson: Parkinson's Law and other Studies in Administration. Houghton-Mifflin. Boston. 1957. Одно из забавных и самых язвительных описаний бед, к которым приводит процесс администрирования.
9. Bertrand Meyer: Object Oriented Software Construction. Prentice Hall. 1988. Страницы 1-64 и 323-334 содержат хорошее описание

одного взгляда на объектно-ориентированное программирование и проектирование, а также много здравых, практических советов. В остальной части книги описывается язык Эйффель (Eiffel).

10. Alan Snyder: Encapsulation and Inheritance in Object-Oriented Programming Languages. Proc. OOPSLA'86. Portland, Oregon. pp.38-45. Возможно первое хорошее описание взаимодействия оболочки и наследования. В статье так же на хорошем уровне рассматриваются некоторые понятия, связанные с множественным наследованием.
11. Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener: Designing Object-Oriented Software. Prentice Hall. 1990. Описывается антропоморфный метод проектирования, основанный на специальных карточках CRC (Classes, Responsibilities, Collaboration) (т.е. Классы, Ответственность, Сотрудничество). Текст, а может быть и сам метод тяготеет к языку Smalltalk.

Проектирование и C++

Эта лекция посвящена связи между проектированием и языком программирования C++. В ней исследуется применение классов при проектировании и указываются определенные виды зависимостей, которые следует выделять как внутри класса, так и между классами. Изучается роль статического контроля типов. Исследуется применение наследования и связь наследования и принадлежности. Обсуждается понятие компонента и даются некоторые образцы для интерфейсов.

12.1 Проектирование и язык программирования.

Стремись к простоте, максимальной простоте, но не сверх того.

А. Эйнштейн

Если бы мне надо было построить мост, то я серьезно подумал бы, из какого материала его строить, и проект моста сильно зависел бы от выбранного материала, а, следовательно, разумные проекты каменного моста отличаются от разумных проектов металлического моста или от разумных проектов деревянного моста и т.д. Не стоит рассчитывать на выбор подходящего для моста материала без определенных знаний о материалах и их использовании. Конечно, вам не надо быть специалистом плотником для проектирования деревянного моста, но вы должны знать основы конструирования из дерева, чтобы предпочесть его металлу в качестве материала для моста. Более того, хотя для проектирования деревянного моста вы и не должны быть специалистом плотником, вам необходимо достаточно детально знать свойства дерева и еще больше знать о плотниках.

Аналогично, при выборе языка программирования для определенного программного обеспечения надо знать несколько языков, а для успешного проектирования программы надо достаточно детально знать выбранный язык реализации, даже если вам лично не предстоит написать ни одной строчки программы. Хороший проектировщик моста ценит свойства используемых им материалов и применяет их для улучшения проекта. Аналогично, хороший разработчик программ использует сильные стороны языка реализации и, насколько возможно,

стремится избежать такого его использования, которое вызовет трудности на стадии реализации.

Можно подумать, что так получается естественным образом, если в проектировании участвует только один разработчик или программист, однако даже в этом случае программист в силу недостатка опыта или из-за неоправданной приверженности к стилю программирования, рассчитанному на совершенно другие языки, может сбиться на неверное использование языка. Если разработчик существенно отличается от программиста, особенно если у них разная программистская культура, возможность появления в окончательной версии системы ошибок, неэффективных и неэлегантных решений почти наверняка превратится в неизбежность.

Итак, чем может помочь разработчику язык программирования? Он может предоставить такие языковые средства, которые позволят выразить прямо на языке программирования основные понятия проекта. Тогда облегчается реализация, проще поддерживать ее соответствие проекту, проще организовать общение между разработчиками и программистами, и появляется возможность создать более совершенные средства как для разработчиков, так и для программистов.

Например, многие методы проектирования уделяют значительное внимание зависимостям между различными частями программы (обычно с целью их уменьшения и гарантии того, что эти части будут понятны и хорошо определены). Язык, допускающий явное задание интерфейсов между частями программы, может помочь в этом вопросе разработчикам. Он может гарантировать, что действительно будут существовать только предполагаемые зависимости. Поскольку большинство зависимостей явно выражено в программе на таком языке, можно разработать средства, читающие программу и выдающие графы зависимостей. В этом случае разработчику и другим исполнителям легче уяснить структуру программы. Такие языки программирования как C++ помогают сократить разрыв между проектом и программой, а значит уменьшают возможность путаницы и недопониманий.

Базовое понятие C++ - это класс. Класс имеет определенный тип. Кроме того, класс является первичным средством упрятывания

информации. Можно описывать программы в терминах пользовательских типов и иерархий этих типов. Как встроенные, так и пользовательские типы подчиняются правилам статического контроля типов. Виртуальные функции предоставляют, не нарушая правил статических типов, механизм связывания на этапе выполнения. Шаблоны типа позволяют создавать параметризованные типы. Особые ситуации позволяют сделать регулярной реакцию на ошибки. Все эти средства C++ можно использовать без дополнительных накладных расходов в сравнении с программой на C. Таковы главнейшие средства C++, которые должен представлять и учитывать разработчик. Кроме того, существенно повлиять на принятие решений на стадии проектирования может наличие доступных больших библиотек следующего назначения: для работы с матрицами, для связи с базами данных, для поддержки параллельного программирования, графические библиотеки и т.д.

Страх перед новизной, непригодный здесь опыт работы на других языках, в других системах или областях приложения, бедные средства проектирования - все это приводит к неоптимальному использованию C++. Следует отметить три момента, когда разработчику не удается извлечь выгоду из возможностей C++ и учесть ограничения языка:

1. Игнорирование классов и составление проекта таким образом, что программистам приходится ограничиваться только C.
2. Игнорирование производных классов и виртуальных функций, использование только подмножества абстрактных данных.
3. Игнорирование статического контроля типов и составление проекта таким образом, что программисты вынуждены применять динамические проверки типов.

Обычно указанные моменты возникают у разработчиков, связанных с:

1. C, или традиционной системой CASE или методами структурного проектирования;
2. Адой или методами проектирования с помощью абстракции данных;
3. языками, близкими Smalltalk или Lisp.

В каждом случае следует решить: неправильно выбран язык реализации

(считая, что метод проектирования выбран верно), или разработчику не удалось приспособиться и оценить язык (считая, что язык реализации выбран верно).

Следует сказать, что нет ничего необычного или позорного в таком расхождении. Просто это расхождение, которое приведет к неоптимальному проекту, возложит дополнительную работу на программистов, а в случае, когда структура понятий проекта значительно беднее структуры языка C++, то и на самих разработчиков.

Отметим, что необязательно все программы должны структурироваться опираясь на понятия классов и (или) иерархий классов, и необязательно всякая программа должна использовать все средства, предоставляемые C++. Как раз наоборот, для успеха проекта необходимо, чтобы людям не навязывали использование языковых средств, с которыми они только познакомились. Цель последующего изложения не в том, чтобы навязать догматичное использование классов, иерархий и строго типизированных интерфейсов, а в том, чтобы показать возможности их использования всюду, где позволяет область приложения, ограничения C++ и опыт исполнителей. В § 12.1.4 будут рассмотрены подходы к различному использованию C++ в проекте под заголовком "Проект-гибрид".

12.1.1 Игнорирование классов

Рассмотрим первый из указанных моментов - игнорирование классов. В таком случае получившаяся программа на C++ будет приблизительно эквивалентна C-программе, разработанной по тому же проекту, и, можно сказать, что они будут приблизительно эквивалентны программам на Аде или Коболе, разработанным по нему же.

По сути проект составлен как не зависящий от языка реализации, что принуждает программиста ограничиваться общим подмножеством языков C, Ада или Кобол. Здесь есть свои преимущества. Например, получившееся в результате строгое разделение данных и программного кода позволяет легко использовать традиционные базы данных, которые разработаны для таких программ. Поскольку используется ограниченный язык программирования, от программистов требуется

меньше опытности (или, по крайней мере другой ее уровень). Для многих приложений, например, для традиционных баз данных, работающих с файлом последовательно, такой подход вполне разумен, а традиционные приемы, отработанные за десятилетия, вполне адекватны задаче.

Однако там, где область приложения существенно отличается от традиционной последовательной обработки записей (или символов), или сложность задачи выше, как, например, в диалоговой системе CASE, недостаток языковой поддержки абстрактных данных из-за отказа от классов (если их не учитывать) повредит проекту. Сложность задачи не уменьшится, но, поскольку система реализована на обедненном языке, структура программы плохо будет отвечать проекту. У нее слишком большой объем, не хватает проверки типов, и, вообще, она плохо приспособлена для использования различных вспомогательных средств. Это путь, приводящий к кошмарам при ее сопровождении.

Обычно для преодоления указанных трудностей создают специальные средства, поддерживающие понятия, используемые в проекте. Благодаря им создаются конструкции более высокого уровня и организуются проверки с целью компенсировать дефекты (или сознательное обеднение) языка реализации. Так метод проектирования становится самоцелью, и для него создается специальный язык программирования. Такие языки программирования в большинстве случаев являются плохой заменой широко распространенных языков программирования общего назначения, которые сопровождаются подходящими средствами проектирования. Использовать C++ с таким ограничением, которое должно компенсироваться при проектировании специальными средствами, бессмысленно. Хотя несоответствие между языком программирования и средствами проектирования может быть просто стадией процесса перехода, а значит временным явлением.

Самой типичной причиной игнорирования классов при проектировании является простая инерция. Традиционные языки программирования не предоставляют понятия класса, и в традиционных методах проектирования отражается этот недостаток. Обычно в процессе проектирования наибольшее внимание уделяется разбиению задачи на процедуры, производящие требуемые действия. В

лекции 1 это понятие называлось процедурным программированием, а в области проектирования оно именуется как функциональная декомпозиция. Возникает типичный вопрос "Можно ли использовать C++ совместно с методом проектирования, базирующимся на функциональной декомпозиции?" Да, можно, но, вероятнее всего, в результате вы придете к использованию C++ как просто улучшенного C со всеми указанными выше проблемами. Это может быть приемлемо на период перехода на новый язык, или для уже завершенного проектирования, или для подзадач, в которых использование классов не дает существенных выгод (если учитывать опыт программирования на C++ к данному моменту), но в общем случае на большом отрезке времени отказ от свободного использования классов, связанный с методом функциональной декомпозиции, никак не совместим с эффективным использованием C++.

Процедурно-ориентированный и объектно-ориентированный подходы к программированию различаются по своей сути и обычно ведут к совершенно разным решениям одной задачи. Этот вывод верен как для стадии реализации, так и для стадии проектирования: вы концентрируете внимание или на предпринимаемых действиях, или на представляемых сущностях, но не на том и другом одновременно.

Тогда почему метод объектно-ориентированного проектирования предпочтительнее метода функциональной декомпозиции? Главная причина в том, что функциональная декомпозиция не дает достаточной абстракции данных. А отсюда уже следует, что проект будет

- менее податливым к изменениям,
- менее приспособленным для использования различных вспомогательных средств,
- менее пригодным для параллельного развития и
- менее пригодным для параллельного выполнения.

Дело в том, что функциональная декомпозиция вынуждает объявлять "важные" данные глобальными, поскольку, если система структурирована как дерево функций, всякое данное, доступное двум функциям, должно быть глобальным по отношению к ним. Это приводит к тому, что "важные" данные "всплывают" к вершине дерева, по мере того как все большее число функций требует доступа к ним.

В точности так же происходит в случае иерархии классов с одним корнем, когда "важные" данные всплывают по направлению к базовому классу.

Когда мы концентрируем внимание на описаниях классов, заключающих определенные данные в оболочку, то зависимости между различными частями программы выражены явно и можно их проследить. Еще более важно то, что при таком подходе уменьшается число зависимостей в системе за счет лучшей расстановки ссылок на данные.

Однако, некоторые задачи лучше решаются с помощью набора процедур. Смысл "объектно-ориентированного" проектирования не в том, чтобы удалить все глобальные процедуры из программы или не иметь в системе процедурно-ориентированных частей. Основная идея скорее в том, что классы, а не глобальные процедуры становятся главным объектом внимания на стадии проектирования. Использование процедурного стиля должно быть осознанным решением, а не решением, принимаемым по умолчанию. Как классы, так и процедуры следует применять сообразно области приложения, а не просто как неизменные методы проектирования.

12.1.2 Игнорирование наследования

Рассмотрим вариант 2 - проект, который игнорирует наследование. В этом случае в окончательной программе просто не используются возможности основного средства C++, хотя и получаются определенные выгоды при использовании C++ по сравнению с использованием языков C, Паскаль, Фортран, Кобол и т.п. Обычные доводы в пользу этого, помимо инерции, утверждения, что "наследование - это деталь реализации", или "наследование препятствует упрятыванию информации", или "наследование затрудняет взаимодействие с другими системами программирования".

Считать наследование всего лишь деталью реализации - значит игнорировать иерархию классов, которая может непосредственно моделировать отношения между понятиями в области приложения. Такие отношения должны быть явно выражены в проекте, чтобы дать

возможность разработчику продумать их.

Сильные доводы можно привести в пользу исключения наследования из тех частей программы на C++, которые непосредственно взаимодействуют с программами, написанными на других языках. Но это не является достаточной причиной, чтобы отказаться от наследования в системе в целом, это просто довод в пользу того, чтобы аккуратно определить и инкапсулировать программный интерфейс с "внешним миром". Аналогично, чтобы избавиться от беспокойства, вызванного путаницей с упрятыванием информации при наличии наследования, надо осторожно использовать виртуальные функции и закрытые члены, но не отказываться от наследования.

Существует достаточно много ситуаций, когда использование наследования не дает явных выгод, но политика "никакого наследования" приведет к менее понятной и менее гибкой системе, в которой наследование "подделывается" с помощью более традиционных конструкций языка и проектирования. Для больших проектов это существенно. Более того, вполне возможно, что несмотря на такую политику, наследование все равно будет использоваться, поскольку программисты, работающие на C++, найдут убедительные доводы в пользу проектирования с учетом наследования в различных частях системы. Таким образом, политика "никакого наследования" приведет лишь к тому, что в системе будет отсутствовать целостная общая структура, а использование иерархии классов будет ограничено определенными подсистемами.

Иными словами, будьте непредубежденными. Иерархия классов не является обязательной частью всякой хорошей программы, но есть масса ситуаций, когда она может помочь как в понимании области приложения, так и в формулировании решений. Утверждение, что наследование может неправильно или чрезмерно использоваться, служит только доводом в пользу осторожности, а вовсе не в пользу отказа от него.

12.1.3 Игнорирование статического контроля типов

Рассмотрим вариант 3, относящийся к проекту, в котором игнорируется

статический контроль типов. Распространенные доводы в пользу отказа на стадии проектирования от статического контроля типов сводятся к тому, что "типы - это продукт языков программирования", или что "более естественно рассуждать об объектах, не заботясь о типах", или "статический контроль типов вынуждает нас думать о реализации на слишком раннем этапе". Такой подход вполне допустим до тех пор, пока он работает и не приносит вреда. Вполне разумно на стадии проектирования не заботиться о деталях проверки типов, и часто вполне допустимо на стадии анализа и начальных стадиях проектирования полностью забыть о вопросах, связанных с типами. В то же время, классы и иерархии классов очень полезны на стадии проектирования, в частности, они дают нам большую определенность понятий, позволяют точно задать взаимоотношения между понятиями и помогают рассуждать о понятиях. По мере развития проекта эта определенность и точность преобразуется во все более конкретные утверждения о классах и их интерфейсах.

Важно понимать, что точно определенные и строго типизированные интерфейсы являются фундаментальным средством проектирования. Язык C++ был создан как раз с учетом этого. Строго типизированный интерфейс гарантирует, что только совместимые части программы могут быть скомпилированы и скомпонованы воедино, и тем самым позволяет делать относительно строгие допущения об этих частях. Эти допущения обеспечиваются системой типов языка.

В результате сводятся к минимуму проверки на этапе выполнения, что повышает эффективность и приводит к значительному сокращению фазы интеграции частей проекта, реализованных разными программистами. Реальный положительный опыт интеграции системы со строго типизированными интерфейсами привел к тому, что вопросы интеграции вообще не фигурируют среди основных тем этой лекции.

Рассмотрим следующую аналогию: в физическом мире мы постоянно соединяем различные устройства, и существует кажущееся бесконечным число стандартов на соединения. Главная особенность этих соединений: они специально спроектированы таким образом, чтобы сделать невозможным соединение двух устройств, нерассчитанных на него, то есть соединение должно быть сделано единственным правильным способом. Вы не можете подсоединить электробритву к

розетке с высоким напряжением. Если бы вы смогли сделать это, то сожгли бы бритву или сгорели сами. Масса изобретательности была проявлена, чтобы добиться невозможности соединения двух несовместимых устройств. Альтернативой одновременного использования нескольких несовместимых устройств может послужить такое устройство, которое само себя защищает от несовместимых с ним устройств, подключающихся к его входу. Хорошим примером может служить стабилизатор напряжения. Поскольку идеальную совместимость устройств нельзя гарантировать только на "уровне соединения", иногда требуется более дорогая защита в электрической цепи, которая позволяет в динамике приспособиться или (и) защититься от скачков напряжения.

Здесь практически прямая аналогия: статический контроль типов эквивалентен совместимости на уровне соединения, а динамические проверки соответствуют защите или адаптации в цепи. Результатом неудачного контроля как в физическом, так и в программном мире будет серьезный ущерб. В больших системах используются оба вида контроля. На раннем этапе проектирования вполне достаточно простого утверждения: "Эти два устройства необходимо соединить"; но скоро становится существенным, как именно следует их соединить: "Какие гарантии дает соединение относительно поведения устройств?", или "Возникновение каких ошибочных ситуаций возможно?", или "Какова приблизительная цена такого соединения?"

Применение "статической типизации" не ограничивается программным миром. В физике и инженерных науках повсеместно распространены единицы измерения (метры, килограммы, секунды), чтобы избежать смешивания несовместимых сущностей.

В нашем описании шагов проектирования в § 11.3.3 типы появляются на сцене уже на шаге 2 (очевидно, после несколько искусственного их рассмотрения на шаге 1) и становятся главной темой шага 4.

Статически контролируемые интерфейсы - это основное средство взаимодействия программных частей системы на C++, созданных разными группами, а описание интерфейсов этих частей (с учетом точных определений типов) становится основным способом сотрудничества между отдельными группами программистов. Эти

интерфейсы являются основным результатом процесса проектирования и служат главным средством общения между разработчиками и программистами.

Отказ от этого приводит к проектам, в которых неясна структура программы, контроль ошибок отложен на стадию выполнения, которые трудно хорошо реализовать на C++.

Рассмотрим интерфейс, описанный с помощью "объектов", определяющих себя самостоятельно. Возможно, например, такое описание: "Функция `f()` имеет аргумент, который должен быть самолетом" (что проверяется самой функцией во время ее выполнения), в отличие от описания "Функция `f()` имеет аргумент, тип которого есть самолет" (что проверяется транслятором). Первое описание является существенно недостаточным описанием интерфейса, т.к. приводит к динамической проверке вместо статического контроля. Аналогичный вывод из примера с самолетом сделан в § 1.5.2. Здесь использованы более точные спецификации, и использован шаблон типа и виртуальные функции взамен неограниченных динамических проверок для того, чтобы перенести выявление ошибок с этапа выполнения на этап трансляции. Различие времен работы программ с динамическим и статическим контролем может быть весьма значительным, обычно оно находится в диапазоне от 3 до 10 раз.

Но не следует впадать в другую крайность. Нельзя обнаружить все ошибки с помощью статического контроля. Например, даже программы с самым обширным статическим контролем уязвимы к сбоям аппаратуры. Но все же, в идеале нужно иметь большое разнообразие интерфейсов со статической типизацией с помощью типов из области приложения, см. § 12.4.

Может получиться, что проект, совершенно разумный на абстрактном уровне, столкнется с серьезными проблемами, если не учитывает ограничения базовых средств, в данном случае C++. Например, использование имен, а не типов для структурирования системы приведет к ненужным проблемам для системы типов C++ и, тем самым, может стать причиной ошибок и накладных расходов при выполнении. Рассмотрим три класса:

```
class X { // pseudo code, not C++
  f()
  g()
}

class Y {
  g()
  h()
}

class Z {
  h()
  f()
}
```

используемые некоторыми функциями бестипового проекта:

```
k(a, b, c)  // pseudo code, not C++
{
  a.f()
  b.g()
  c.h()
}
```

Здесь обращения

```
X x
Y y
Z z

k(x,y,z)  // ok
k(z,x,y)  // ok
```

будут успешными, поскольку `k()` просто требует, чтобы ее первый параметр имел операцию `f()`, второй параметр - операцию `g()`, а третий параметр - операцию `h()`. С другой стороны обращения

```
k(y,x,z);  // fail
k(x,z,y);  // fail
```

завершатся неудачно. Этот пример допускает совершенно разумные реализации на языках с полным динамическим контролем (например, Smalltalk или CLOS), но в C++ он не имеет прямого представления, поскольку язык требует, чтобы общность типов была реализована как отношение к базовому классу. Обычно примеры, подобные этому, можно представить на C++, если записывать утверждения об общности с помощью явных определений классов, но это потребует большого хитроумия и вспомогательных средств. Можно сделать, например, так:

```
class F {
  virtual void f();
};

class G {
  virtual void g();
};

class H {
  virtual void h();
};

class X : public virtual F, public virtual G {
  void f();
  void g();
};

class Y : public virtual G, public virtual H {
  void g();
  void h();
};

class Z : public virtual H, public virtual F {
  void h();
  void f();
};

k(const F& a, const G& b, const H& c)
{
  a.f();
```

```
  b.g();
  c.h();
 }

 main()
 {
  X x;
  Y y;
  Z z;

  k(x,y,z);  // ok
  k(z,x,y);  // ok

  k(y,x,z);  // error F required for first argument
  k(x,z,y);  // error G required for second argument
 }
```

Обратите внимание, что сделав предположения `k()` о своих аргументах явными, мы переместили контроль ошибок с этапа выполнения на этап трансляции. Сложные примеры, подобные приведенному, возникают, когда пытаются реализовать на C++ проекты, сделанные на основе опыта работы с другими системами типов. Обычно это возможно, но в результате получается неестественная и неэффективная программа. Такое несовпадение между приемами проектирования и языком программирования можно сравнить с несовпадением при пословном переводе с одного естественного языка на другой. Ведь английский с немецкой грамматикой выглядит столь же неуклюже, как и немецкий с английской грамматикой, но оба языка могут быть доступны пониманию того, кто бегло говорит на одном из них.

Этот пример подтверждает тот вывод, что классы в программе являются конкретным воплощением понятий, используемых при проектировании, поэтому нечеткие отношения между классами приводят к нечеткости основных понятий проектирования.

12.1.4 Гибридный проект

Переход на новые методы работы может быть мучителен для любой организации. Раскол внутри нее и расхождения между сотрудниками могут быть значительными. Но резкий решительный переход, способный в одночасье превратить эффективных и квалифицированных сторонников "старой школы" в неэффективных новичков "новой школы" обычно неприемлем. В то же время, нельзя достичь больших высот без изменений, а значительные изменения обычно связаны с риском.

Язык C++ создавался с целью сократить такой риск за счет постепенного введения новых методов. Хотя очевидно, что наибольшие преимущества при использовании C++ достигаются за счет абстракции данных, объектно-ориентированного программирования и объектно-ориентированного проектирования, совершенно неочевидно, что быстрее всего достичь этого можно решительным разрывом с прошлым. Вряд ли такой явный разрыв будет возможен, обычно стремление к усовершенствованиям сдерживается или должно сдерживаться, чтобы переход к ним был управляемым. Нужно учитывать следующее:

- Разработчикам и программистам требуется время для овладения новыми методами.
- Новые программы должны взаимодействовать со старыми программами.
- Старые программы нужно сопровождать (часто бесконечно).
- Работа по текущим проектам и программам должна быть выполнена в срок.
- Средства, рассчитанные на новые методы, нужно адаптировать к локальному окружению.

Здесь рассматриваются как раз ситуации, связанные с перечисленными требованиями. Легко недооценить два первых требования.

Поскольку в C++ возможны несколько схем программирования, язык допускает постепенный переход на него, используя следующие преимущества такого перехода:

- Изучая C++, программисты могут продолжать работать.
- В окружении, бедном на программные средства, использование C ++ может принести значительные выгоды.
- Программы, написанные на C++, могут хорошо

взаимодействовать с программами, написанными на C или других традиционных языках.
- Язык имеет большое подмножество, совместимое с C.

Идея заключается в постепенном переходе программиста с традиционного языка на C++: вначале он программирует на C++ в традиционном процедурном стиле, затем с помощью методов абстракции данных, и наконец, когда овладеет языком и связанными с ним средствами, полностью переходит на объектно-ориентированное программирование. Заметим, что хорошо спроектированную библиотеку использовать намного проще, чем проектировать и реализовывать, поэтому даже с первых своих шагов новичок может получить преимущества, используя более развитые средства C++.

Идея постепенного, пошагового овладения C++, а также возможность смешивать программы на C++ с программами, написанными на языках, не имеющих средств абстракции данных и объектно-ориентированного программирования, естественно приводит к проекту, имеющему гибридный стиль. Большинство интерфейсов можно пока оставить на процедурном уровне, поскольку что-либо более сложное не принесет немедленного выигрыша. Например, обращение к стандартной библиотеке `math` из C определяется на C++ так:

```
extern "C" {
  #include <math.h>
}
```

и стандартные математические функции из библиотеки можно использовать так же, как и в C. Для всех основных библиотек такое включение должно быть сделано теми, кто поставляет библиотеки, так что программист на C++ даже не будет знать, на каком языке реализована библиотечная функция. Использование библиотек, написанных на таких языках как C, является первым и вначале самым важным способом повторного использования на C++.

На следующем шаге, когда станут необходимы более сложные приемы, средства, реализованные на таких языках как C или Фортран, представляются в виде классов за счет инкапсуляции структур данных и функций в интерфейс классов C++. Простым примером введения более

высокого семантического уровня за счет перехода от уровня процедур плюс структур данных к уровню абстракции данных может служить класс строк из § 7.6. Здесь за счет инкапсуляции символьных строк и стандартных строковых функций C получается новый строковый тип, который гораздо проще использовать.

Подобным образом можно включить в иерархию классов любой встроенный или отдельно определенный тип. Например, тип `int` можно включить в иерархию классов так:

```
class Int : public My_object {
  int i;
public:
  // definition of operations
  // see exercises [8]-[11] in section 7.14 for ideas
  // определения операций получаются в упражнениях [8]-[11]
  // за идеями обратитесь к разделу 7.14
};
```

Так следует делать, если действительно есть потребность включить такие типы в иерархию.

Обратно, классы C++ можно представить в программе на C или Фортране как функции и структуры данных. Например:

```
class myclass {
   // representation
public:
  void f();
  T1 g(T2);
  // ...
};

extern "C" {  // map myclass into C callable functions:

  void myclass_f(myclass* p) { p->f(); }
  T1 myclass_g(myclass* p, T2 a) { return p->g(a); }
  // ...
};
```

В C-программе следует определить эти функции в заголовочном файле следующим образом:

```
// in C header file

extern void myclass_f(struct myclass*);
extern T1 myclass_g(struct myclass*, T2);
```

Такой подход позволяет разработчику на C++, если у него уже есть запас программ, написанных на языках, в которых отсутствуют понятия абстракции данных и иерархии классов, постепенно приобщаться к этим понятиям, даже при том требовании, что окончательную версии программы можно будет вызывать из традиционных процедурных языков.

12.2 Классы

Основное положение объектно-ориентированного проектирования и программирования заключается в том, что программа служит моделью некоторых понятий реальности. Классы в программе представляют основные понятия области приложения и, в частности, основные понятия самого процесса моделирования реальности. Объекты классов представляют предметы реального мира и продукты процесса реализации.

Мы рассмотрим структуру программы с точки зрения следующих взаимоотношений между классами:

- отношения наследования,
- отношения принадлежности,
- отношения использования и
- запрограммированные отношения.

При рассмотрении этих отношений неявно предполагается, что их анализ является узловым моментом в проекте системы. В § 12.4 исследуются свойства, которые делают класс и его интерфейс полезными для представления понятий. Вообще говоря, в идеале, зависимость класса от остального мира должна быть минимальна и

четко определена, а сам класс должен через интерфейс открывать лишь минимальный объем информации для остального мира.

Подчеркнем, что класс в C++ является типом, поэтому сами классы и взаимоотношения между ними обеспечены значительной поддержкой со стороны транслятора и в общем случае поддаются статическому анализу.

12.2.1 Что представляют классы?

По сути в системе бывают классы двух видов:

1. классы, которые прямо отражают понятия области приложения, т.е. понятия, которые использует конечный пользователь для описания своих задач и возможных решений; и
2. классы, которые являются продуктом самой реализации, т.е. отражают понятия, используемые разработчиками и программистами для описания способов реализации.

Некоторые из классов, являющихся продуктами реализации, могут представлять и понятия реального мира. Например, программные и аппаратные ресурсы системы являются хорошими кандидатами на роль классов, представляющих область приложения. Это отражает тот факт, что систему можно рассматривать с нескольких точек зрения, и то, что с одной является деталью реализации, с другой может быть понятием области приложения. Хорошо спроектированная система должна содержать классы, которые дают возможность рассматривать систему с логически разных точек зрения. Приведем пример:

1. классы, представляющие пользовательские понятия (например, легковые машины и грузовики),
2. классы, представляющие обобщения пользовательских понятий (движущиеся средства),
3. классы, представляющие аппаратные ресурсы (например, класс управления памятью),
4. классы, представляющие системные ресурсы (например, выходные потоки),

5. классы, используемые для реализации других классов (например, списки, очереди, блокировщики) и
6. встроенные типы данных и структуры управления.

В больших системах очень трудно сохранять логическое разделение типов различных классов и поддерживать такое разделение между различными уровнями абстракции. В приведенном выше перечислении представлены три уровня абстракции:

- [1+2] представляет пользовательское отражение системы,
- [3+4] представляет машину, на которой будет работать система,
- [5+6] представляет низкоуровневое (со стороны языка программирования) отражение реализации.

Чем больше система, тем большее число уровней абстракции необходимо для ее описания, и тем труднее определять и поддерживать эти уровни абстракции. Отметим, что таким уровням абстракции есть прямое соответствие в природе и в различных построениях человеческого интеллекта. Например, можно рассматривать дом как объект, состоящий из

1. атомов,
2. молекул,
3. досок и кирпичей,
4. полов, потолков и стен;
5. комнат.

Пока удается хранить раздельно представления этих уровней абстракции, можно поддерживать целостное представление о доме. Однако, если смешать их, возникнет бессмыслица. Например, предложение "Мой дом состоит из нескольких тысяч фунтов углерода, некоторых сложных полимеров, из 5000 кирпичей, двух ванных комнат и 13 потолков" - явно абсурдно. Из-за абстрактной природы программ подобное утверждение о какой-либо сложной программной системе далеко не всегда воспринимают как бессмыслицу.

В процессе проектирования выделение понятий из области приложения в класс вовсе не является простой механической

операцией. Обычно эта задача требует большой проницательности. Заметим, что сами понятия области приложения являются абстракциями. Например, в природе не существуют "налогоплательщики", "монахи" или "сотрудники".

Эти понятия не что иное, как метки, которыми обозначают бедную личность, чтобы классифицировать ее по отношению к некоторой системе. Часто реальный или воображаемый мир (например, литература, особенно фантастика) служат источником понятий, которые кардинально преобразуются при переводе их в классы. Так, экран моего компьютера (Макинтош) совсем не походит на поверхность моего стола, хотя компьютер создавался с целью реализовать понятие "настольный", а окна на моем дисплее имеют самое отдаленное отношение к приспособлениям для презентации чертежей в моей комнате. Я бы не вынес такого беспорядка у себя на экране.

Суть моделирования реальности не в покорном следовании тому, что мы видим, а в использовании реальности как начала для проектирования, источника вдохновения и как якоря, который удерживает, когда стихия программирования грозит лишить нас способности понимания своей собственной программы.

Здесь полезно предостеречь: новичкам обычно трудно "находить" классы, но вскоре это преодолевается без каких-либо неприятностей. Далее обычно приходит этап, когда классы и отношения наследования между ними бесконтрольно множатся. Здесь уже возникают проблемы, связанные со сложностью, эффективностью и ясностью полученной программы. Далеко не каждую отдельную деталь следует представлять отдельным классом, и далеко не каждое отношение между классами следует представлять как отношение наследования. Старайтесь не забывать, что цель проекта - смоделировать систему с подходящим уровнем детализации и подходящим уровнем абстракции. Для больших систем найти компромисс между простотой и общностью далеко не простая задача.

12.2.2 Иерархии классов

Рассмотрим моделирование транспортного потока в городе, цель

которого достаточно точно определить время, требующееся, чтобы аварийные движущиеся средства достигли пункта назначения. Очевидно, нам надо иметь представления легковых и грузовых машин, машин скорой помощи, всевозможных пожарных и полицейских машин, автобусов и т.п. Поскольку всякое понятие реального мира не существует изолированно, а соединено многочисленными связями с другими понятиями, возникает такое отношение как наследование. Не разобравшись в понятиях и их взаимных связях, мы не в состоянии постичь никакое отдельное понятие. Также и модель, если не отражает отношения между понятиями, не может адекватно представлять сами понятия. Итак, в нашей программе нужны классы для представления понятий, но этого недостаточно. Нам нужны способы представления отношений между классами. Наследование является мощным способом прямого представления иерархических отношений. В нашем примере, мы, по всей видимости, сочли бы аварийные средства специальными движущимися средствами и, помимо этого, выделили бы средства, представленные легковыми и грузовыми машинами. Тогда иерархия классов приобрела бы такой вид:

- движущееся средство
- легковая машина
- аварийное средство
- грузовая машина
- полицейская машина
- скорой помощи
- пожарная машина
- машина с выдвижной лестницей.

Здесь класс `Emergency` представляет всю информацию, необходимую для моделирования аварийных движущихся средств, например: аварийная машина может нарушать некоторые правила движения, она имеет приоритет на перекрестках, находится под контролем диспетчера и т.д.

На C++ это можно задать так:

```
class Vehicle { /*...*/ };
class Emergency { /*   */ };
class Car : public Vehicle { /*...*/ };
```

```
class Truck : public Vehicle { /*...*/ };
class Police_car : public Car , public Emergency {
  //...
};
class Ambulance : public Car , public Emergency {
  //...
};
class Fire_engine : public Truck , Emergency {
  //...
};
class Hook_and_ladder : public Fire_engine {
  //...
};
```

Наследование - это отношение самого высокого порядка, которое прямо представляется в C++ и используется преимущественно на ранних этапах проектирования. Часто возникает проблема выбора: использовать наследование для представления отношения или предпочесть ему принадлежность. Рассмотрим другое определение понятия аварийного средства: движущееся средство считается аварийным, если оно несет соответствующий световой сигнал. Это позволит упростить иерархию классов, заменив класс Emergency на член класса Vehicle:

- движущееся средство (Vehicle {eptr})
- легковая машина (Car) грузовая машина (Truck)
- полицейская машина (Police_car) машина скорой помощи (Ambulance)
- пожарная машина (Fire_engine)
- машина с выдвижной лестницей (Hook_and_ladder)

Теперь класс Emergency используется просто как член в тех классах, которые представляют аварийные движущиеся средства:

```
class Emergency { /*...*/ };
class Vehicle { public: Emergency* eptr;  /*...*/ };
class Car : public Vehicle { /*...*/ };
class Truck : public Vehicle { /*...*/ };
class Police_car : public Car { /*...*/ };
```

```
class Ambulance : public Car { /*...*/ };
class Fire_engine : public Truck { /*...*/ };
class Hook_and_ladder : public Fire_engine { /*...*/ };
```

Здесь движущееся средство считается аварийным, если `Vehicle::eptr` не равно нулю. "Простые" легковые и грузовые машины инициализируются `Vehicle::eptr` равным нулю, а для других `Vehicle::eptr` должно быть установлено в ненулевое значение, например:

```
Car::Car()    // конструктор Car
{
  eptr = 0;
}
Police_car::Police_car()   // конструктор Police_car
{
  eptr = new Emergency;
}
```

Такие определения упрощают преобразование аварийного средства в обычное и наоборот:

```
void f(Vehicle* p)
{
  delete p->eptr;
  p->eptr = 0;   // больше нет аварийного движущегося средства

  //...

  p->eptr = new Emergency;   // оно появилось снова
}
```

Так какой же вариант иерархии классов лучше? В общем случае ответ такой: "Лучшей является программа, которая наиболее непосредственно отражает реальный мир". Иными словами, при выборе модели мы должны стремиться к большей ее"реальности", но с учетом неизбежных ограничений, накладываемых требованиями простоты и эффективности. Поэтому, несмотря на простоту преобразования обычного движущегося средства в аварийное, второе решение

представляется непрактичным.

Пожарные машины и машины скорой помощи - это движущиеся средства специального назначения со специально подготовленным персоналом, они действуют под управлением команд диспетчера, требующих специального оборудования для связи. Такое положение означает, что принадлежность к аварийным движущимся средствам - это базовое понятие, которое для улучшения контроля типов и применения различных программных средств должно быть прямо представлено в программе. Если бы мы моделировали ситуацию, в которой назначение движущихся средств не столь определенно, скажем, ситуацию, в которой частный транспорт периодически используется для доставки специального персонала к месту происшествия, а связь обеспечивается с помощью портативных приемников, тогда мог бы оказаться подходящим и другой способ моделирования системы.

Для тех, кто считает пример моделирования движения транспорта экзотичным, имеет смысл сказать, что в процессе проектирования почти постоянно возникает подобный выбор между наследованием и принадлежностью. Аналогичный пример есть в § 12.2.5, где описывается свиток (scrollbar) - прокручивание информации в окне.

12.2.3 Зависимости в рамках иерархии классов.

Естественно, производный класс зависит от своих базовых классов. Гораздо реже учитывают, что обратное также может быть справедливо.

Эту мысль можно выразить таким способом: "Сумасшествие наследуется, вы можете получить его от своих детей."

Если класс содержит виртуальную функцию, производные классы могут по своему усмотрению решать, реализовывать ли часть операций этой функции каждый раз, когда она переопределяется в производном классе. Если член базового класса сам вызывает одну из виртуальных функций производного класса, тогда реализация базового класса зависит от реализаций его производных классов. Точно так же, если класс использует защищенный член, его реализация будет зависеть от производных классов. Рассмотрим определения:

```
class B {
    //...
protected:
    int a;
public:
    virtual int f();
    int g() { int x = f(); return x-a; }
};
```

Каков результат работы `g()`? Ответ существенно зависит от определения `f()` в некотором производном классе. Ниже приводится вариант, при котором `g()` будет возвращать 1:

```
class D1 : public B {
    int f() { return a+1; }
};
```

а при нижеследующем определении g() напечатает "Hello, World" и вернет 0:

```
class D1 : public B {
    int f() { cout<<"Hello, World\n"; return a; }
};
```

Этот пример демонстрирует один из важнейших моментов, связанных с виртуальными функциями. Хотя вы можете сказать, что это глупость, и программист никогда не напишет ничего подобного. Дело здесь в том, что виртуальная функция является частью интерфейса с базовым классом, и что этот класс будет, по всей видимости, использоваться без информации о его производных классах. Следовательно, можно так описать поведение объекта базового класса, чтобы в дальнейшем писать программы, ничего не зная о его производных классах.

Всякий класс, который переопределяет производную функцию, должен реализовать вариант этой функции. Например, виртуальная функция `rotate()` из класса `Shape` вращает геометрическую фигуру, а функции `rotate()` для производных классов, таких, как `Circle` и `Triangle`, должны вращать объекты соответствующих типов, иначе будет нарушено основное положение о классе `Shape`. Но о поведении

класса B или его производных классов `D1` и `D2` не сформулировано никаких положений, поэтому приведенный пример и кажется неразумным. При построении класса главное внимание следует уделять описанию ожидаемых действий виртуальных функций.

Следует ли считать нормальной зависимость от неизвестных (возможно еще неопределенных) производных классов? Ответ, естественно, зависит от целей программиста. Если цель состоит в том, чтобы изолировать класс от всяких внешних влияний и, тем самым, доказать, что он ведет себя определенным образом, то лучше избегать виртуальных функций и защищенных членов. Если цель состоит в том, чтобы разработать структуру, в которую последующие программисты (или вы сами через неделю) смогут встраивать свои программы, то именно виртуальные функции и предлагают элегантный способ решения, а защищенные члены могут быть полезны при его реализации.

В качестве примера рассмотрим простой шаблон типа, определяющий буфер:

```
template<class T> class buffer {
  // ...
  void put(T);
  T get();
};
```

Если реакция на переполнение и обращение к пустому буферу, "запаяна" в сам класс, его применение будет ограничено. Но если функции `put()` и `get()` обращаются к виртуальным функциям `overflow()` и `underflow()` соответственно, то пользователь может, удовлетворяя своим нуждам, создать буфера различных типов:

```
template<class T> class buffer {
  //...
  virtual int overflow(T);
  virtual int underflow();
  void put(T);   // вызвать overflow(T), когда буфер полон
  T get();   // вызвать underflow(T), когда буфер пуст
};
```

```
template<class T> class circular_buffer : public buffer<T> {
    //...
    int overflow(T);  // перейти на начало буфера, если он полон
    int underflow();
};

template<class T> class expanding_buffer : public buffer<T> {
    //...
    int overflow(T);   // увеличить размер буфера, если он полон
    int underflow();
};
```

Этот метод использовался в библиотеках потокового ввода-вывода (§ 10.5.3).

12.2.4 Отношения принадлежности

Если используется отношение принадлежности, то существует два основных способа представления объекта класса `X`:

1. Описать член типа `X`.
2. Описать член типа `X*` или `X&`.

Если значение указателя не будет меняться и вопросы эффективности не волнуют, эти способы эквивалентны:

```
class X {
    //...
public:
    X(int);
    //...
};

class C {
     X a;
     X* p;
public:
```

```
    C(int i, int j) : a(i), p(new X(j)) { }
    ~C()  { delete p; }
};
```

В таких ситуациях предпочтительнее непосредственное членство объекта, как `X::a` в примере выше, потому что оно дает экономию времени, памяти и количества вводимых символов. Обратитесь также к §12.4 и §13.9.

Способ, использующий указатель, следует применять в тех случаях, когда приходится перестраивать указатель на "объект-элемент" в течение жизни "объекта-владельца". Например:

```
class C2 {
    X* p;
public:
    C(int i) : p(new X(i))  { }
    ~C() { delete p; }

    X* change(X* q)
    {
        X* t = p;
        p = q;
        return t;
    }
};
```

Член типа указатель может также использоваться, чтобы дать возможность передавать "объект-элемент" в качестве параметра:

```
class C3 {
  X* p;
public:
    C(X* q) : p(q) {  }
    // ...
}
```

Разрешая объектам содержать указатели на другие объекты, мы создаем то, что обычно называется "иерархия объектов". Это альтернативный и

вспомогательный способ структурирования по отношению к иерархии классов. Как было показано на примере аварийного движущегося средства в § 12.2.2, часто это довольно тонкий вопрос проектирования: представлять ли свойство класса как еще один базовый класс или как член класса. Потребность в переопределении следует считать указанием, что первый вариант лучше. Но если надо иметь возможность представлять некоторое свойство с помощью различных типов, то лучше остановиться на втором варианте. Например:

```
class XX : public X { /*...*/ };

class XXX : public X { /*...*/ };

void f()
{
  C3* p1 = new C3(new X);     // C3 "содержит" X
  C3* p2 = new C3(new XX);    // C3 "содержит" XX
  C3* p3 = new C3(new XXX);   // C3 "содержит" XXX
  //...
}
```

Приведенные определения нельзя смоделировать ни с помощью производного класса `C3` от `X`, ни с помощью `C3`, имеющего член типа `X`, поскольку необходимо указывать точный тип члена. Это важно для классов с виртуальными функциями, таких, например,как класс `Shape` (§ 1.1.2.5), и для класса абстрактного множества (§ 13.3).

Заметим, что ссылки можно применять для упрощения классов, использующих члены-указатели, если в течение жизни объекта-владельца ссылка настроена только на один объект, например:

```
class C4 {
   X&  r;
public:
   C(X& q) : r(q) { }
   // ...
 };
```

12.2.5 Принадлежность и наследование

Учитывая сложность и важность отношений наследования, нет ничего удивительного в том, что часто их неправильно понимают и используют сверх меры. Если класс `D` описан как общий производный от класса `B`, то часто говорят, что `D` есть `B`:

```
class B { /* ... */ };
class D : public B  {/* ... */ }; // D сорта B
```

Иначе это можно сформулировать так: наследование - это отношение "есть", или, более точно для классов `D` и `B`, наследование - это отношение `D` сорта `B`. В отличие от этого, если класс `D` содержит в качестве члена другой класс `B`, то говорят, что `D` "имеет" `B`:

```
class D {  // D имеет B
           // ...
         public:
           B b;
           // ...
         };
```

Иными словами, принадлежность - это отношение "иметь" или для классов `D` и `B` просто: `D` содержит `B`.

Имея два класса `B` и `D`, как выбирать между наследованием и принадлежностью? Рассмотрим классы самолет и мотор.Новички обычно спрашивают: будет ли хорошим решением сделать класс самолет производным от класса мотор. Это плохое решение, поскольку самолет не "есть" мотор, самолет "имеет" мотор. Следует подойти к этому вопросу, рассмотрев, может ли самолет "иметь" два или больше моторов. Поскольку это представляется вполне возможным (даже если мы имеем дело с программой, в которой все самолеты будут с одним мотором), следует использовать принадлежность, а не наследование. Вопрос "Может ли он иметь два..?" оказывается удивительно полезным во многих сомнительных случаях. Как всегда, наше изложение затрагивает неуловимую сущность программирования. Если бы все классы было так же легко представить, как самолет и мотор, то было бы

просто избежать и тривиальных ошибок типа той, когда самолет определяется как производное от класса мотор. Однако, такие ошибки достаточно часты, особенно у тех, кто считает наследование еще одним механизмом для сочетания конструкций языка программирования. Несмотря на удобство и лаконичность записи, которую предоставляет наследование, его надо использовать только для выражения тех отношений, которые четко определены в проекте. Рассмотрим определения:

```
class B {
public:
    virtual void f();
    void g();
};

class D1 {      // D1 содержит B
public:
    B b;
    void f();   // не переопределяет b.f()
};

void h1(D1* pd)
{
   B* pb = pd;  // ошибка: невозможно преобразование D1* в B*
   pb = &pd->b;
   pb->g();     // вызов B::g
   pd->g();     // ошибка: D1 не имеет член g()
   pd->b.g();
   pb->f();     // вызов B::f (здесь D1::f не переопределяет)
   pd->f();     // вызов D1::f
}
```

Обратите внимание, что в этом примере нет неявного преобразования класса к одному из его элементов, и что класс, содержащий в качестве члена другой класс, не переопределяет виртуальные функции этого члена. Здесь явное отличие от примера, приведенного ниже:

```
class D2 : public B {    // D2 есть B
public:
```

```
    void f();          // переопределение B::f()
};

void h2(D2* pd)
{
  B* pb = pd;  // нормально: D2* неявно преобразуется в B*
  pb->g();     // вызов B::g
  pd->g();     // вызов B::g
  pb->f();     // вызов виртуальной функции: обращение к D2::f
  pd->f();     // вызов D2::f
}
```

Удобство записи, продемонстрированное в примере с классом `D2`, по сравнению с записью в примере с классом `D1`, является причиной, по которой таким наследованием злоупотребляют. Но следует помнить, что существует определенная плата за удобство записи в виде возросшей зависимости между `B` и `D2` (см. § 12.2.3). В частности, легко забыть о неявном преобразовании `D2` в `B`. Если только такие преобразования не относятся к семантике ваших классов, следует избегать описания производного класса в общей части. Если класс представляет определенное понятие, а наследование используется как отношение "есть", то такие преобразования обычно как раз то, что нужно.

Однако, бывают такие ситуации, когда желательно иметь наследование, но нельзя допускать преобразования. Рассмотрим задание класса `cfield` (controled field - управляемое поле), который, помимо всего прочего, дает возможность контролировать на стадии выполнения доступ к другому классу `field`. На первый взгляд кажется совершенно правильным определить класс `cfield` как производный от класса `field`:

```
class cfield : public field {
  // ...
};
```

Это выражает тот факт, что `cfield`, действительно, есть сорта `field`, упрощает запись функции, которая использует член части `field` класса `cfield`, и, что самое главное, позволяет в классе cfield переопределять виртуальные функции из `field`. Загвоздка здесь в том, что

преобразование `cfield*` к `field*`, встречающееся в определении класса `cfield`, позволяет обойти любой контроль доступа к `field`:

```
void g(cfield* p)
{
    *p = "asdf";   // обращение к field контролируется
   // функцией присваивания cfield:
        // p->cfield::operator=("asdf")

    field* q = p;  // неявное преобразование cfield* в field*
    *q = "asdf";   // приехали! контроль обойден
}
```

Можно было бы определить класс `cfield` так, чтобы `field` был его членом, но тогда `cfield` не может переопределять виртуальные функции `field`. Лучшим решением здесь будет использование наследования со спецификацией `private` (частное наследование):

```
class cfield : private field { /* ... */ }
```

С позиции проектирования, если не учитывать (иногда важные) вопросы переопределения, частное наследование эквивалентно принадлежности. В этом случае применяется метод, при котором класс определяется в общей части как производный от абстрактного базового класса заданием его интерфейса, а также определяется с помощью частного наследования от конкретного класса, задающего реализацию (§ 13.3). Поскольку наследование, используемое как частное, является спецификой реализации, и оно не отражается в типе производного класса, то его иногда называют "наследованием по реализации", и оно является контрастом для наследования в общей части, когда наследуется интерфейс базового класса и допустимы неявные преобразования к базовому типу. Последнее наследование иногда называют определением подтипа или "интерфейсным наследованием".

Для дальнейшего обсуждения возможности выбора наследования или принадлежности рассмотрим, как представить в диалоговой графической системе свиток (область для прокручивания в ней информации), и как привязать свиток к окну на экране. Потребуются свитки двух видов: горизонтальные и вертикальные. Это можно

представить с помощью двух типов `horizontal_scrollbar` и `vertical_scrollbar` или с помощью одного типа scrollbar, который имеет аргумент, определяющий, является расположение вертикальным или горизонтальным. Первое решение предполагает, что есть еще третий тип, задающий просто свиток - `scrollbar`, и этот тип является базовым классом для двух определенных свитков. Второе решение предполагает дополнительный аргумент у типа scrollbar и наличие значений, задающих вид свитка. Например, так:

```
enum orientation { horizontal, vertical };
```

Как только мы остановимся на одном из решений, определится объем изменений, которые придется внести в систему. Допустим, в этом примере нам потребуется ввести свитки третьего вида. Вначале предполагалось, что могут быть свитки только двух видов (ведь всякое окно имеет только два измерения), но в этом примере, как и во многих других, возможны расширения, которые возникают как вопросы перепроектирования. Например, может появиться желание использовать "управляющую кнопку" (типа мыши) вместо свитков двух видов. Такая кнопка задавала бы прокрутку в различных направлениях в зависимости от того, в какой части окна нажал ее пользователь. Нажатие в середине верхней строчки должно вызывать "прокручивание вверх", нажатие в середине левого столбца - "прокручивание влево", нажатие в левом верхнем углу - "прокручивание вверх и влево". Такая кнопка не является чем-то необычным, и ее можно рассматривать как уточнение понятия свитка, которое особенно подходит для тех областей приложения, которые связаны не с обычными текстами, а с более сложной информацией.

Для добавления управляющей кнопки к программе, использующей иерархию из трех свитков, требуется добавить еще один класс, но не нужно менять программу, работающую со старыми свитками:

- свиток
- горизонтальный_свиток
- вертикальный_свиток
- управляющая_кнопка

Это положительная сторона "иерархического решения".

Задание ориентации свитка в качестве параметра приводит к заданию полей типа в объектах свитка и использованию переключателей в теле функций-членов свитка. Иными словами, перед нами обычная дилемма: выразить данный аспект структуры системы с помощью определений или реализовать его в операторной части программы. Первое решение увеличивает объем статических проверок и объем информации, над которой могут работать разные вспомогательные средства. Второе решение откладывает проверки на стадию выполнения и разрешает менять тела отдельных функций, не изменяя общую структуру системы, какой она представляется с точки зрения статического контроля или вспомогательных средств. В большинстве случаев предпочтительнее первое решение.

Положительной стороной решения с единым типом свитка является то, что легко передавать информацию о виде нужного нам свитка другой функции:

```
void helper(orientation oo)
{
  //...
  p = new scrollbar(oo);
  //...
}

void me()
{
   helper(horizontal);
}
```

Такой подход позволяет на стадии выполнения легко перенастроить свиток на другую ориентацию. Вряд ли это очень важно в примере со свитками, но это может оказаться существенным в похожих примерах. Суть в том, что всегда надо делать определенный выбор, а это часто непросто.

Теперь рассмотрим как привязать свиток к окну. Если считать `window_with_scrollbar` (окно_со_свитком) как нечто, что является `window` и `scrollbar`, мы получим подобное:

```
class window_with_scrollbar
: public window, public scrollbar {
   // ...
};
```

Это позволяет любому объекту типа `window_with_scrollbar` выступать и как `window`, и как `scrollbar`, но от нас требуется решение использовать только единственный тип `scrollbar`.

Если, с другой стороны, считать `window_with_scrollbar` объектом типа `window`, который имеет `scrollbar`, мы получим такое определение:

```
class window_with_scrollbar : public window {
 // ...
 scrollbar* sb;
public:
   window_with_scrollbar(scrollbar* p, /* ... */)
   : window(/* ... */), sb(p)
   {
     // ...
   }
 // ...
};
```

Здесь мы можем использовать решение со свитками трех типов. Передача самого свитка в качестве параметра позволяет окну (window) не запоминать тип его свитка. Если потребуется, чтобы объект типа `window_with_scrollbar` действовал как scrollbar, можно добавить операцию преобразования:

```
window_with_scrollbar :: operator scrollbar&()
{
  return *sb;
}
```

12.2.6 Отношения использования

Для составления и понимания проекта часто необходимо знать, какие классы и каким способом использует данный класс. Такие отношения классов на C++ выражаются неявно. Класс может использовать только те имена, которые где-то определены, но нет такой части в программе на C++, которая содержала бы список всех используемых имен. Для получения такого списка необходимы вспомогательные средства (или, при их отсутствии, внимательное чтение). Можно следующим образом классифицировать те способы, с помощью которых класс `X` может использовать класс `Y`:

- `X` использует имя `Y`
- `X` использует `Y`
  - `X` вызывает функцию-член `Y`
  - `X` читает член `Y`
  - `X` пишет в член `Y`
- `X` создает `Y`
  - `X` размещает `auto` или `static` переменную из `Y`
  - `X` создает `Y` с помощью `new`
  - `X` использует размер `Y`

Мы отнесли использование размера объекта к его созданию, поскольку для этого требуется знание полного определения класса. С другой стороны, мы выделили в отдельное отношение использование имени `Y`, поскольку, указывая его в описании `Y*` или в описании внешней функции, мы вовсе не нуждаемся в доступе к определению `Y`:

```
class Y;  // Y - имя класса
Y* p;
extern Y f(const Y&);
```

Мы отделили создание `Y` с помощью `new` от случая описания переменной, поскольку возможна такая реализация C++, при которой для создания `Y` с помощью `new` необязательно знать размер `Y`. Это может быть существенно для ограничения всех зависимостей в проекте и сведения к минимуму перетрансляции после внесения изменений.

Язык C++ не требует, чтобы создатель классов точно определял, какие классы и как он будет использовать. Одна из причин этого заключена в

том, что самые важные классы зависят от столь большого количества других классов, что для придания лучшего вида программе нужна сокращенная форма записи списка используемых классов, например, с помощью команды `#include`. Другая причина в том, что классификация этих зависимостей и, в частности, обЪединение некоторых зависимостей не является обязанностью языка программирования. Наоборот, цели разработчика, программиста или вспомогательного средства определяют то, как именно следует рассматривать отношения использования. Наконец, то, какие зависимости представляют больший интерес, может зависеть от специфики реализации языка.

12.2.7 Отношения внутри класса

До сих пор мы обсуждали только классы, и хотя операции упоминались, если не считать обсуждения шагов процесса развития программного обеспечения (§ 11.3.3.2), то они были на втором плане, объекты же практически вообще не упоминались. Понять это просто: в C++ класс, а не функция или объект, является основным понятием организации системы.

Класс может скрывать в себе всякую специфику реализации, наравне с "грязными" приемами программирования, а иногда он вынужден это делать. В то же время объекты большинства классов сами образуют регулярную структуру и используются такими способами, что их достаточно просто описать. Объект класса может быть совокупностью других вложенных объектов (их часто называют членами), многие из которых, в свою очередь, являются указателями или ссылками на другие объекты. Поэтому отдельный объект можно рассматривать как корень дерева объектов, а все входящие в него объекты как "иерархию объектов", которая дополняет иерархию классов, рассмотренную в § 12.2.4. Рассмотрим в качестве примера класс строк из § 7.6:

```
class String {
    int sz;
    char* p;
public:
```

```
    String(const char* q);
    ~String();
    //...
};
```

12.2.7.1 Инварианты

Значение членов или объектов, доступных с помощью членов класса, называется состоянием объекта (или просто значением объекта). Главное при построении класса - это: привести объект в полностью определенное состояние (инициализация), сохранять полностью определенное состояние обЪекта в процессе выполнения над ним различных операций, и в конце работы уничтожить объект без всяких последствий. Свойство, которое делает состояние объекта полностью определенным, называется инвариантом.

Поэтому назначение инициализации - задать конкретные значения, при которых выполняется инвариант объекта. Для каждой операции класса предполагается, что инвариант должен иметь место перед выполнением операции и должен сохраниться после операции. В конце работы деструктор нарушает инвариант, уничтожая объект. Например, конструктор `String::String(const char*)` гарантирует, что p указывает на массив из, по крайней мере, `sz` элементов, причем `sz` имеет осмысленное значение и `v[sz-1]==0`. Любая строковая операция не должна нарушать это утверждение.

При проектировании класса требуется большое искусство, чтобы сделать реализацию класса достаточно простой и допускающей наличие полезных инвариантов, которые несложно задать. Легко требовать, чтобы класс имел инвариант, труднее предложить полезный инвариант, который понятен и не накладывает жестких ограничений на действия разработчика класса или на эффективность реализации. Здесь "инвариант" понимается как программный фрагмент, выполнив который, можно проверить состояние объекта. Вполне возможно дать более строгое и даже математическое определение инварианта, и в некоторых ситуациях оно может оказаться более подходящим. Здесь же под инвариантом понимается практическая, а значит, обычно экономная, но неполная проверка состояния объекта.

Понятие инварианта появилось в работах Флойда, Наура и Хора, посвященных пред- и пост-условиям, оно встречается во всех важных статьях по абстрактным типам данных и верификации программ за последние 20 лет. Оно же является основным предметом отладки в C++.

Обычно, в течение работы функции-члена инвариант не сохраняется. Поэтому функции, которые могут вызываться в те моменты, когда инвариант не действует, не должны входить в общий интерфейс класса. Такие функции должны быть частными или защищенными.

Как можно выразить инвариант в программе на C++? Простое решение - определить функцию, проверяющую инвариант, и вставить вызовы этой функции в общие операции. Например:

```
class String {
    int sz;
    int* p;
public:
    class Range {};
    class Invariant {};

    void check();

    String(const char* q);
    ~String();
    char& operator[](int i);
    int size() { return sz; }
    //...
};

void String::check()
{
    if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz-1])
        throw Invariant;
}

char& String::operator[](int i)
{
```

```
    check();        // проверка на входе
    if (i<0 || i<sz) throw Range;   // действует
    check();        // проверка на выходе
    return v[i];
}
```

Этот вариант прекрасно работает и не осложняет жизнь программиста. Но для такого простого класса как `String` проверка инварианта будет занимать большую часть времени счета. Поэтому программисты обычно выполняют проверку инварианта только при отладке:

```
inline void String::check()
{
    if (!NDEBUG)
        if (p==0 || sz<0 || TOO_LARGE<=sz || p[sz])
            throw Invariant;
}
```

Мы выбрали имя `NDEBUG`, поскольку это макроопределение, которое используется для аналогичных целей в стандартном макроопределении C `assert()`. Традиционно `NDEBUG` устанавливается с целью указать, что отладки нет. Указав, что `check()` является подстановкой, мы гарантировали, что никакая программа не будет создана, пока константа `NDEBUG` не будет установлена в значение, обозначающее отладку. С помощью шаблона типа `Assert()` можно задать менее регулярные утверждения, например:

```
template<class T, class X> inline void Assert(T expr,X x)
{
    if (!NDEBUG)
        if (!expr) throw x;
}
```

вызовет особую ситуацию x, если expr ложно, и мы не отключили проверку с помощью `NDEBUG`. Использовать `Assert()` можно так:

```
class Bad_f_arg { };

void f(String& s, int i)
```

```
{
    Assert(0<=i && i<s.size(),Bad_f_arg());
    //...
}
```

Шаблон типа `Assert()` подражает макрокоманде `assert()` языка C. Если `i` не находится в требуемом диапазоне, возникает особая ситуация `Bad_f_arg`.

С помощью отдельной константы или константы из класса проверить подобные утверждения или инварианты - пустяковое дело. Если же необходимо проверить инварианты с помощью объекта, можно определить производный класс, в котором проверяются операциями из класса, где нет проверки, см. упр.8 в § 13.11.

Для классов с более сложными операциями расходы на проверки могут быть значительны, поэтому проверки можно оставить только для "поимки" трудно обнаруживаемых ошибок. Обычно полезно оставлять по крайней мере несколько проверок даже в очень хорошо отлаженной программе. При всех условиях сам факт определения инвариантов и использования их при отладке дает неоценимую помощь для получения правильной программы и, что более важно, делает понятия, представленные классами, более регулярными и строго определенными. Дело в том, что когда вы создаете инварианты, то рассматриваете класс с другой точки зрения и вносите определенную избыточность в программу. То и другое увеличивает вероятность обнаружения ошибок, противоречий и недосмотров.

Мы указали в § 11.3.3.5, что две самые общие формы преобразования иерархии классов состоят в разбиении класса на два и в выделении общей части двух классов в базовый класс. В обоих случаях хорошо продуманный инвариант может подсказать возможность такого преобразования. Если, сравнивая инвариант с программами операций, можно обнаружить, что большинство проверок инварианта излишни, то значит класс созрел для разбиения. В этом случае подмножество операций имеет доступ только к подмножеству состояний объекта. Обратно, классы созрели для слияния, если у них сходные инварианты, даже при некотором различии в их реализации.

12.2.7.2 Инкапсуляция

Отметим, что в C++ класс, а не отдельный объект, является той единицей, которая должна быть инкапсулирована (заключена в оболочку).

Например:

```
class list {
    list* next;
public:
    int on(list*);
};

int list::on(list* p)
{
    list* q = this;
    for(;;) {
        if (p == q) return 1;
        if (q == 0) return 0;
        q = q->next;
    }
}
```

Здесь обращение к частному указателю `list::next` допустимо, поскольку `list::on()` имеет доступ ко всякому объекту класса `list`, на который у него есть ссылка. Если это неудобно, ситуацию можно упростить, отказавшись от возможности доступа через функцию-член к представлениям других объектов, например:

```
int list::on(list* p)
{
    if (p == this) return  1;
    if (p == 0) return 0;
    return next->on(p);
}
```

Но теперь итерация превращается в рекурсию, что может сильно замедлить выполнение программы, если только транслятор не сумеет

обратно преобразовать рекурсию в итерацию.

12.2.8 Программируемые отношения

Конкретный язык программирования не может прямо поддерживать любое понятие любого метода проектирования. Если язык программирования не способен прямо представить понятие проектирования, следует установить удобное отображение конструкций, используемых в проекте, на языковые конструкции. Например, метод проектирования может использовать понятие делегирования, означающее, что всякая операция, которая не определена для класса `A`, должна выполняться в нем с помощью указателя p на соответствующий член класса `B`, в котором она определена. На C++ нельзя выразить это прямо. Однако, реализация этого понятия настолько в духе C++, что легко представить программу реализации:

```
class A {
    B* p;
    //...
    void f();
    void ff();
};

class B {
    //...
    void f();
    void g();
    void h();
 };
```

Тот факт, что `B` делегирует `A` с помощью указателя `A::p`, выражается в следующей записи:

```
class A {
    B* p;      // делегирование с помощью p
    //...
    void f();
    void ff();
```

```
    void g() { p->g(); }  // делегирование g()
    void h() { p->h(); }  // делегирование h()
};
```

Для программиста совершенно очевидно, что здесь происходит, однако здесь явно нарушается принцип взаимнооднозначного соответствия. Такие "программируемые" отношения трудно выразить на языках программирования, и поэтому к ним трудно применять различные вспомогательные средства. Например, такое средство может не отличить "делегирование" от B к `A` с помощью `A::p` от любого другого использования B*.

Все-таки следует всюду, где это возможно, добиваться взаимнооднозначного соответствия между понятиями проекта и понятиями языка программирования. Оно дает определенную простоту и гарантирует, что проект адекватно отображается в программе, что упрощает работу программиста и вспомогательных средств.

Операции преобразований типа являются механизмом, с помощью которого можно представить в языке класс программируемых отношений, а именно: операция преобразования `X::operator Y()` гарантирует, что всюду, где допустимо использование `Y`, можно применять и `X`. Такое же отношение задает конструктор `Y::Y(X)`. Отметим, что операция преобразования типа (как и конструктор) скорее создает новый объект, чем изменяет тип существующего объекта. Задать операцию преобразования к функции `Y` - означает просто потребовать неявного применения функции, возвращающей `Y`. Поскольку неявные применения операций преобразования типа и операций, определяемых конструкторами, могут привести к неприятностям, полезно проанализировать их в отдельности еще в проекте.

Важно убедиться, что граф применений операций преобразования типа не содержит циклов. Если они есть, возникает двусмысленная ситуация, при которой типы, участвующие в циклах, становятся несовместимыми в комбинации. Например:

```
class Big_int {
    //...
    friend Big_int operator+(Big_int,Big_int);
```

```
    //...
    operator Rational();
    //...
};

class Rational {
    //...
    friend Rational operator+(Rational,Rational);
    //...
    operator Big_int();
};
```

Типы `Rational` и `Big_int` не так гладко взаимодействуют, как можно было бы подумать:

```
void f(Rational r, Big_int i)
{
    //...
    g(r+i);   // ошибка, неоднозначность:
              //       operator+(r,Rational(i)) или
              //       operator+(Big_int(r),i)
    g(r,Rational(i)); // явное разрешение неопределенности
    g(Big_int(r),i);  // еще одно
}
```

Можно было бы избежать таких "взаимных" преобразований, сделав некоторые из них явными. Например, преобразование `Big_int` к типу Rational можно было бы задать явно с помощью функции `make_Rational()` вместо операции преобразования, тогда сложение в приведенном примере разрешалось бы как `g(BIg_int(r),i)`. Если нельзя избежать "взаимных" операций преобразования типов, то нужно преодолевать возникающие столкновения или с помощью явных преобразований (как было показано), или с помощью определения нескольких различных версий бинарной операции (в нашем случае +).

12.3 Компоненты

В языке C++ нет конструкций, которые могут выразить прямо в

программе понятие компонента, т.е. множества связанных классов. Основная причина этого в том, что множество классов (возможно с соответствующими глобальными функциями и т.п.) может соединяться в компонент по самым разным признакам. Отсутствие явного представления понятия в языке затрудняет проведение границы между информацией (имена), используемой внутри компонента, и информацией (имена), передаваемой из компонента пользователям.

В идеале, компонент определяется множеством интерфейсов, используемых для его реализации, плюс множеством интерфейсов, представляемых пользователем, а все прочее считается "спецификой реализации" и должно быть скрыто от остальных частей системы. Таково может быть в действительности представление о компоненте у разработчика.

Рассмотрим два класса, которые должны совместно использовать функцию `f()` и переменную `v`. Проще всего описать `f` и `v` как глобальные имена. Однако, всякий опытный программист знает, что такое "засорение" пространства имен может привести в конце концов к неприятностям: кто-то может ненарочно использовать имена `f` или `v` не по назначению или нарочно обратиться к `f` или `v`, прямо используя "специфику реализации" и обойдя тем самым явный интерфейс компонента. Здесь возможны три решения:

1. Дать "необычные" имена объектам и функциям, которые не рассчитаны на пользователя.
2. Объекты или функции, не предназначенные для пользователя, описать в одном из файлов программы как статические (static).
3. Поместить объекты и функции, не предназначенные для пользователя, в класс, определение которого закрыто для пользователей.

Первое решение примитивно и достаточно неудобно для создателя программы, но оно действует:

```
// не используйте специфику реализации compX,
// если только вы не разработчик compX:
extern void compX_f(T2*, const char*);
extern T3 compX_v;
```

```
// ...
```

Такие имена как `compX_f` и `compX_v` вряд ли могут привести к коллизии, а на тот довод, что пользователь может быть злоумышленником и использовать эти имена прямо, можно ответить, что пользователь в любом случае может оказаться злоумышленником, и что языковые механизмы защиты предохраняют от несчастного случая, а не от злого умысла. Преимущество этого решения в том, что оно применимо всегда и хорошо известно. В то же время оно некрасиво, ненадежно и усложняет ввод текста.

Второе решение более надежно, но менее универсально:

```
// специфика реализации compX:
static void compX_f(T2* a1, const char *a2) { /* ... */ }
static T3 compX_v;
// ...
```

Трудно гарантировать, что информация, используемая в классах одного компонента, будет доступна только в одной единице трансляции, поскольку операции, работающие с этой информацией, должны быть доступны везде. Это решение может к тому же привести к громадным единицам трансляции, а в некоторых отладчиках для C++ не организован доступ к именам статических функций и переменных. В то же время это решение надежно и часто оптимально для небольших компонентов.

Третье решение можно рассматривать как формализацию и обобщение первых двух:

```
class compX_details {  // специфика реализации compX
public:
   static void f(T2*, const char*);
   static T3 v;
   // ...
            };
```

Описание `compX_details` будет использовать только создатель класса, остальные не должны включать его в свои программы.

В компоненте конечно может быть много классов, не предназначенных для общего пользования. Если их имена тоже рассчитаны только на локальное использование, то их также можно "спрятать" внутри классов, содержащих специфику реализации:

```
class compX_details { // специфика реализации compX.
public:
  // ...
  class widget {
  // ...
  };
  // ...
};
```

Укажем, что вложенность создает барьер для использования widget в других частях программы. Обычно классы, представляющие ясные понятия, считаются первыми кандидатами на повторное использование, и, значит составляют часть интерфейса компонента, а не деталь реализации. Другими словами, хотя для сохранения надлежащего уровня абстракции вложенные объекты, используемые для представления некоторого объекта класса, лучше считать скрытыми деталями реализации, классы, определяющие такие вложенные объекты, лучше не делать скрытыми, если они имеют достаточную общность. Так, в следующем примере упрятывание, пожалуй, излишне:

```
class Car {
  class Wheel {
    // ...
  };
  Wheel flw, frw, rlw, rrw;
  // ...
};
```

Во многих ситуациях для поддержания уровня абстракции понятия машины (`Car`) следует упрятывать реальные колеса (класс `Wheel`), ведь когда вы работаете с машиной, вы не можете независимо от нее использовать колеса. С другой стороны, сам класс `Wheel` является вполне подходящим для широкого использования, поэтому лучше вынести его определение из класса `Car`:

```
class Wheel {
  // ...
};
class Car {
   Wheel flw, frw, rlw, rrw;
   // ...
};
```

Использовать ли вложенность? Ответ на этот вопрос зависит от целей проекта и общности используемых понятий. Как вложенность, так и ее отсутствие могут быть вполне допустимыми решениями для данного проекта. Но поскольку вложенность предохраняет от засорения общего пространства имен, в своде правил ниже рекомендуется использовать вложенность, если только нет причин не делать этого.

Отметим, что заголовочные файлы дают мощное средство для различных представлений компонент разным пользователям, и они же позволяют удалять из представления компонента для пользователя те классы, которые связаны со спецификой реализации.

Другим средством построения компонента и представления его пользователю служит иерархия. Тогда базовый класс используется как хранилище общих данных и функций. Таким способом устраняется проблема, связанная с глобальными данными и функциями, предназначенными для реализации общих запросов классов данного компонента.

С другой стороны, при таком решении классы компонента становятся слишком связанными друг с другом, а пользователь попадает в зависимость от всех базовых классов тех компонентов, которые ему действительно нужны. Здесь также проявляется тенденция к тому, что члены, представляющие "полезные" функции и данные "всплывают" к базовому классу, так что при слишком большой иерархии классов проблемы с глобальными данными и функциями проявятся уже в рамках этой иерархии. Вероятнее всего, это произойдет для иерархии с одним корнем, а для борьбы с этим явлением можно применять виртуальные базовые классы (§ 6.5.4). Иногда лучше выбрать иерархию для представления компонента, а иногда нет. Как всегда сделать выбор предстоит разработчику.

12.4 Интерфейсы и реализации

Идеальный интерфейс должен

- представлять полное и согласованное множество понятий для пользователя,
- быть согласованным для всех частей компонента,
- скрывать специфику реализации от пользователя,
- допускать несколько реализаций,
- иметь статическую систему типов,
- определяться с помощью типов из области приложения,
- зависеть от других интерфейсов лишь частично и вполне определенным образом.

Отметив необходимость согласованности для всех классов, которые образуют интерфейс компонента с остальным миром, мы можем упростить вопрос интерфейса, рассмотрев только один класс, например:

```
class X {  // пример плохого определения интерфейса
  Y a;
  Z b;
public:
  void f(const char* ...);
  void g(int[],int);
  void set_a(Y&);
  Y& get_a();
};
```

В этом интерфейсе содержится ряд потенциальных проблем:

- Типы `Y` и `Z` используются так, что определения `Y` и `Z` должны быть известны во время трансляции.
- У функции `X::f` может быть произвольное число параметров неизвестного типа (возможно, они каким-то образом контролируются "строкой формата", которая передается в качестве первого параметра).
- Функция `X::g` имеет параметр типа `int[]`. Возможно это нормально, но обычно это свидетельствует о том, что

определение слишком низкого уровня абстракции. Массив целых не является достаточным определением, так как неизвестно из скольких он может состоять элементов.

- Функции `set_a()` и `get_a()`, по всей видимости, раскрывают представление объектов класса `X`, разрешая прямой доступ к `X::a`.

Здесь функции-члены образуют интерфейс на слишком низком уровне абстракции. Как правило классы с интерфейсом такого уровня относятся к специфике реализации большого компонента, если они вообще могут к чему-нибудь относиться. В идеале параметр функции из интерфейса должен сопровождаться такой информацией, которой достаточно для его понимания. Можно сформулировать такое правило: надо уметь передавать запросы на обслуживание удаленному серверу по узкому каналу.

Язык C++ раскрывает представление класса как часть интерфейса. Это представление может быть скрытым (с помощью private или protected), но обязательно доступным транслятору, чтобы он мог разместить автоматические (локальные) переменные, сделать подстановку тела функции и т.д. Отрицательным следствием этого является то, что использование типов классов в представлении класса может привести к возникновению нежелательных зависимостей. Приведет ли использование членов типа `Y` и `Z` к проблемам, зависит от того, каковы в действительности типы `Y` и `Z`. Если это достаточно простые типы, наподобие `complex` или `String`, то их использование будет вполне допустимым в большинстве случаев. Такие типы можно считать устойчивыми, и необходимость включать определения их классов будет вполне допустимой нагрузкой для транслятора. Если же `Y` и `Z` сами являются классами интерфейса большого компонента (например, типа графической системы или системы обеспечения банковских счетов), то прямую зависимость от них можно считать неразумной. В таких случаях предпочтительнее использовать член, являющийся указателем или ссылкой:

```
class X {
  Y* a;
  Z& b;
  // ...
```

```
};
```

При этом способе определение `X` отделяется от определений `Y` и `Z`, т.е. теперь определение `X` зависит только от имен `Y` и `Z`. Реализация `X`, конечно, будет по-прежнему зависеть от определений `Y` и `Z`, но это уже не будет оказывать неблагоприятного влияния на пользователей `X`.

Вышесказанное иллюстрирует важное утверждение: У интерфейса, скрывающего значительный объем информации (что и должен делать полезный интерфейс), должно быть существенно меньше зависимостей, чем у реализации, которая их скрывает. Например, определение класса `X` можно транслировать без доступа к определениям `Y` и `Z`. Однако, в определениях функций-членов класса `X`, которые работают со ссылками на объекты `Y` и `Z`, доступ к определениям `Y` и `Z` необходим. При анализе зависимостей следует рассматривать раздельно зависимости в интерфейсе и в реализации. В идеале для обоих видов зависимостей граф зависимостей системы должен быть направленным нецикличным графом, что облегчает понимание и тестирование системы. Однако, эта цель более важна и чаще достижима для реализаций, чем для интерфейсов.

Отметим, что класс определяет три интерфейса:

```
class X {
private:
  // доступно только для членов и друзей
protected:
  // доступно только для членов и друзей, а также
  // для членов и друзей производных классов
public:
  // общедоступно
};
```

Члены должны образовывать самый ограниченный из возможных интерфейсов. Иными словами, член должен быть описан как `private`, если нет причин для более широкого доступа к нему; если же таковые есть, то член должен быть описан как `protected`, если нет дополнительных причин задать его как `public`. В большинстве случаев плохо задавать все данные, представляемые членами, как

`public`. Функции и классы, образующие общий интерфейс, должны быть спроектированы таким образом, чтобы представление класса совпадало с его ролью в проекте как средства представления понятий. Напомним, что друзья являются частью общего интерфейса.

Отметим, что абстрактные классы можно использовать для представления понятия упрятывания более высокого уровня (§ 1.4.6, § 6.3, § 13.3).

12.5 Свод правил

В этой лекции мы коснулись многих тем, но, как правило, избегали давать настоятельные и конкретные рекомендации по рассматриваемым вопросам. Это отвечает моему убеждению, что нет "единственно верного решения". Принципы и приемы следует применять способом, наиболее подходящим для конкретной задачи. Здесь требуются вкус, опыт и разум. Тем не менее, можно предложить свод правил, которые разработчик может использовать в качестве ориентиров, пока не приобретет достаточно опыта, чтобы выработать лучшие. Этот свод правил приводится ниже.

Он может служить отправной точкой в процессе выработки основных направлений проекта конкретной задачи, или же он может использоваться организацией в качестве проверочного списка. Подчеркну еще раз, что эти правила не являются универсальными и не могут заменить собой размышления.

- Нацеливайте пользователя на применение абстракции данных и объектно-ориентированного программирования.
    - Постепенно переходите на новые методы, не спешите.
    - Используйте возможности C++ и методы обЪектно-ориентированного программирования только по мере надобности.
    - Добейтесь соответствия стиля проекта и программы.
- Концентрируйте внимание на проектировании компонента.
- Используйте классы для представления понятий.
    - Используйте общее наследование для представления отношений "есть".

- Используйте принадлежность для представления отношений "имеет".
- Убедитесь, что отношения использования понятны, не образуют циклов, и что число их минимально.
- Активно ищите общность среди понятий области приложения и реализации, и возникающие в результате более общие понятия представляйте как базовые классы.

- Определяйте интерфейс так, чтобы открывать минимальное количество требуемой информации:
  - Используйте, всюду где это можно, частные данные и функции-члены.
  - Используйте описания `public` или `protected`, чтобы отличить запросы разработчика производных классов от запросов обычных пользователей.
  - Сведите к минимуму зависимости одного интерфейса от других.
  - Поддерживайте строгую типизацию интерфейсов.
  - Задавайте интерфейсы в терминах типов из области приложения.

Дополнительные правила можно найти § 11.5.

Проектирование библиотек

Эта лекция содержит описание различных приемов, оказавшихся полезными при создании библиотек для языка C++. В частности, в ней рассматриваются конкретные типы, абстрактные типы, узловые классы, управляющие классы и интерфейсные классы. Помимо этого обсуждаются понятия обширного интерфейса и структуры области приложения, использование динамической информации о типах и методы управления памятью. Внимание акцентируется на том, какими свойствами должны обладать библиотечные классы, а не на специфике языковых средств, которые используются для реализации таких классов и не на определенных полезных функциях, которые должна предоставлять библиотека.

13.1 Введение

Проект библиотеки - это проект языка,

(фольклор фирмы Bell Laboratories)

... и наоборот.

А. Кениг

Разработка библиотеки общего назначения - это гораздо более трудная задача, чем создание обычной программы. Программа - это решение конкретной задачи для конкретной области приложения, тогда как библиотека должна предоставлять возможность решение для множества задач, связанных с многими областями приложения. В обычной программе позволительны сильные допущения об ее окружении, тогда как хорошую библиотеку можно успешно использовать в разнообразных окружениях, создаваемых множеством различных программ. Чем более общей и полезной окажется библиотека, тем в большем числе окружений она будет проверяться, и тем жестче будут требования к ее корректности, гибкости, эффективности, расширяемости, переносимости, непротиворечивости, простоте, полноте, легкости использования и т.д. Все же библиотека не может дать вам все, поэтому нужен определенный компромисс. Библиотеку можно рассматривать как специальный, интересный вариант того, что в предыдущей лекции

мы называли компонентом. Каждый совет по проектированию и сопровождению компонентов становится предельно важным для библиотек, и, наоборот, многие методы построения библиотек находят применение при проектировании различных компонентов.

Было бы слишком самонадеянно указывать как следует конструировать библиотеки. В прошлом оказались успешными несколько различных методов, а сам предмет остается полем активных дискуссий и экспериментов. Здесь только обсуждаются некоторые важные аспекты этой задачи и предлагаются некоторые приемы, оказавшиеся полезными при создании библиотек. Не следует забывать, что библиотеки предназначены для совершенно разных областей программирования, поэтому не приходится рассчитывать, что какой-то один метод окажется наиболее приемлемым для всех библиотек. Действительно, нет никаких причин полагать, что методы, оказавшиеся полезными при реализации средств параллельного программирования для ядра многопроцессорной операционной системы, окажутся наиболее приемлемыми при создании библиотеки, предназначенной для решения научных задач, или библиотеки, представляющей графический интерфейс.

Понятие класса C++ может использоваться самыми разными способами, поэтому разнообразие стилей программирования может привести к беспорядку. Хорошая библиотека для сведения такого беспорядка к минимуму обеспечивает согласованный стиль программирования, или, по крайней мере, несколько таких стилей. Этот подход делает библиотеку более "предсказуемой", а значит позволяет легче и быстрее изучить ее и правильно использовать. Далее описываются пять "архетипичных" классов, и обсуждаются присущие им сильные и слабые стороны: конкретные типы (§ 13.2), абстрактные типы (§ 13.3), узловые классы (§ 13.4), интерфейсные классы (§ 13.8), управляющие классы (§ 13.9). Все эти виды классов относятся к области понятий, а не являются конструкциями языка. Каждое понятие воплощается с помощью основной конструкции - класса. В идеале надо иметь минимальный набор простых и ортогональных видов классов, исходя из которого можно построить любой полезный и разумно-определенный класс. Идеал нами не достигнут и, возможно, недостижим вообще. Важно понять, что любой из перечисленных видов классов играет свою роль при проектировании библиотеки и,

если рассчитывать на общее применение, никакой из них не является по своей сути лучше других.

В этой лекции вводится понятие обширного интерфейса (§ 13.6), чтобы выделить некоторый общий случай всех этих видов классов. С помощью него определяется понятие каркаса области приложения (§ 13.7).

Здесь рассматриваются прежде всего классы, относящиеся строго к одному из перечисленных видов, хотя, конечно, используются классы и гибридного вида. Но использование класса гибридного вида должно быть результатом осознанного решения, возникшего при оценке плюсов и минусов различных видов, а не результатом пагубного стремления уклониться от выбора вида класса (слишком часто "отложим пока выбор" означает просто нежелание думать). Неискушенным разработчикам библиотеки лучше всего держаться подальше от классов гибридного вида. Им можно посоветовать следовать стилю программирования той из существующих библиотек, которая обладает возможностями, необходимыми для проектируемой библиотеки. Отважиться на создание библиотеки общего назначения может только искушенный программист, и каждый создатель библиотеки впоследствии будет "осужден" на долгие годы использования, документирования и сопровождения своего собственного создания.

В языке C++ используются статические типы. Однако, иногда возникает необходимость в дополнение к возможностям, непосредственно предоставляемым виртуальными функциями, получать динамическую информацию о типах. Как это сделать, описано в § 13.5. Наконец, перед всякой нетривиальной библиотекой встает задача управления памятью. Приемы ее решения рассматриваются в § 13.10. Естественно, в этой лекции невозможно рассмотреть все методы, оказавшиеся полезными при создании библиотеки. Поэтому можно отослать к другим местам книги, где рассмотрены следующие вопросы: работа с ошибками и устойчивость к ошибкам (§ 9.8), использование функциональных объектов и обратных вызовов (§ 10.4.2 и § 9.4.3) , использование шаблонов типа для построения классов (§ 8.4).

Многие темы этой лекции связаны с классами, являющимися

контейнерами, (например, массивы и списки). Конечно, такие контейнерные классы являются шаблонами типа (как было сказано в § 1.и 4.3 § 8). Но здесь для упрощения изложения в примерах используются классы, содержащие указатели на объекты типа класс. Чтобы получить настоящую программу, надо использовать шаблоны типа, как показано в лекции 8.

13.2 Конкретные типы

Такие классы как vector (§ 1.4), Slist (§ 8.3), date (§ 5.2.2) и complex (§ 7.3) являются конкретными в том смысле, что каждый из них представляет довольно простое понятие и обладает необходимым набором операций. Имеется взаимнооднозначное соответствие между интерфейсом класса и его реализацией. Ни один из них (изначально) не предназначался в качестве базового для получения производных классов. Обычно в иерархии классов конкретные типы стоят особняком. Каждый конкретный тип можно понять изолированно, вне связи с другими классами. Если реализация конкретного типа удачна, то работающие с ним программы сравнимы по размеру и скорости со сделанными вручную программами, в которых используется некоторая специальная версия общего понятия. Далее, если произошло значительное изменение реализации, обычно модифицируется и интерфейс, чтобы отразить эти изменения. Интерфейс, по своей сути, обязан показать, какие изменения оказались существенными в данном контексте. Интерфейс более высокого уровня оставляет больше свободы для изменения реализации, но может ухудшить характеристики программы. Более того, хорошая реализация зависит только от минимального числа действительно существенных классов. Любой из этих классов можно использовать без накладных расходов, возникающих на этапе трансляции или выполнения, и вызванных приспособлением к другим, "сходным" классам программы.

Подводя итог, можно указать такие условия, которым должен удовлетворять конкретный тип:

1. полностью отражать данное понятие и метод его реализации;
2. с помощью подстановок и операций, полностью использующих

полезные свойства понятия и его реализации, обеспечивать эффективность по скорости и памяти, сравнимую с "ручными программами";
3. иметь минимальную зависимость от других классов;
4. быть понятным и полезным даже изолированно.

Все это должно привести к тесной связи между пользователем и программой, реализующей конкретный тип. Если в реализации произошли изменения, программу пользователя придется перетранслировать, поскольку в ней наверняка содержатся вызовы функций, реализуемые подстановкой, а также локальные переменные конкретного типа.

Для некоторых областей приложения конкретные типы обеспечивают основные типы, прямо не представленные в C++, например: комплексные числа, вектора, списки, матрицы, даты, ассоциативные массивы, строки символов и символы, из другого (не английского) алфавита. В мире, состоящем из конкретных понятий, на самом деле нет такой вещи как список. Вместо этого есть множество списочных классов, каждый из которых специализируется на представлении какой-то версии понятия список. Существует дюжина списочных классов, в том числе: список с односторонней связью; список с двусторонней связью; список с односторонней связью, в котором поле связи не принадлежит объекту; список с двусторонней связью, в котором поля связи не принадлежат объекту; список с односторонней связью, для которого можно просто и эффективно определить входит ли в него данный объект; список с двусторонней связью, для которого можно просто и эффективно определить входит ли в него данный объект и т.д.

Название "конкретный тип" (CDT - concrete data type, т.е. конкретный тип данных) , было выбрано по контрасту с термином "абстрактный тип" (ADT - abstract data type, т.е. абстрактный тип данных). Отношения между CDT и ADT обсуждаются в § 13.3.

Существенно, что конкретные типы не предназначены для явного выражения некоторой общности. Так, типы `slist` и `vector` можно использовать в качестве альтернативной реализации понятия множества, но в языке это явно не отражается. Поэтому, если программист хочет работать с множеством, использует конкретные

типы и не имеет определения класса множество, то он должен выбирать между типами `slist` и `vector`. Тогда программа записывается в терминах выбранного класса, скажем, `slist`, и если потом предпочтут использовать другой класс, программу придется переписывать.

Это потенциальное неудобство компенсируется наличием всех "естественных" для данного класса операций, например таких, как индексация для массива и удаление элемента для списка. Эти операции представлены в оптимальном варианте, без "неестественных" операций типа индексации списка или удаления массива, что могло бы вызвать путаницу. Приведем пример:

```
void my(slist& sl)
{
  for (T* p = sl.first(); p; p = sl.next())
  {
    // мой код
  }
  // ...
}

void your(vector& v)
{
  for (int i = 0; i<v.size(); i++)
  {
    // ваш код
  }
  // ...
}
```

Существование таких "естественных" для выбранного метода реализации операций обеспечивает эффективность программы и значительно облегчает ее написание. К тому же, хотя реализация вызова подстановкой обычно возможна только для простых операций типа индексации массива или получения следующего элемента списка, она оказывает значительный эффект на скорость выполнения программы. Загвоздка здесь состоит в том, что фрагменты программы, использующие по своей сути эквивалентные операции, как, например,

два приведенных выше цикла, могут выглядеть непохожими друг на друга, а фрагменты программы, в которых для эквивалентных операций используются разные конкретные типы, не могу заменять друг друга. Обычно, вообще, невозможно свести сходные фрагменты программы в один.

Пользователь, обращающийся к некоторой функции, должен точно указать тип объекта, с которым работает функция, например:

```
void user()
{
  slist sl;
  vector v(100);

  my(sl);
  your(v);

  my(v);    // ошибка: несоответствие типа
  your(sl); // ошибка: несоответствие типа
}
```

Чтобы компенсировать жесткость этого требования, разработчик некоторой полезной функции должен предоставить несколько ее версий, чтобы у пользователя был выбор:

```
void my(slist&);
void my(vector&);

void your(slist&);
void your(vector&);

void user()
{
  slist sl;
  vector v(100);

  my(sl);
  your(v);
```

```
  my(v);    // теперь нормально: вызов my(vector&)
  your(sl); // теперь нормально: вызов your(slist&)
}
```

Поскольку тело функции существенно зависит от типа ее параметра, надо написать каждую версию функций `my()` и `your()` независимо друг от друга, что может быть хлопотно.

С учетом всего изложенного, конкретный тип, можно сказать, походит на встроенные типы. Положительной стороной этого является тесная связь между пользователем типа и его создателем, а также между пользователями, которые создают объекты данного типа, и пользователями, которые пишут функции, работающие с этими объектами. Чтобы правильно использовать конкретный тип, пользователь должен разбираться в нем детально. Обычно не существует каких-то универсальных свойств, которыми обладали бы все конкретные типы библиотеки, и что позволило бы пользователю, рассчитывая на эти свойства, не тратить силы на изучение отдельных классов. Такова плата за компактность программы и эффективность ее выполнения. Иногда это вполне разумная плата, иногда нет. Кроме того, возможен такой случай, когда отдельный конкретный класс проще понять и использовать, чем более общий (абстрактный) класс. Именно так бывает с классами, представляющими хорошо известные типы данных, такие как массивы или списки.

Тем не менее, укажем, что в идеале надо скрывать, насколько возможно, детали реализации, пока это не ухудшает характеристики программы. Большую помощь здесь оказывают функции-подстановки. Если сделать открытыми переменные, являющиеся членами, с помощью описания public, или непосредственно работать с ними с помощью функций, которые устанавливают и получают значения этих переменных, то почти всегда это приводит к плохому результату. Конкретные типы должны быть все-таки настоящими типами, а не просто программной кучей с нескольким функциями, добавленными ради удобства.

13.3 Абстрактные типы

Самый простой способ ослабить связь между пользователем класса и его создателем, а также между программами, в которых объекты

создаются, и программами, в которых они используются, состоит в введении понятия абстрактных базовых классов. Эти классы представляют интерфейс со множеством реализаций одного понятия. Рассмотрим класс `set`, содержащий множество объектов типа `T`:

```
class set {
public:
    virtual void insert(T*) = 0;
    virtual void remove(T*) = 0;

    virtual int is_member(T*) = 0;

    virtual T* first() = 0;
    virtual T* next() = 0;

    virtual ~set() { }
};
```

Этот класс определяет интерфейс с произвольным множеством (set), опираясь на встроенное понятие итерации по элементам множества. Здесь типично отсутствие конструктора и наличие виртуального деструктора, см. также § 6.7. Рассмотрим пример:

```
class slist_set : public set, private slist {
    slink* current_elem;
public:
    void insert(T*);
    void remove(T*);

    int is_member(T*);

    virtual T* first();
    virtual T* next();

    slist_set() : slist(), current_elem(0) { }
};

class vector_set : public set, private vector {
    int current_index;
```

```
public:
    void insert(T*);
    void remove(T*);

    int is_member(T*);

    T* first() { current_index = 0; return next(); }
    T* next();

    vector_set(int initial_size)
      : array(initial_size), current_index(0) { }
};
```

Реализация конкретного типа используется как частный базовый класс, а не член класса. Это сделано и для удобства записи, и потому, что некоторые конкретные типы могут иметь защищенный интерфейс с целью предоставить более прямой доступ к своим членам из производных классов. Кроме того, подобным образом в реализации могут использоваться некоторые классы, которые имеют виртуальные функции и не являются конкретными типами. Только с помощью образования производных классов можно в новом классе изящно переопределить (подавить) виртуальную функцию класса реализации. Интерфейс определяется абстрактным классом.

Теперь пользователь может записать свои функции из § 13.2 таким образом:

```
void my(set& s)
{
  for (T* p = s.first(); p; p = s.next())
  {
    // мой код
  }
  // ...
}

void your(set& s)
{
  for (T* p = s.first(); p; p = s.next())
```

```
  {
    // ваш код
  }
  // ...
}
```

Стало очевидным сходство между двумя функциями, и теперь достаточно иметь только одну версию для каждой из функций `my()` или `your()`, поскольку для общения с `slist_set` и `vector_set` обе версии используют интерфейс, определяемый классом `set`:

```
void user()
{
  slist_set sl;
  vector_set v(100);

  my(sl);
  your(v);

  my(v);
  your(sl);
}
```

Более того, создатели функций `my()` и `your()` не обязаны знать описаний классов `slist_set` и `vector_set`, и функции `my()` и `your()` никоим образом не зависят от этих описаний. Их не надо перетранслировать или как-то изменять, ни если изменились классы `slist_set` или `vector_set` ни даже, если предложена новая реализация этих классов. Изменения отражаются лишь на функциях, которые непосредственно используют эти классы, допустим `vector_set`. В частности, можно воспользоваться традиционным применением заголовочных файлов и включить в программы с функциями `my()` или `your()` файл определений `set.h`, а не файлы `slist_set.h` или `vector_set.h`.

В обычной ситуации операции абстрактного класса задаются как чистые виртуальные функции, и такой класс не имеет членов, представляющих данные (не считая скрытого указателя на таблицу виртуальных функций). Это объясняется тем, что добавление

невиртуальной функции или члена, представляющего данные, потребует определенных допущений о классе, которые будут ограничивать возможные реализации. Изложенный здесь подход к абстрактным классам близок по духу традиционным методам, основанным на строгом разделении интерфейса и его реализаций. Абстрактный тип служит в качестве интерфейса, а конкретные типы представляют его реализации.

Такое разделение интерфейса и его реализаций предполагает недоступность операций, являющихся "естественными" для какой-то одной реализации, но не достаточно общими, чтобы войти в интерфейс. Например, поскольку в произвольном множестве нет упорядоченности, в интерфейс `set` нельзя включать операцию индексирования, даже если для реализации конкретного множества используется массив. Это приводит к ухудшению характеристик программы из-за отсутствия ручной оптимизации. Далее, становится как правило невозможной реализация функций подстановкой (если не считать каких-то конкретных ситуаций, когда настоящий тип известен транслятору), поэтому все полезные операции интерфейса, задаются как вызовы виртуальных функций. Как и для конкретных типов здесь плата за абстрактные типы иногда приемлема, иногда слишком высока.

Подводя итог, перечислим каким целям должен служить абстрактный тип:

1. определять некоторое понятие таким образом, что в программе могут сосуществовать для него несколько реализаций;
2. применяя виртуальные функции, обеспечивать достаточно высокую степень компактности и эффективности выполнения программы;
3. сводить к минимуму зависимость любой реализации от других классов;
4. представлять само по себе осмысленное понятие.

Нельзя сказать, что абстрактные типы лучше конкретных типов, это просто другие типы. Какие из них предпочесть - это, как правило, трудный и важный вопрос для пользователя. Создатель библиотеки может уклониться от ответа на него и предоставить варианты с обеими типами, тем самым выбор перекладывается на пользователя. Но здесь

важно ясно понимать, с классом какого вида имеешь дело. Обычно неудачей заканчивается попытка ограничить общность абстрактного типа, чтобы скорость программ, работающих с ним, приблизилась к скорости программ, рассчитанных на конкретный тип. В этом случае нельзя использовать взаимозаменяемые реализации без большой перетрансляции программы после внесения изменений. Столь же неудачна бывает попытка дать "общность" в конкретных типах, чтобы они могли по мощности понятий приблизиться к абстрактным типам. Это снижает эффективность и применимость простых классов. Классы этих двух видов могут сосуществовать, и они должны мирно сосуществовать в программе. Конкретный класс воплощает реализацию абстрактного типа, и смешивать его с абстрактным классом не следует.

Отметим, что ни конкретные, ни абстрактные типы не создаются изначально как базовые классы для построения в дальнейшем производных классов. Построение производных к абстрактным типам классов скорее нужно для задания реализаций, чем для развития самого понятия интерфейса. Всякий конкретный или абстрактный тип предназначен для четкого и эффективного представления в программе отдельного понятия. Классы, которым это удается, редко бывают хорошими кандидатами для создания на их базе новых, но связанных с ними, классов. Действительно, попытки построить производные, "более развитые" классы на базе конкретных или абстрактных типов, таких как, строки, комплексные числа, списки или ассоциативные массивы приводят обычно к громоздким конструкциям. Как правило эти классы следует использовать как члены или частные базовые классы, тогда их можно эффективно применять, не вызывая путаницы и противоречий в интерфейсах и реализациях этих и новых классов.

Когда создается конкретный или абстрактный тип, акцент следует сделать на том, чтобы предложить простой, реализующий хорошо продуманное понятие, интерфейс. Попытки расширить область приложения класса, нагружая его описание всевозможными "полезными" свойствами, приводят только к беспорядку и неэффективности. Этим же кончаются напрасные усилия гарантировать повторное использование класса, когда каждую функцию-член объявляют виртуальной, не подумав зачем и как эти функции будут переопределяться.

Почему мы не стали определять классы `slist` и `vector` как прямые производные от класса `set`, обойдясь тем самым без классов `slist_set` и `vector_set`? Другими словами зачем нужны конкретные типы, когда уже определены абстрактные типы? Можно предложить три ответа:

1. Эффективность: такие типы, как `vector` или `slist` надо создавать без накладных расходов, вызванных отдалением реализаций от интерфейсов (разделения интерфейса и реализации требует концепция абстрактного типа).
2. Множественный интерфейс: часто разные понятия лучше всего реализовать как производные от одного класса.
3. Повторное использование: нужен механизм, который позволит приспособить для нашей библиотеки типы, разработанные "где-то в другом месте".

Конечно, все эти ответы связаны. В качестве примера [2] рассмотрим понятие генератора итераций. Требуется определить генератор итераций (в дальнейшем итератор) для любого типа так, чтобы с его помощью можно было порождать последовательность объектов этого типа. Естественно для этого нужно использовать уже упоминавшийся класс `slist`. Однако, нельзя просто определить общий итератор над `slist`, или даже над `set`, поскольку общий итератор должен допускать итерации и более сложных объектов, не являющихся множествами, например, входные потоки или функции, которые при очередном вызове дают следующее значение итерации. Значит нам нужны и множество и итератор, и в тоже время нежелательно дублировать конкретные типы, которые являются очевидными реализациями различных видов множеств и итераторов. Можно графически представить желательную структуру классов так:

Здесь классы `set` и `iter` предоставляют интерфейсы, а `slist` и `stream` являются частными классами и представляют реализации. Очевидно, нельзя перевернуть эту иерархию классов и, предоставляя общие интерфейсы, строить производные конкретные типы от абстрактных классов. В такой иерархии каждая полезная операция над каждым полезным абстрактным понятием должна представляться в общем абстрактном базовом классе. Дальнейшее обсуждение этой темы

содержится в § 13.6.

Приведем пример простого абстрактного типа, являющегося итератором объектов типа T:

```
class iter {
  virtual T* first() = 0;
  virtual T* next() = 0;
  virtual ~iter() { }
};

class slist_iter : public iter, private slist {
  slink* current_elem;
public:
  T* first();
  T* next();

  slist_iter() : current_elem(0) { }
};

class input_iter : public iter {
  isstream& is;
public:
  T* first();
  T* next();

  input_iter(istream& r) : is(r) { }
};
```

Можно таким образом использовать определенные нами типы:

```
void user(const iter& it)
{
 for (T* p = it.first(); p; p = it.next()) {
    // ...
 }
}

void caller()
```

```
{
  slist_iter sli;
  input_iter ii(cin);

  // заполнение sli

  user(sli);
  user(ii);
}
```

Мы применили конкретный тип для реализации абстрактного типа, но можно использовать его и независимо от абстрактных типов или просто вводить такие типы для повышения эффективности программы, см. также § 13.5. Кроме того, можно использовать один конкретный тип для реализации нескольких абстрактных типов.

В разделе § 13.9 описывается более гибкий итератор. Для него зависимость от реализации, которая поставляет подлежащие итерации объекты, определяется в момент инициализации и может изменяться в ходе выполнения программы.

13.4 Узловые классы

В действительности иерархия классов строится, исходя из совсем другой концепции производных классов, чем концепция интерфейс-реализация, которая использовалась для абстрактных типов. Класс рассматривается как фундамент строения. Но даже, если в основании находится абстрактный класс, он допускает некоторое представление в программе и сам предоставляет для производных классов какие-то полезные функции. Примерами узловых классов могут служить классы `rectangle` (§ 6.4.2) и `satellite` (§ 6.5.1). Обычно в иерархии класс представляет некоторое общее понятие, а производные классы представляют конкретные варианты этого понятия. Узловой класс является неотъемлемой частью иерархии классов. Он пользуется сервисом, представляемым базовыми классами, сам обеспечивает определенный сервис и предоставляет виртуальные функции и (или) защищенный интерфейс, чтобы позволить дальнейшую детализацию своих операций в производных классах.

Типичный узловой класс не только предоставляет реализацию интерфейса, задаваемого его базовым классом (как это делает класс реализации по отношению к абстрактному типу), но и сам расширяет интерфейс, добавляя новые функции. Рассмотрим в качестве примера класс `dialog_box`, который представляет окно некоторого вида на экране. В этом окне появляются вопросы пользователю и в нем он задает свой ответ с помощью нажатия клавиши или "мыши":

```
class dialog_box : public window {
  // ...
public:
  dialog_box(const char* ...); // заканчивающийся нулем список
      // обозначений клавиш
  // ...
virtual int ask();
        };
```

Здесь важную роль играет функция `ask()` и конструктор, с помощью которого программист указывает используемые клавиши и задает их числовые значения. Функция `ask()` изображает на экране окно и возвращает номер нажатой в ответ клавиши. Можно представить такой вариант использования:

```
void user()
{
  for (;;) {
    // какие-то команды

    dialog_box cont("continue",
      "try again",
      "abort",
      (char*) 0);
    switch (cont.ask()) {
    case 0: return;
    case 1: break;
    case 2: abort();
    }
  }
}
```

Обратим внимание на использование конструктора. Конструктор, как правило, нужен для узлового класса и часто это нетривиальный конструктор. Этим узловые классы отличаются от абстрактных классов, для которых редко нужны конструкторы.

Пользователь класса `dialog_box` (а не только создатель этого класса) рассчитывает на сервис, представляемый его базовыми классами. В рассматриваемом примере предполагается, что существует некоторое стандартное размещение нового окна на экране. Если пользователь захочет управлять размещением окна, базовый для `dialog_box` класс `window` (окно) должен предоставлять такую возможность, например:

```
dialog_box cont("continue","try again","abort",(char*)0);
cont.move(some_point);
```

Здесь функция движения окна `move()` рассчитывает на определенные функции базовых классов.

Сам класс `dialog_box` является хорошим кандидатом для построения производных классов. Например, вполне разумно иметь такое окно, в котором, кроме нажатия клавиши или ввода с мышью, можно задавать строку символов (скажем, имя файла). Такое окно `dbox_w_str` строится как производный класс от простого окна `dialog_box`:

```
class dbox_w_str : public dialog_box {
    // ...
public:
    dbox_w_str (
     const char* sl,  // строка запроса пользователю
     const char* ...  // список обозначений клавиш
    );
    int ask();
    virtual char* get_string();
    //...
};
```

Функция `get_string()` является той операцией, с помощью которой программист получает заданную пользователем строку. Функция `ask()` из класса `dbox_w_str` гарантирует, что строка введена

правильно, а если пользователь не стал вводить строку, то тогда в программу возвращается соответствующее значение (0).

```
void user2()
{
  // ...
  dbox_w_str file_name("please enter file name",
          "done",
          (char*)0);
  file_name.ask();
  char* p = file_name.get_string();
  if (p) {
    // используем имя файла
  }
  else {
    // имя файла не задано
  }
  //
}
```

Подведем итог - узловой класс должен:

1. рассчитывать на свои базовые классы как для их реализации, так и для представления сервиса пользователям этих классов;
2. представлять более полный интерфейс (т.е. интерфейс с большим числом функций-членов) пользователям, чем базовые классы;
3. основывать в первую очередь (но не исключительно) свой общий интерфейс на виртуальных функциях;
4. зависеть от всех своих (прямых и косвенных) базовых классов;
5. иметь смысл только в контексте своих базовых классов;
6. служить базовым классом для построения производных классов;
7. воплощаться в объекте.

Не все, но многие, узловые классы будут удовлетворять условиям 1, 2, 6 и 7. Класс, который не удовлетворяет условию 6, походит на конкретный тип и может быть назван конкретным узловым классом. Класс, который не удовлетворяет условию 7, походит на абстрактный тип и может быть назван абстрактным узловым классом. У многих узловых классов есть защищенные члены, чтобы предоставить для

производных классов менее ограниченный интерфейс.

Укажем на следствие условия 4: для трансляции своей программы пользователь узлового класса должен включить описания всех его прямых и косвенных базовых классов, а также описания всех тех классов, от которых, в свою очередь, зависят базовые классы. В этом узловой класс опять представляет контраст с абстрактным типом. Пользователь абстрактного типа не зависит от всех классов, использующихся для реализации типа, и для трансляции своей программы не должен включать их описания.

13.5 Динамическая информация о типе

Иногда бывает полезно знать истинный тип объекта до его использования в каких-либо операциях. Рассмотрим функцию my(set&) из § 13.3.

```
void my_set(set& s)
{
  for ( T* p = s.first(); p; p = s.next()) {
    // мой код
  }
  // ...
}
```

Она хороша в общем случае, но представим,- стало известно, что многие параметры множества представляют собой объекты типа `slist`. Возможно также стал известен алгоритм перебора элементов, который значительно эффективнее для списков, чем для произвольных множеств. В результате эксперимента удалось выяснить, что именно этот перебор является узким местом в системе. Тогда, конечно, имеет смысл учесть в программе отдельно вариант с `slist`. Допустив возможность определения истинного типа параметра, задающего множество, функцию `my` (set&) можно записать так:

```
void my(set& s)
{
  if (ref_type_info(s) == static_type_info(slist_set)) {
```

```
        // сравнение двух представлений типа

        // s типа slist

        slist& sl = (slist&)s;
        for (T* p = sl.first(); p; p = sl.next()) {

          // эффективный вариант в расчете на list

        }
    }
    else {

      for ( T* p = s.first(); p; p = s.next()) {

          // обычный вариант для произвольного множества

      }
    }
    // ...
          }
```

Как только стал известен конкретный тип `slist`, стали доступны определенные операции со списками, и даже стала возможна реализация основных операций подстановкой.

Приведенный вариант функции действует отлично, поскольку `slist` - это конкретный класс, и действительно имеет смысл отдельно разбирать вариант, когда параметр является `slist_set`. Рассмотрим теперь такую ситуацию, когда желательно отдельно разбирать вариант как для класса, так и для всех его производных классов. Допустим, мы имеем класс `dialog_box` из § 13.4 и хотим узнать, является ли он классом `dbox_w_str`. Поскольку может существовать много производных классов от `dbox_w_str`, простую проверку на совпадение с ним нельзя считать хорошим решением. Действительно, производные классы могут представлять самые разные варианты запроса строки. Например, один производный от `dbox_w_str` класс может предлагать пользователю варианты строк на выбор, другой

может обеспечить поиск в каталоге и т.д. Значит, нужно проверять и на совпадение со всеми производными от `dbox_w_str` классами. Это так же типично для узловых классов, как проверка на вполне определенный тип типична для абстрактных классов, реализуемых конкретными типами.

```
void f(dialog_box& db)
{
  dbox_w_str* dbws = ptr_cast(dbox_w_str, &db);
  if (dbws) {  // dbox_w_str
    // здесь можно использовать dbox_w_str::get_string()
  }
  else {

    // ``обычный'' dialog_box
  }

  // ...
}
```

Здесь "операция" приведения `ptr_cast()` свой второй параметр (указатель) приводит к своему первому параметру (типу) при условии, что указатель настроен на объект, тип которого совпадает с заданным (или является производным классом от заданного типа). Для проверки типа `dialog_box` используется указатель, чтобы после приведения его можно было сравнить с нулем.

Возможно альтернативное решение с помощью ссылки на `dialog_box`:

```
void g(dialog_box& db)
{
 try {
    dbox_w_str& dbws = ref_cast(dialog_box,db);

    // здесь можно использовать dbox_w_str::get_string()

  }
  catch (Bad_cast) {
```

```
        // ``обычный" dialog_box

    }

    // ...
}
```

Поскольку нет приемлемого представления нулевой ссылки, с которой можно сравнивать, используется особая ситуация, обозначающая ошибку приведения (т.е. случай, когда тип не есть `dbox_w_str`). Иногда лучше избегать сравнения с результатом приведения.

Различие функций `ref_cast()` и `ptr_cast()` служит хорошей иллюстрацией различий между ссылками и указателями: ссылка обязательно ссылается на объект, тогда как указатель может и не ссылаться, поэтому для указателя часто нужна проверка.

13.5.1 Информация о типе

В C++ нет иного стандартного средства получения динамической информации о типе, кроме вызовов виртуальных функций.

Хотя было сделано несколько предложений по расширению C++ в этом направлении.

Смоделировать такое средство довольно просто и в большинстве больших библиотек есть возможности динамических запросов о типе. Здесь предлагается решение, обладающее тем полезным свойством, что объем информации о типе можно произвольно расширять. Его можно реализовать с помощью вызовов виртуальных функций, и оно может входить в расширенные реализации C++.

Достаточно удобный интерфейс с любым средством, поставляющим информацию о типе, можно задать с помощью следующих операций:

```
typeid static_type_info(type) // получить typeid для имени типа
typeid ptr_type_info(pointer) // получить typeid для указателя
```

```
typeid ref_type_info(reference) // получить typeid для ссылки
pointer ptr_cast(type,pointer)  // преобразование указателя
reference ref_cast(type,reference)  // преобразование ссылки
```

Пользователь класса может обойтись этими операциями, а создатель класса должен предусмотреть в описаниях классов определенные "приспособления", чтобы согласовать операции с реализацией библиотеки.

Большинство пользователей, которым вообще нужна динамическая идентификация типа, может ограничиться операциями приведения `ptr_cast()` и `ref_cast()`. Таким образом пользователь отстраняется от дальнейших сложностей, связанных с динамической идентификацией типа. Кроме того, ограниченное использование динамической информации о типе меньше всего чревато ошибками.

Если недостаточно знать, что операция приведения прошла успешно, а нужен истинный тип (например, объектно-ориентированный ввод-вывод), то можно использовать операции динамических запросов о типе: `static_type_info()`, `ptr_type_info()` и `ref_type_info()`. Эти операции возвращают объект класса `typeid`. Как было показано в примере с `set` и `slist_set`, объекты класса `typeid` можно сравнивать. Для большинства задач этих сведений о классе `typeid` достаточно. Но для задач, которым нужна более полная информация о типе, в классе `typeid` есть функция `get_type_info()`:

```
class typeid {
    friend class Type_info;
private:
    const Type_info* id;
public:
    typeid(const Type_info* p) : id(p) { }
    const Type_info* get_type_info() const { return id; }
    int operator==(typeid i) const ;
};
```

Функция `get_type_info()` возвращает указатель на неменяющийся (const) объект класса `Type_info` из `typeid`. Существенно, что объект

не меняется: это должно гарантировать, что динамическая информация о типе отражает статические типы исходной программы. Плохо, если при выполнении программы некоторый тип может изменяться.

С помощью указателя на объект класса `Type_info` пользователь получает доступ к информации о типе из `typeid` и, теперь его программа начинает зависеть от конкретной системы динамических запросов о типе и от структуры динамической информации о нем. Но эти средства не входят в стандарт языка, а задать их с помощью хорошо продуманных макроопределений непросто.

13.5.2 Класс Type_info

В классе `Type_info` есть минимальный объем информации для реализации операции `ptr_cast()` ; его можно определить следующим образом:

```
class Type_info {
    const char* n;       // имя
    const Type_info** b; // список базовых классов
public:
    Type_info(const char* name, const Type_info* base[]);

    const char* name() const;
    Base_iterator bases(int direct=0) const;
    int same(const Type_info* p) const;
    int has_base(const Type_info*, int direct=0) const;
    int can_cast(const Type_info* p) const;

    static const Type_info info_obj;
    virtual typeid get_info() const;
    static typeid info();
};
```

Две последние функции должны быть определены в каждом производном от `Type_info` классе.

Пользователь не должен заботиться о структуре объекта `Type_info`, и

она приведена здесь только для полноты изложения. Строка, содержащая имя типа, введена для того, чтобы дать возможность поиска информации в таблицах имен, например, в таблице отладчика. С помощью нее а также информации из объекта `Type_info` можно выдавать более осмысленные диагностические сообщения. Кроме того, если возникнет потребность иметь несколько объектов типа `Type_info`, то имя может служить уникальным ключом этих объектов.

```
const char* Type_info::name() const
{
  return n;
}

int Type_info::same(const Type_info* p) const
{
  return this==p || strcmp(n,p->n)==0;
}

int Type_info::can_cast(const Type_info* p) const
{
  return same(p) || p->has_base(this);
}
```

Доступ к информации о базовых классах обеспечивается функциями `bases()` и `has_base()`. Функция `bases()` возвращает итератор, который порождает указатели на базовые классы объектов `Type_info`, а с помощью функции `has_base()` можно определить является ли заданный класс базовым для другого класса. Эти функции имеют необязательный параметр `direct`, который показывает, следует ли рассматривать все базовые классы (`direct=0`), или только прямые базовые классы (`direct=1`). Наконец, как описано ниже, с помощью функций `get_info()` и `info()` можно получить динамическую информацию о типе для самого класса `Type_info`.

Здесь средство динамических запросов о типе сознательно реализуется с помощью совсем простых классов. Так можно избежать привязки к определенной библиотеке. Реализация в расчете на конкретную библиотеку может быть иной. Можно, как всегда, посоветовать

пользователям избегать излишней зависимости от деталей реализации.

Функция `has_base()` ищет базовые классы с помощью имеющегося в `Type_info` списка базовых классов. Хранить информацию о том, является ли базовый класс частным или виртуальным, не нужно, поскольку все ошибки, связанные с ограничениями доступа или неоднозначностью, будут выявлены при трансляции.

```
class base_iterator {
  short i;
  short alloc;
  const Type_info* b;
public:
  const Type_info* operator() ();
  void reset() { i = 0; }

  base_iterator(const Type_info* bb, int direct=0);
  ~base_iterator() { if (alloc) delete[] (Type_info*)b; }
            };
```

В следующем примере используется необязательный параметр для указания, следует ли рассматривать все базовые классы (`direct==0`) или только прямые базовые классы (`direct==1`).

```
base_iterator::base_iterator(const Type_info* bb, int direct)
{
  i = 0;

  if (direct) { // использование списка прямых базовых классов
    b = bb;
    alloc = 0;
    return;
  }

  // создание списка прямых базовых классов:

  // int n = число базовых
  b = new const Type_info*[n+1];
  // занести базовые классы в b
```

```
  alloc = 1;
  return;
}

const Type_info* base_iterator::operator() ()
{
  const Type_info* p = &b[i];
  if (p) i++;
  return p;
}
```

Теперь можно задать операции запросов о типе с помощью макроопределений:

```
#define static_type_info(T)  T::info()

#define ptr_type_info(p)   ((p)->get_info())
#define ref_type_info(r)   ((r).get_info())

#define ptr_cast(T,p) \
   (T::info()->can_cast((p)->get_info()) ? (T*)(p) : 0)
#define ref_cast(T,r) \
   (T::info()->can_cast((r).get_info()) \
      ? 0 : throw Bad_cast(T::info()->name()), (T&)(r))
```

Предполагается, что тип особой ситуации `Bad_cast` (Ошибка_приведения) описан так:

```
class Bad_cast {
  const char* tn;
  // ...
public:
  Bad_cast(const char* p) : tn(p) { }
  const char* cast_to() { return tn; }
  // ...
};
```

В разделе § 4.7 было сказано, что появление макроопределений

служит сигналом возникших проблем. Здесь проблема в том, что только транслятор имеет непосредственный доступ к литеральным типам, а макроопределения скрывают специфику реализации. По сути для хранения информации для динамических запросов о типах предназначена таблица виртуальных функций. Если реализация непосредственно поддерживает динамическую идентификацию типа, то рассматриваемые операции можно реализовать более естественно, эффективно и элегантно. В частности, очень просто реализовать функцию `ptr_cast()`, которая преобразует указатель на виртуальный базовый класс в указатель на его производные классы.

13.5.3 Как создать систему динамических запросов о типе

Здесь показано, как можно прямо реализовать динамические запросы о типе, когда в трансляторе таких возможностей нет. Это достаточно утомительная задача и можно пропустить этот раздел, так как в нем есть только детали конкретного решения.

Классы `set` и `slist_set` из § 13.3 следует изменить так, чтобы с ними могли работать операции запросов о типе. Прежде всего, в базовый класс `set` нужно ввести функции-члены, которые используют операции запросов о типе:

```
class set {
  public:
    static const Type_info info_obj;
    virtual typeid get_info() const;
    static typeid info();

    // ...
            };
```

При выполнении программы единственным представителем объекта типа `set` является `set::info_obj`, который определяется так:

```
const Type_info set::info_obj("set",0);
```

С учетом этого определения функции тривиальны:

```
typeid set::get_info() const { return &info_obj; }
typeid set::info() { return &info_obj; }
typeid slist_set::get_info() const { return &info_obj; }
typeid slist_set::info() { return &info_obj; }
```

Виртуальная функция `get_info()` будет предоставлять операции `ref_type_info()` и `ptr_type_info()`, а статическая функция `info()` - операцию `static_type_info()`.

При таком построении системы запросов о типе основная трудность на практике состоит в том, чтобы для каждого класса объект типа `Type_info` и две функции, возвращающие указатель на этот объект, определялись только один раз.

Нужно несколько изменить класс `slist_set`:

```
class slist_set : public set, private slist {
  // ...
public:
  static const Type_info info_obj;
  virtual typeid get_info() const;
  static typeid info();

  // ...
};

static const Type_info* slist_set_b[]
  = { &set::info_obj, &slist::info_obj, 0 };
const Type_info slist_set::info_obj("slist_set",slist_set_b);

typeid slist_set::get_info() const { return &info_obj; }
typeid slist_set::info() { return &info_obj; }
```

13.5.4 Расширенная динамическая информация о типе

В классе `Type_info` содержится только минимум информации,

необходимой для идентификации типа и безопасных операций приведения. Но поскольку в самом классе `Type_info` есть функции-члены `info()` и `get_info()`, можно построить производные от него классы, чтобы в динамике определять, какие объекты `Type_info` возвращают эти функции. Таким образом, не меняя класса `Type_info`, пользователь может получать больше информации о типе с помощью объектов, возвращаемых функциями `dynamic_type()` и `static_type()`. Во многих случаях дополнительная информация должна содержать таблицу членов объекта:

```
struct Member_info {
  char* name;
  Type_info* tp;
  int offset;
};

class Map_info : public Type_info {
   Member_info** mi;
public:
   static const Type_info info_obj;
   virtual typeid get_info() const;
   static typeid info();

   // функции доступа
};
```

Класс `Type_info` вполне подходит для стандартной библиотеки. Это базовый класс с минимумом необходимой информации, из которого можно получать производные классы, предоставляющие больше информации. Эти производные классы могут определять или сами пользователи, или какие-то служебные программы, работающие с текстом на C++, или сами трансляторы языка.

13.5.5 Правильное и неправильное использование динамической информации о типе

Динамическая информация о типе может использоваться во многих

ситуациях, в том числе для: объектного ввода-вывода, объектно-ориентированных баз данных, отладки. В тоже время велика вероятность ошибочного использования такой информации. Известно,что в языке Симула использование таких средств, как правило, приводит к ошибкам. Поэтому эти средства не были включены в C++. Слишком велик соблазн воспользоваться динамической информацией о типе, тогда как правильнее вызвать виртуальную функцию. Рассмотрим в качестве примера класс `Shape` из § 1.2.5. Функцию `rotate` можно было задать так:

```
void rotate(const Shape& s)
    // неправильное использование динамической
    // информации о типе

{
  if (ref_type_info(s)==static_type_info(Circle)) {
    // для этой фигуры ничего не надо
  }
  else if (ref_type_info(s)==static_type_info(Triangle)) {
    // вращение треугольника
  }
  else if (ref_type_info(s)==static_type_info(Square)) {
    // вращение квадрата
  }
  // ...
}
```

Если для переключателя по типу поля мы используем динамическую информацию о типе, то тем самым нарушаем в программе принцип модульности и отрицаем сами цели объектно-ориентированного программирования. К тому же это решение чревато ошибками: если в качестве параметра функции будет передан объект производного от `Circle` класса, то она сработает неверно (действительно, вращать круг (Circle) нет смысла, но для объекта, представляющего производный класс, это может потребоваться). Опыт показывает, что программистам, воспитанным на таких языках как C или Паскаль, трудно избежать этой ловушки. Стиль программирования этих языков требует меньше предусмотрительности, а при создании библиотеки такой стиль можно просто считать небрежностью.

Может возникнуть вопрос, почему в интерфейс с системой динамической информации о типе включена условная операция приведения `ptr_cast()`, а не операция `is_base()`, которая непосредственно определяется с помощью операции `has_base()` из класса `Type_info`. Рассмотрим такой пример:

```
void f(dialog_box& db)
{
  if (is_base(&db,dbox_w_str)) { // является ли db базовым
                // для  dbox_w-str?
    dbox_w_str* dbws = (dbox_w_str*) &db;
    // ...
  }

  // ...
}
```

Решение с помощью `ptr_cast` (§ 13.5) более короткое, к тому же здесь явная и безусловная операция приведения отделена от проверки в операторе if, значит появляется возможность ошибки, неэффективности и даже неверного результата. Неверный результат может возникнуть в тех редких случаях, когда система динамической идентификации типа распознает, что один тип является производным от другого, но транслятору этот факт неизвестен, например:

```
class D;
class B;

void g(B* pb)
{
  if (is_base(pb,D)) {
    D* pb = (D*)pb;

    // ...
  }

  // ...
}
```

Если транслятору пока неизвестно следующее описание класса `D`:

```
class D : public A, public B {
  // ...
};
```

то возникает ошибка, т.к. правильное приведение указателя `pb` к `D*` требует изменения значения указателя. Решение с операцией `ptr_cast()` не сталкивается с этой трудностью, поскольку эта операция применима только при условии, что в области видимости находятся описания обеих ее параметров. Приведенный пример показывает, что операция приведения для неописанных классов по сути своей ненадежна, но запрещение ее существенно ухудшает совместимость с языком C.

13.6 Обширный интерфейс

Когда обсуждались абстрактные типы (§ 13.3) и узловые классы (§ 13.4), было подчеркнуто, что все функции базового класса реализуются в самом базовом или в производном классе. Но существует и другой способ построения классов. Рассмотрим, например, списки, массивы, ассоциативные массивы, деревья и т.д. Естественно желание для всех этих типов, часто называемых контейнерами, создать обобщающий их класс, который можно использовать в качестве интерфейса с любым из перечисленных типов. Очевидно, что пользователь не должен знать детали, касающиеся конкретного контейнера. Но задача определения интерфейса для обобщенного контейнера нетривиальна. Предположим, что такой контейнер будет определен как абстрактный тип, тогда какие операции он должен предоставлять? Можно предоставить только те операции, которые есть в каждом контейнере, т.е. пересечение множеств операций, но такой интерфейс будет слишком узким. На самом деле, во многих, имеющих смысл случаях такое пересечение пусто. В качестве альтернативного решения можно предоставить объединение всех множеств операций и предусмотреть динамическую ошибку, когда в этом интерфейсе к объекту применяется "несуществующая" операция. Объединение интерфейсов классов, представляющих множество понятий, называется обширным интерфейсом. Опишем "общий" контейнер объектов типа T:

```
class container {
public:
  struct Bad_operation { // класс особых ситуаций
     const char* p;
     Bad_operation(const char* pp) : p(pp) { }
  };

  virtual void put(const T*)
    { throw Bad_operation("container::put"); }
  virtual T* get()
    { throw Bad_operation("container::get"); }

  virtual T*& operator[](int)
    { throw Bad_operation("container::[](int)"); }
  virtual T*& operator[](const char*)
    { throw Bad_operation("container::[](char*)"); }
  // ...
};
```

Все-таки существует мало реализаций, где удачно представлены как индексирование, так и операции типа списочных, и, возможно, не стоит совмещать их в одном классе.

Отметим такое различие: для гарантии проверки на этапе трансляции в абстрактном типе используются чистые виртуальные функции, а для обнаружения ошибок на этапе выполнения используются функции обширного интерфейса, запускающие особые ситуации.

Можно следующим образом описать контейнер, реализованный как простой список с односторонней связью:

```
class slist_container : public container, private slist {
public:
  void put(const T*);
  T* get();

  T*& operator[](int)
     { throw Bad_operation("slist::[](int)"); }
  T*& operator[](const* char)
```

```
    { throw Bad_operation("slist::[](char*)"); }
  // ...
};
```

Чтобы упростить обработку динамических ошибок для списка введены операции индексирования. Можно было не вводить эти нереализованные для списка операции и ограничиться менее полной информацией, которую предоставляют особые ситуации, запущенные в классе `container`:

```
class vector_container : public container, private vector {
public:
  T*& operator[](int);
  T*& operator[](const char*);
  // ...
};
```

Если быть осторожным, то все работает нормально:

```
void f()
{
  slist_container sc;
  vector_container vc;
  // ...
}

void user(container& c1, container& c2)
{
  T* p1 = c1.get();
  T* p2 = c2[3];
  // нельзя использовать c2.get() или c1[3]
  // ...
}
```

Все же для избежания ошибок при выполнении программы часто приходится использовать динамическую информацию о типе (§ 13.5) или особые ситуации (§ 9). Приведем пример:

```
void user2(container& c1, container& c2)
```

```
/*
   обнаружение ошибки просто, восстановление - трудная задача
*/
{
  try {
     T* p1 = c1.get();
     T* p2 = c2[3];
     // ...
   }
   catch(container::Bad_operation& bad) {
     // Приехали!
     // А что теперь делать?
   }
}
```

или другой пример:

```
void user3(container& c1, container& c2)
/*
  обнаружение ошибки непросто,
  а восстановление по прежнему трудная задача
*/
{
  slist* sl = ptr_cast(slist_container,&c1);
  vector* v = ptr_cast(vector_container, &c2);

  if (sl && v) {
     T* p1 = c1.get();
     T* p2 = c2[3];
     // ...
  }
  else {
     // Приехали!
     // А что теперь делать?
  }
}
```

Оба способа обнаружения ошибки, показанные на этих примерах, приводят к программе с "раздутым" кодом и низкой скоростью

выполнения. Поэтому обычно просто игнорируют возможные ошибки в надежде, что пользователь на них не натолкнется. Но задача от этого не упрощается, ведь полное тестирование затруднительно и требует многих усилий.

Поэтому, если целью является программа с хорошими характеристиками, или требуются высокие гарантии корректности программы, или, вообще, есть хорошая альтернатива, лучше не использовать обширные интерфейсы. Кроме того, использование обширного интерфейса нарушает взаимнооднозначное соответствие между классами и понятиями, и тогда начинают вводить новые производные классы просто для удобства реализации.

13.7 Каркас области приложения

Мы перечислили виды классов, из которых можно создать библиотеки, нацеленные на проектирование и повторное использование прикладных программ. Они предоставляют определенные "строительные блоки" и объясняют как из них строить. Разработчик прикладного обеспечения создает каркас, в который должны вписаться универсальные строительные блоки. Задача проектирования прикладных программ может иметь иное, более обязывающее решение: написать программу, которая сама будет создавать общий каркас области приложения. Разработчик прикладного обеспечения в качестве строительных блоков будет встраивать в этот каркас прикладные программы. Классы, которые образуют каркас области приложения, имеют настолько обширный интерфейс, что их трудно назвать типами в обычном смысле слова. Они приближаются к тому пределу, когда становятся чисто прикладными классами, но при этом в них фактически есть только описания, а все действия задаются функциями, написанными прикладными программистами.

Для примера рассмотрим фильтр, т.е. программу, которая может выполнять следующие действия: читать входной поток, производить над ним некоторые операции, выдавать выходной поток и определять конечный результат. Примитивный каркас для фильтра будет состоять из определения множества операций, которые должен реализовать прикладной программист:

```
class filter {
public:
  class Retry {
  public:
    virtual const char* message() { return 0; }
  };

virtual void start() { }
virtual int retry() { return 2; }
virtual int read() = 0;
virtual void write() { }
virtual void compute() { }
virtual int result() = 0;
        };
```

Нужные для производных классов функции описаны как чистые виртуальные, остальные функции просто пустые. Каркас содержит основной цикл обработки и зачаточные средства обработки ошибок:

```
int main_loop(filter* p)
{
  for (;;) {
     try {
        p->start();
        while (p->read()) {
    p->compute();
    p->write();
        }
        return p->result();
     }
     catch (filter::Retry& m) {
       cout << m.message() << '\n';
       int i = p->retry();
       if (i) return i;
     }
     catch (...) {
       cout << 'Fatal filter error\n';
       return 1;
     }
```

```
    }
  }
```

Теперь прикладную программу можно написать так:

```
class myfilter : public filter {
  istream& is;
  ostream& os;
  char c;
  int nchar;

public:
  int read() { is.get(c); return is.good(); }
  void compute() { nchar++; };
  int result()
      { os << nchar
<< "characters read\n";
return 0;
      }

  myfilter(istream& ii, ostream& oo)
    : is(ii), os(oo), nchar(0) { }
};
```

и вызывать ее следующим образом:

```
int main()
{
 myfilter f(cin,cout);
 return main_loop(&f);
}
```

Настоящий каркас, чтобы рассчитывать на применение в реальных задачах, должен создавать более развитые структуры и предоставлять больше полезных функций, чем в нашем простом примере. Как правило, каркас образует дерево узловых классов. Прикладной программист поставляет только классы, служащие листьями в этом многоуровневом дереве, благодаря чему достигается общность между различными прикладными программами и упрощается повторное использование полезных функций, предоставляемых каркасом. Созданию каркаса могут

способствовать библиотеки, в которых определяются некоторые полезные классы, например, такие как `scrollbar` (§ 12.2.5) и `dialog_box` (§ 13.4). После определения своих прикладных классов программист может использовать эти классы.

13.8 Интерфейсные классы

Про один из самых важных видов классов обычно забывают - это "скромные" интерфейсные классы. Такой класс не выполняет какой-то большой работы, ведь иначе, его не называли бы интерфейсным. Задача интерфейсного класса приспособить некоторую полезную функцию к определенному контексту. Достоинство интерфейсных классов в том, что они позволяют совместно использовать полезную функцию, не загоняя ее в жесткие рамки. Действительно, невозможно рассчитывать, что функция сможет сама по себе одинаково хорошо удовлетворить самые разные запросы.

Интерфейсный класс в чистом виде даже не требует генерации кода. Вспомним описание шаблона типа `Splist` из § 8.3.2:

```
template<class T>
class Splist : private Slist<void*> {
public:
  void insert(T* p) { Slist<void*>::insert(p); }
  void append(T* p) { Slist<void*>::append(p); }
  T* get() { return (T*) Slist<void*>::get(); }
};
```

Класс `Splist` преобразует список ненадежных обобщенных указателей типа `void*` в более удобное семейство надежных классов, представляющих списки. Чтобы применение интерфейсных классов не было слишком накладно, нужно использовать функции-подстановки. В примерах, подобных приведенному, где задача функций-подстановок только подогнать тип, накладные расходы в памяти и скорости выполнения программы не возникают.

Естественно, можно считать интерфейсным абстрактный базовый класс, который представляет абстрактный тип, реализуемый

конкретными типами (§ 13.3), также как и управляющие классы из раздела 13.9. Но здесь мы рассматриваем классы, у которых нет иных назначений - только задача адаптации интерфейса.

Рассмотрим задачу слияния двух иерархий классов с помощью множественного наследования. Как быть в случае коллизии имен, т.е. ситуации, когда в двух классах используются виртуальные функции с одним именем, производящие совершенно разные операции? Пусть есть видеоигра под названием "Дикий запад", в которой диалог с пользователем организуется с помощью окна общего вида (класс Window):

```
class Window {
  // ...
  virtual void draw();
};

class Cowboy {
  // ...
  virtual void draw();
      };

      class CowboyWindow : public Cowboy, public Window {
// ...
      };
```

В этой игре класс `CowboyWindow` представляет движение ковбоя на экране и управляет взаимодействием игрока с ковбоем. Очевидно, появится много полезных функций, определенных в классе `Window` и `Cowboy`, поэтому предпочтительнее использовать множественное наследование, чем описывать `Window` или `Cowboy` как члены. Хотелось бы передавать этим функциям в качестве параметра объект типа `CowboyWindow`, не требуя от программиста указания каких-то спецификаций объекта. Здесь как раз и возникает вопрос, какую функции выбрать для `CowboyWindow`: `Cowboy::draw()` или `Window::draw()`.

В классе `CowboyWindow` может быть только одна функция с именем `draw()`, но поскольку полезная функция работает с объектами `Cowboy`

или Window и ничего не знает о CowboyWindow, в классе CowboyWindow должны подавляться (переопределяться) и функция Cowboy::draw(), и функция Window_draw(). Подавлять обе функции с помощью одной - draw() неправильно, поскольку, хотя используется одно имя, все же все функции draw() различны и не могут переопределяться одной.

Наконец, желательно, чтобы в классе CowboyWindow наследуемые функции Cowboy::draw() и Window::draw() имели различные однозначно заданные имена.

Для решения этой задачи нужно ввести дополнительные классы для Cowboy и Window. Вводится два новых имени для функций draw() и гарантируется, что их вызов в классах Cowboy и Window приведет к вызову функций с новыми именами:

```
class CCowboy : public Cowboy {
  virtual int cow_draw(int) = 0;
  void draw() { cow_draw(i); } // переопределение Cowboy::draw
};

class WWindow : public Window {
  virtual int win_draw() = 0;
  void draw() { win_draw(); } // переопределение Window::draw
};
```

Теперь с помощью интерфейсных классов CCowboy и WWindow можно определить класс CowboyWindow и сделать требуемые переопределения функций cow_draw() и win_draw:

```
class CowboyWindow : public CCowboy, public WWindow {
  // ...
  void cow_draw();
  void win_draw();
};
```

Отметим, что в действительности трудность возникла лишь потому, что у обеих функций draw() одинаковый тип параметров. Если бы типы параметров различались, то обычные правила разрешения

неоднозначности при перегрузке гарантировали бы, что трудностей не возникнет, несмотря на наличие различных функций с одним именем.

Для каждого случая использования интерфейсного класса можно предложить такое расширение языка, чтобы требуемая адаптация проходила более эффективно или задавалась более элегантным способом. Но такие случаи являются достаточно редкими, и нет смысла чрезмерно перегружать язык, предоставляя специальные средства для каждого отдельного случая. В частности, случай коллизии имен при слиянии иерархий классов довольно редки, особенно если сравнивать с тем, насколько часто программист создает классы. Такие случаи могут возникать при слиянии иерархий классов из разных областей (как в нашем примере: игры и операционные системы). Слияние таких разнородных структур классов всегда непростая задача, и разрешение коллизии имен является в ней далеко не самой трудной частью. Здесь возникают проблемы из-за разных стратегий обработки ошибок, инициализации, управления памятью. Пример, связанный с коллизией имен, был приведен потому, что предложенное решение: введение интерфейсных классов с функциями-переходниками, - имеет много других применений. Например, с их помощью можно менять не только имена, но и типы параметров и возвращаемых значений, вставлять определенные динамические проверки и т.д.

Функции-переходники `CCowboy::draw()` и `WWindow_draw` являются виртуальными, и простая оптимизация с помощью подстановки невозможна. Однако, есть возможность, что транслятор распознает такие функции и удалит их из цепочки вызовов.

Интерфейсные функции служат для приспособления интерфейса к запросам пользователя. Благодаря им в интерфейсе собираются операции, разбросанные по всей программе. Обратимся к классу `vector` из § 1.4.

Для таких векторов, как и для массивов, индекс отсчитывается от нуля. Если пользователь хочет работать с диапазоном индексов, отличным от диапазона `0..size-1`, нужно сделать соответствующие приспособления, например, такие:

```
void f()
```

```
{
  vector v(10);  // диапазон [0:9]

  // как будто v в диапазоне [1:10]:

  for (int i = 1; i<=10; i++) {
    v[i-1] = ... // не забыть пересчитать индекс

  }
  // ...
}
```

Лучшее решение дает класс vec с произвольными границами индекса:

```
class vec : public vector {
  int lb;
public:
  vec(int low, int high)
    : vector(high-low+1) { lb=low; }

  int& operator[](int i)
    { return vector::operator[](i-lb); }

  int low() { return lb; }
  int high() { return lb+size() - 1; }
};
```

Класс vec можно использовать без дополнительных операций, необходимых в первом примере:

```
void g()
{
  vec v(1,10);  // диапазон [1:10]

  for (int i = 1; i<=10; i++) {
    v[i] = ...

  }
  // ...
```

```
}
```

Очевидно, вариант с классом vec нагляднее и безопаснее.

Интерфейсные классы имеют и другие важные области применения, например, интерфейс между программами на C++ и программами на другом языке (§ 12.1.4) или интерфейс с особыми библиотеками C++.

13.9 Управляющие классы

Концепция абстрактного класса дает эффективное средство для разделения интерфейса и его реализации. Мы применяли эту концепцию и получали постоянную связь между интерфейсом, заданным абстрактным типом, и реализацией, представленной конкретным типом. Так, невозможно переключить абстрактный итератор с одного класса-источника на другой, например, если исчерпано множество (класс `set`), невозможно перейти на потоки.

Далее, пока мы работаем с объектами абстрактного типа с помощью указателей или ссылок, теряются все преимущества виртуальных функций. Программа пользователя начинает зависеть от конкретных классов реализации. Действительно, не зная размера объекта, даже при абстрактном типе нельзя разместить объект в стеке, передать как параметр по значению или разместить как статический. Если работа с объектами организована через указатели или ссылки, то задача распределения памяти перекладывается на пользователя (§ 13.10).

Существует и другое ограничение, связанное с использованием абстрактных типов. Объект такого класса всегда имеет определенный размер, но классы, отражающие реальное понятие, могут требовать память разных размеров.

Есть распространенный прием преодоления этих трудностей, а именно, разбить отдельный объект на две части: управляющую, которая определяет интерфейс объекта, и содержательную, в которой находятся все или большая часть атрибутов объекта. Связь между двумя частями реализуется с помощью указателя в управляющей части на содержательную часть. Обычно в управляющей части кроме указателя

есть и другие данные, но их немного. Суть в том, что состав управляющей части не меняется при изменении содержательной части, и она настолько мала, что можно свободно работать с самими объектами, а не с указателями или ссылками на них.

управляющая часть содержательная часть

Простым примером управляющего класса может служить класс string из § 7.6. В нем содержится интерфейс, контроль доступа и управление памятью для содержательной части. В этом примере управляющая и содержательная части представлены конкретными типами, но чаще содержательная часть представляется абстрактным классом.

Теперь вернемся к абстрактному типу `set` из § 13.3. Как можно определить управляющий класс для этого типа, и какие это даст плюсы и минусы? Для данного класса set можно определить управляющий класс просто перегрузкой операции ->:

```
class set_handle {
  set* rep;
public:
  set* operator->() { return rep; }

  set_handler(set* pp) : rep(pp) { }
};
```

Это не слишком влияет на работу с множествами, просто передаются объекты типа `set_handle` вместо объектов типа `set&` или `set*`, например:

```
void my(set_handle s)
{
  for (T* p = s->first(); p; p = s->next())
  {
    // ...
  }
  // ...
}
```

```
void your(set_handle s)
{
  for (T* p = s->first(); p; p = s->next())
  {
    // ...
  }
  // ...
}

void user()
{
  set_handle sl(new slist_set);
  set_handle v(new vector_set v(100));

  my(sl);
  your(v);

  my(v);
  your(sl);
}
```

Если классы `set` и `set_handle` разрабатывались совместно,легко реализовать подсчет числа создаваемых множеств:

```
class set {
friend class set_handle;
protected:
  int handle_count;
public:
  virtual void insert(T*) = 0;
  virtual void remove(T*) = 0;

  virtual int is_member(T*) = 0;

  virtual T* first() = 0;
  virtual T* next() = 0;

  set() : handle_count(0) { }
};
```

Чтобы подсчитать число объектов данного типа `set`, в управляющем классе нужно увеличивать или уменьшать значение счетчика `set_handle`:

```
class set_handle {
  set* rep;
public:
  set* operator->() { return rep; }

set_handle(set* pp)
  : rep(pp) { pp->handle_count++; }
set_handle(const set_handle& r)
  : rep(r.rep) { rep->handle_count++; }

set_handle& operator=(const set_handle& r)
{
  rep->handle_count++;
  if (--rep->handle_count == 0) delete rep;
  rep = r.rep;
  return *this;
 }

 ~set_handle()
   { if (--rep->handle_count == 0) delete rep; }
        };
```

Если все обращения к классу `set` обязательно идут через `set_handle`, пользователь может не беспокоиться о распределении памяти под объекты типа `set`.

На практике иногда приходится извлекать указатель на содержательную часть из управляющего класса и пользоваться непосредственно им. Можно, например, передать такой указатель функции, которая ничего не знает об управляющем классе. Если функция не уничтожает объект, на который она получила указатель, и если она не сохраняет указатель для дальнейшего использования после возврата, никаких ошибок быть не должно. Может оказаться полезным переключение управляющего класса на другую содержательную часть:

```
class set_handle {
  set* rep;
public:
  // ...

set* get_rep() { return rep; }

void bind(set* pp)
{
  pp->handle_count++;
  if (--rep->handle_count == 0) delete rep;
  rep = pp;
}
        };
```

Создание новых производных от `set_handle` классов обычно не имеет особого смысла, поскольку это - конкретный тип без виртуальных функций. Другое дело - построить управляющий класс для семейства классов, определяемых одним базовым. Полезным приемом будет создание производных от такого управляющего класса. Этот прием можно применять как для узловых классов, так и для абстрактных типов.

Естественно задавать управляющий класс как шаблон типа:

```
template<class T> class handle {
  T* rep;
public:
  T* operator->() { return rep; }
  // ...
};
```

Но при таком подходе требуется взаимодействие между управляющим и "управляемым" классами. Если управляющий и управляемые классы разрабатываются совместно, например, в процессе создания библиотеки, то это может быть допустимо. Однако, существуют и другие решения (§ 13.10).

За счет перегрузки операции `->` управляющий класс получает

возможность контроля и выполнения каких-то операций при каждом обращении к объекту. Например, можно вести подсчет частоты использования объектов через управляющий класс:

```
template<class T>
  class Xhandle {
    T* rep;
    int count;
  public:
    T* operator->() { count++; return rep; }

    // ...
  };
```

Нужна более сложная техника, если требуется выполнять операции как перед, так и после обращения к объекту. Например, может потребоваться множество с блокировкой при выполнении операций добавления к множеству и удаления из него. Здесь, по сути, в управляющем классе приходится дублировать интерфейс с объектами содержательной части:

```
class set_controller {
  set* rep;
  // ...
public:

  lock();
  unlock();

  virtual void insert(T* p)
    { lock(); rep->insert(p); unlock(); }
  virtual void remove(T* p)
    { lock(); rep->remove(p); unlock(); }

  virtual int is_member(T* p)
    { return rep->is_member(p); }

  virtual T* first() { return rep->first(); }
  virtual T* next() { return rep->next(); }
```

```
    // ...
};
```

Писать функции-переходники для всего интерфейса утомительно (а значит могут появляться ошибки), но не трудно и это не ухудшает характеристик программы.

Заметим, что не все функции из `set` следует блокировать. Как показывает опыт автора, типичный случай, когда операции до и после обращения к объекту надо выполнять не для всех, а только для некоторых функций-членов. Блокировка всех операций, как это делается в мониторах некоторых операционных систем, является избыточной и может существенно ухудшить параллельный режим выполнения.

Переопределив все функции интерфейса в управляющем классе, мы получили по сравнению с приемом перегрузки операции `->`, то преимущество, что теперь можно строить производные от `set_controller` классы. К сожалению, мы можем потерять и некоторые достоинства управляющего класса, если к производным классам будут добавляться члены, представляющие данные. Можно сказать, что программный объем, который разделяется между управляемыми классами уменьшается по мере роста программного объема управляющего класса.

13.10 Управление памятью

При проектировании библиотеки или просто программы с большим временем счета один из ключевых вопросов связан с управлением памятью. В общем случае создатель библиотеки не знает, в каком окружении она будет работать. Будет ли там ресурс памяти настолько критичен, что ее нехватка станет серьезной проблемой, или же серьезной помехой станут накладные расходы, связанные с управлением памятью?

Один из основных вопросов управления памятью можно сформулировать так: если функция `f()` передает или возвращает указатель на объект, то кто должен уничтожать этот объект? Необходимо ответить и на связанный с ним вопрос: в какой момент объект может

быть уничтожен? Ответы на эти вопросы особенно важны для создателей и пользователей таких контейнеров, как списки, массивы и ассоциативные массивы. С точки зрения создателя библиотеки идеальными будут ответы: "Система" и "В тот момент, когда объект больше никто не использует". Когда система уничтожает объект, обычно говорят, что она занимается сборкой мусора, а та часть системы, которая определяет, что объект больше никем не используется, и уничтожает его, называется сборщиком мусора.

К сожалению, использование сборщика мусора может повлечь за собой накладные расходы на время счета и память, прерывания полезных функций, определенную аппаратную поддержку, трудности связывания частей программы на разных языках или просто усложнение системы. Многие пользователи не могут позволить себе этого.

Говорят, что программисты на Лиспе знают, насколько важно управление памятью, и поэтому не могут отдать его пользователю. Программисты на С тоже знают, насколько важно управление памятью, и поэтому не могут оставить его системе.

Поэтому в большинстве программ на C++ не приходится рассчитывать на сборщик мусора, и нужно предложить свою стратегию размещения объектов в свободной памяти, не обращаясь к системе. Но реализации C ++ со сборщиком мусора все-таки существуют.

Рассмотрим самую простую схему управления памятью для программ на C++. Для этого заменим operator `new()` на тривиальную функцию размещения, а operator `delete()` - на пустую функцию:

```
inline size_t align(size_t s)
/*
   Даже в простой функции размещения нужно
   выравнивание памяти, чтобы на объект
   можно было настроить указатель
   произвольного типа
*/
{
  union Word { void* p; long double d; long l; }
```

```
  int x = s + sizeof(Word) - 1;
  x -= x%sizeof(Word);
  return x;
}

static void* freep; // настроим start на свободную память

void* operator  new(size_t s) // простая линейная функция размещения
{
  void* p = freep;
  s = align(s);
  freep += s;
  return p;
}

void operator delete(void*) { }  // пусто
```

Если память бесконечна, то наше решение дает сборщик мусора без всяких сложностей и накладных расходов. Такой подход не применим для библиотек, когда заранее неизвестно, каким образом будет использоваться память, и когда программа, пользующаяся библиотекой, будет иметь большое время счета. Такой способ выделения памяти идеально подходит для программ, которым требуется ограниченный объем памяти или объем, пропорциональный размеру входного потока данных.

13.10.1 Сборщик мусора

Сборку мусора можно рассматривать как моделирование бесконечной памяти на памяти ограниченного размера. Помня об этом, можно ответить на типичный вопрос: должен ли сборщик мусора вызывать деструктор для тех объектов, память которых он использует? Правильный ответ - нет, поскольку, если размещенный в свободной памяти объект не был удален, то он не будет и уничтожен. Исходя из этого, операцию delete можно рассматривать как запрос на вызов деструктора (и еще это - сообщение системе, что память объекта можно использовать). Но как быть, если действительно требуется уничтожить размещенный в свободной памяти объект, который не был удален?

Заметим, что для статических и автоматических объектов такой вопрос не встает, - деструкторы для них неявно вызываются всегда. Далее, уничтожение объекта "во время сборки мусора" по сути является операцией с непредсказуемым результатом. Она может совершиться в любое время между последним использованием объекта и "концом программы", а значит, в каком состоянии будет программа в этот момент, неизвестно.

Здесь использованы кавычки, потому что трудно точно определить, что такое конец программы. (прим. перев.)

Задачу уничтожения объектов, если время этой операции точно не задано, можно решить с помощью программы обслуживания заявок на уничтожение. Назовем ее сервером заявок. Если объект необходимо уничтожить в конце программы, то надо записать в глобальный ассоциативный массив его адрес и указатель на функцию "очистки". Если объект удален явной операцией, заявка аннулируется. При уничтожении самого сервера (в конце программы) вызываются функции очистки для всех оставшихся заявок. Это решение подходит и для сборки мусора, поскольку мы рассматриваем ее как моделирование бесконечной памяти. Для сборщика мусора нужно выбрать одно из двух решений: либо удалять объект, когда единственной оставшейся ссылкой на него будет ссылка, находящаяся в массиве самого сервера, либо (стандартное решение) не удалять объект до конца программы, поскольку все-таки ссылка на него есть.

Сервер заявок можно реализовать как ассоциативный массив (§ 8.8):

```
class Register {
    Map<void*, void (*) (void*)> m;
public:
    insert(void* po, void(*pf)()) { m[po]=pf; }
    remove(void* po) { m.remove(po); }
};

Register cleanup_register;
```

Класс, постоянно обращающийся к серверу, может выглядеть так:

```
class X {
  // ...
  static void cleanup(void*);
public:


  X()
  {
    cleanup_register.insert(this,&cleanup);
    // ...
  }


  ~X() { cleanup(this); }

  // ...
};

void X::cleanup(void* pv)
{
  X* px = (X*)pv;
  cleanup_register.remove(pv);
  // очистка
}
```

Чтобы в классе `Register` не иметь дела с типами, мы использовали статическую функцию-член с указателем типа `void*`.

13.10.2 Контейнеры и удаление

Допустим, что у нас нет бесконечной памяти и сборщика мусора. На какие средства управления памятью может рассчитывать создатель контейнера, например, класса `Vector`? Для случая таких простых элементов, как `int`, очевидно, надо просто копировать их в контейнер. Столь же очевидно, что для других типов, таких, как абстрактный класс `Shape`, в контейнере следует хранить указатель. Создатель библиотеки должен предусмотреть оба варианта. Приведем набросок очевидного

решения:

```
template<class T> Vector {
  T* p;
  int sz;
public:
  Vector(int s) { p = new T[sz=s]; }
  // ...
};
```

Если пользователь не будет заносить в контейнер вместо указателей на объекты сами объекты типа `Shape`, то это решение подходит для обоих вариантов.

```
Vector<Shape*> vsp(200);  // нормально
Vector<Shape> vs(200);    // ошибка при трансляции
```

К счастью, транслятор отслеживает попытку создать массив объектов абстрактного базового класса `Shape`.

Однако, если используются указатели, создатель библиотеки и пользователь должны договориться, кто будет удалять хранимые в контейнере объекты. Рассмотрим пример:

```
void f()
    // противоречивое использование средств
    // управления памятью
{
  Vector<Shape*> v(10);
  Circle* cp = new Circle;
  v[0] = cp;
  v[1] = new Triangle;
  Square s;
  v[2] = &s;
  delete cp; // не удаляет объекты, на которые настроены
          // указатели, находящиеся в контейнере
 }
```

Если использовать реализацию класса `Vector` из § 1.4.3, объект

`Triangle` в этом примере навсегда останется в подвешенном состоянии (на него нет указателей), если только нет сборщика мусора. Главное в управлении памятью это - это корректность. Рассмотрим такой пример:

```
void g()
// корректное использование средств управления памятью
{
  Vector<Shape*> v(10);
  Circle* cp = new Circle;
  v[0] = cp;
  v[1] = new Triangle;
  Square s;
  v[2] = &s;
  delete cp;
  delete v[1];
}
```

Рассмотрим теперь такой векторный класс,который следит за удалением занесенных в него указателей:

```
template<class T> MVector {
  T* p;
  int sz;
public:
  MVector(int s);
  ~MVector();
  // ...
};

template<class T> MVector<T>::MVector(int s)
{
  // проверка s
  p = new T[sz=s];
  for (int i = 0; i<s; i++) p[i] = 0;
}

template<class T> MVector<T>::~MVector()
{
```

```
  for (int i = 0; i<s; i++) delete p[i];
  delete p;
}
```

Пользователь может рассчитывать, что содержащиеся в `MVector` указатели будут удалены. Отсюда следует, что после удаления `MVector` пользователь не должен обращаться с помощью указателей к объектам, заносившимся в этот контейнер. В момент уничтожения `MVector` в нем не должно быть указателей на статические или автоматические объекты, например:

```
void h()
// корректное использование средств управления памятью
        {
  MVector<Shape*> v(10);
  Circle* cp = new circle();
  v[0] = cp;
  v[1] = new Triangle;
  Square s;
  v[2] = &s;
  v[2] = 0;  // предотвращает удаление s

  // все оставшиеся указатели
  // автоматически удаляются при выходе
}
```

Естественно, такое решение годится только для контейнеров, в которых не содержатся копии объектов, а для класса `Map` (§ 8.8), например, оно не годится. Здесь приведен простой вариант деструктора для `MVector`, но содержится ошибка, поскольку один и тот же указатель, дважды занесенный в контейнер, будет удаляться тоже два раза.

Построение и уничтожение таких контейнеров, которые следят за удалением содержащихся в них объектах, довольно дорогостоящая операция. Копирование же этих контейнеров следует запретить или, по крайней мере, сильно ограничить (действительно, кто будет отвечать за удаление - контейнер или его копия?):

```
template<class T> MVector {
```

```
    // ...
  private:
    MVector(const MVector&); //предотвращает копирование
    MVector& operator=(const MVector&); //то же самое
    // ...
  };
```

Отсюда следует, что такие контейнеры надо передавать по ссылке или указателю (если, вообще, это следует делать), но тогда в управлении памятью возникает трудность другого рода.

Часто бывает полезно уменьшить число указателей, за которыми должен следить пользователь. Действительно, намного проще следить за 100 объектами первого уровня, которые, в свою очередь, управляют 1000 объектов нулевого уровня, чем непосредственно работать с 1100 объектами. Собственно, приведенные в этом разделе приемы, как и другие приемы, используемые для управления памятью, сводятся к стандартизации и универсализации за счет применения конструкторов и деструкторов. Это позволяет свести задачу управления памятью для практически невообразимого числа объектов, скажем 100 000, до вполне управляемого числа, скажем 100.

Можно ли таким образом определить класс контейнера, чтобы программист, создающий объект типа контейнера, мог выбрать стратегию управления памятью из нескольких возможных, хотя определен контейнер только одного типа? Если это возможно, то будет ли оправдано? На второй вопрос ответ положительный, поскольку большинство функций в системе вообще не должны заботиться о распределении памяти. Существование же нескольких разных типов для каждого контейнерного класса является для пользователя ненужным усложнением. В библиотеке должен быть или один вид контейнеров (`Vector` или `MVector`), или же оба, но представленные как варианты одного типа, например:

```
template<class T> PVector {
    T** p;
    int sz;
    int managed;
  public:
```

```
    PVector(int s, int managed = 0 );
    ~PVector();
    // ...
};

template<class T> PVector<T>::PVector(int s, int m)
{
  // проверка s
  p = new T*[sz=s];
  if (managed = m)
     for (int i = 0; i<s; i++) p[i] = 0;
}

template<class T> PVector<T>::~PVector()
{
  if (managed) {
     for (int i = 0; i<s; i++) delete p[i];
  }
  delete p;
}
```

Примером класса, который может предложить библиотека для облегчения управления памятью, является управляющий класс из § 13.9. Раз в управляющем классе ведется подсчет ссылок на него, можно спокойно передавать объект этого класса, не думая о том, кто будет удалять доступные через него объекты. Это сделает сам объект управляющего класса. Но такой подход требует, чтобы в управляемых объектах было поле для подсчета числа использований. Введя дополнительный объект, можно просто снять это жесткое требование:

```
template<class T>
class Handle {
    T* rep;
    int* pcount;
public:
    T* operator->() { return rep; }

    Handle(const T* pp)
      : rep(pp), pcount(new int) { (*pcount) = 0; }
```

```
    Handle(const Handle& r)
      : rep(r.rep), pcount(r.count) { (*pcount)++; }

    void bind(const Handle& r)
    {
      if (rep == r.rep) return;
      if (--(*pcount) == 0) { delete rep; delete pcount; }
      rep = r.rep;
      pcount = r.pcount;
      (*pcount)++;
    }

    Handle& operator=(const Handle& r)
    {
      bind(r);
      return *this;
    }

    ~Handle()
    {
      if (--(*pcount) == 0) { delete rep; delete pcount; }
    }
  };
```

13.10.3 Функции размещения и освобождения

Во всех приведенных примерах память рассматривалась как нечто данное. Однако, обычная функция общего назначения для распределения свободной памяти оказывается до удивления менее эффективной, чем функция размещения специального назначения. Вырожденным случаем таких функций можно считать приведенный пример с размещением в "бесконечной" памяти и с пустой функцией освобождения. В библиотеке могут быть более содержательные функции размещения, и бывает, что с их помощью удается удвоить скорость выполнения программы. Но прежде, чем пытаться с их помощью оптимизировать программу, запустите для нее профилировщик, чтобы выявить накладные расходы, связанные с выделением памяти.

В разделах § 5.5.6 и § 6.7 было показано как с помощью определения функций `X::operator new()` и `X::operator delete()` можно использовать функцию размещения для объектов класса `X`. Здесь есть определенная трудность. Для двух классов `X` и `Y` функции размещения могут быть настолько сходными, что желательно иметь одну такую функцию. Иными словами, желательно иметь в библиотеке такой класс, который предоставляет функции размещения и освобождения, пригодные для размещения объектов данного класса. Если такой класс есть, то функции размещения и освобождения для данного класса получаются за счет привязки к нему общих функций размещения и освобождения:

```
class X {
  static Pool my_pool;
  // ...
public:
  // ...
  void* operator new(size_t) { return my_pool.alloc(); }
  void operator delete(void* p) { my_pool.free(p); }
};

Pool X::my_pool(sizeof(X));
```

С помощью класса `Pool` память распределяется блоками одного размера. В приведенном примере объект `my_pool` отводит память блоками размером `sizeof(X)`.

Составляется описание класса `X` и используется `Pool` с учетом оптимизации скорости программы и компактности представления. Обратите внимание, что размер выделяемых блоков памяти является для класса "встроенным", поэтому задающий размер параметр функции `X::operator new()` не используется. Используется вариант функции `X::operator delete()` без параметра. Если класс `Y` является производным класса `X`, и `sizeof(Y)>sizeof(X)`, то для класса `Y` должны быть свои функции размещения и освобождения. Наследование функций класса `X` приведет к катастрофе. К счастью, задать такие функции для `Y` очень просто.

Класс `Pool` предоставляет связанный список элементов требуемого размера. Элементы выделяются из блока памяти фиксированного размера и по мере надобности запрашиваются новые блоки памяти. Элементы группируются большими блоками, чтобы минимизировать число обращений за памятью к функции размещения общего назначения. До тех пор пока не будет уничтожен сам объект `PooL`, память никогда не возвращается функции размещения общего назначения.

Приведем описание класса Pool:

```
class Pool {

  struct Link { Link* next; }

  const unsigned esize;
  Link* head;

  Pool(Pool&);            // защита от копирования
  void operator= (Pool&); // защита от копирования
  void grow();
public:
  Pool(unsigned n);
  ~Pool();

  void* alloc();
  void free(void* b);
};

inline void* Pool::alloc()
{
  if (head==0) grow();
  Link* p = head;
  head = p->next;
  return p;
}

inline void Pool::free(void* b)
{
```

```
    Link* p = (Link*) b;
    p->next = head;
    head = p;
  }
```

Приведенные описания логично поместить в заголовочный файл Pool.h. Следующие определения могут находиться в любом месте программе и завершают наш пример. Объект `Pool` должен инициализироваться конструктором:

```
Pool::Pool(unsigned sz)
  : esize(sz)
{
  head = 0;
}
```

Функция `Pool::grow()` будет связывать все элементы в список квантов свободной памяти `head`, образуя из них новый блок. Определения остальных функций-членов оставлены в качестве упражнений 5 и 6 в § 13.11.

```
void Pool::grow()
{
  const int overhead = 12;
  const int chunk_size = 8*1024-overhead;
  const int nelem = (chunk_size-esize)/esize;

  char* start = new char[chunk_size];
  char* last = &start[(nelem-1)*esize];

  for (char* p = start; p<last; p+=esize)
    ((Link*)p)->next = ((Link*)p)+1;
  ((Link*)last)->next = 0;
  head = (Link*)start;
}
```

13.11 Упражнения

1. (*3) Завершите определения функций-членов класса `Type_info`.
2. (*3) Предложите такую структуру объекта `Type_info`, чтобы функция `Type_info::get_info()` стала лишней, и перепишите с учетом этого функции-члены `Type_info`.
3. (*2.5) Насколько наглядно вы сможете записать примеры с `Dialog_box`, не используя макроопределения (а также расширения языка)? Насколько наглядно вам удастся записать их, используя расширения языка?
4. (*4) Исследуйте две широко распространенные библиотеки. Классифицируйте все библиотечные классы, разбив их на: конкретные типы, абстрактные типы, узловые классы, управляющие классы и интерфейсные классы. Используются ли абстрактные узловые классы и конкретные узловые классы? Можно ли предложить более подходящее разбиение классов этих библиотек? Используется ли обширный интерфейс? Какие имеются средства динамической информации о типе (если они есть)? Какова стратегия управления памятью?
5. (*3) Определите шаблонный вариант класса `Pool` из § 13.10.3. Пусть размер выделяемого элемента памяти будет параметром шаблона типа, а не конструктора.
6. (*2.5) Усовершенствуйте шаблон типа `Pool` из предыдущего упражнения так, чтобы некоторые элементы размещались во время работы конструктора. Сформулируйте в чем будет проблема переносимости, если использовать Pool с типом элементов `char`, покажите как ее устранить.
7. (*3) Если ваша версия C++ прямо не поддерживает динамические запросы о типе, обратитесь к своей основной библиотеке. Реализован ли там механизм динамических запросов о типе? Если это так, задайте операции из § 13.5 как надстройку над этим механизмом.
8. (*2.5) Определите такой строковый класс, в котором нет никакого динамического контроля, и второй производный от него строковый класс, который только проводит динамический контроль и обращается к первому. Укажите плюсы и минусы такого решения по сравнению с решением,в котором делается выборочный динамический контроль, сравните с подходом, использующим инварианты, как было предложено в § 12.2.7.1. Насколько можно совмещать эти подходы?

9. (*4) Определите класс `Storable` как абстрактный базовый класс с виртуальными функциями `writeout()` и `readin()`. Для простоты допустим, что для задания нужного адресного пространства достаточно строки символов. С помощью класса `Storable` реализуйте обмен объектами с диском. Проверьте его на объектах нескольких классов по своему усмотрению.
10. (*4) Определите базовый класс `Persistent` с операциями `save()` и `nosave()`, который будет проверять, что деструктор создал объект в определенной памяти. Какие еще полезные операции можно предложить? Проверьте класс `Persistent` на нескольких классах по своему выбору. Является ли класс `Persistent` узловым классом, конкретным или абстрактным типом? Аргументируйте ответ.
11. (*3) Составьте только описание класса `stack`, который реализует стек с помощью операций `create()` (создать стек), `delete()` (уничтожить стек), `push()` (записать в стек) и `pop()` (читать из стека). Используйте только статические члены. Для привязки и обозначения стеков определите класс `id`. Гарантируйте, что пользователь сможет копировать объекты `stack::id`, но не сможет работать с ними иным способом. Сравните это определение стека с классом `stack` из § 8.2.
12. (*3) Составьте описание класса `stack`, который является абстрактным типом (§ 13.3). Предложите две различные реализации для интерфейса, заданного `stack`. Напишите небольшую программу, работающую с этими классами. Сравните это решение с классами, определяющими стек, из предыдущего упражнения и из § 8.2.
13. (*3) Составьте такое описание класса `stack`, для которого можно в динамике менять реализацию. Подсказка: "Всякую задачу можно решить, введя еще одну косвенность".
14. (*3.5) Определите класс `Oper`, содержащий идентификатор (некоторого подходящего типа) и операцию (некоторый указатель на функцию). Определите класс `cat_object`, содержащий список объектов `Oper` и объект типа `void*`. Задайте в классе `cat_object` операции: `add_oper()`, которая добавляет объект к списку; `remove_oper(id)`, которая удаляет из списка объект `Oper` с идентификатором `id` ; `operator()`

`(id,arg)`, которая вызывает функцию из объекта `Oper` с идентификатором `id`. Реализуйте с помощью класса `cat_object` стек объектов `Oper`. Напишите небольшую программу, работающую с этими классами.

15. (*3) Определите шаблон типа `Object`, служащий базовым классом для `cat_object`. С помощью `Object` реализуйте стек для объектов класса `String`. Напишите небольшую программу, использующую этот шаблон типа.
16. (*3) Определите вариант класса `Object` под именем `Class`, в котором объекты с одинаковым идентификатором имеют общий список операций. Напишите небольшую программу, использующую этот шаблон типа.
17. (*3) Определите шаблон типа `Stack`, который задает традиционный и надежный интерфейс со стеком, реализуемым объектом шаблона типа `Object`. Сравните это определение стека с классами, задающими стек, из предыдущих упражнений. Напишите небольшую программу, использующую этот шаблон типа.

Список литературы

1. A.V.Aho, J.E.Hopcroft, and J.D.Ulman, Data Structures and Algoritms, Addison-Wesley, Reading, Massachusetts. 1983

2. O-J.Dahl, B.Myrhaug, and K.Nugaard, SIMULA Common Base Language, Norwegian Computing Ctnter S-22. Oslo, Norway. 1970

3. O-J.Dahl and C.A.R.Hoare, Hierarhical Program Construction in Structured Programming, Academic Press, New York. 1972. pp. 174-220

4. Margaret A.Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, Reading, Massachusetts. 1990

5. A.Goldberg and D.Rodson, SMALLTALK-80 - The Language and Its Implementation, Addison-Wesley, Reading, Massachusetts. 1983

6. R.E.Griswold et.al, The Snobol14 Programming Language, Prentice-Hall, Englewood Cliffs, New Jersy, 1970

7. R.E.Griswold and M.T.Griswold, The ICON Programming Language, Prentice-Hall, Englewood Cliffs, New Jersy. 1983

8. Brian W.Kernighan and Dennis M.Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersy. 197108. Second edition 1988

9. Andrew Koenig and Bjarne Stroustrup, C++: As Close to C as possible - but no closer, The C++ Report. Vol.1 No.7. July 1989

10. Andrew Koenig and Bjarne Stroustrup, Exception Handling for C++ (revised), Proc USENIX C++ Conference, April 1990. Also, Journal of Object Oriented Programming, Vol.3 No.2, July/August 1990. pp.16-33

11. Barbara Liskov et.al, CLU Reference Manual, MIT/LCS/TR-225

12. George Orwell, 1984. Secker and Warburg, London. 1949

13. Martin Richards and Colin Whitby-Strevens, BCPL - The Language and Its Compiler, Cambridge University Press. 1980

14. L.Rosle, The Evolution of C - Past and Future, AT&T Bell Laboratories Technical Journal. Vol.63 No.8 Part 2. October 1984. pp.1685-1700

15. Ravi Sethi, Uniform Syntax for Type Expressions and Declarations, Software Practice & Experience, Vol.11. 1981. pp.623-628

16. Bjarne Stroustrup, Adding Classes to C: An Exercise in Language Evolution, Software Practice & Experience, Vol.13. 1983. pp.139-61

17. Bjarne Stroustrup, The C++ Programming Language, Addison-Wesley. 1986

18. Bjarne Stroustrup, Multiple Inheritance for C++, Proc. EUUG Spring Conference, May 1987. Also USENIX Computer Systems, Vol.2 No 4, Fall 1989

19. Bjarne Stroustrup and Jonathan Shopiro, A Set of C classes for Co-Routine Style Programming, Proc. USENIX C++ conference, Santa Fe. November 1987. pp.417-439

20. Bjarne Stroustrup, Type-safe Linkage for C++, USENIX Computer Systems, Vol.1 No.4 Fall 1988

21. Bjurne Stroustrup, Parameterized Type for C++, Proc. USENIX C++ Conference, Denver, October 1988. pp.1-18. Also, USENIX Computer Systems, Vol.2 No.1 Winter 1989

22. Bjarne Stroustrup, Standardizing C++, The C++ Report. Vol.1 No.1. January 1989

23. Bjarne Stroustrup, The Evolution of C++: 1985-1989, USENIX Computer Systems, Vol.2 No.3. Summer 1989

24. P.M.Woodward and S.G.Bond, Algol 68-R Users Guide, Her Majesty's Stationery Office,

London. 1974

25. UNIX Time-Sharing System: Programmer's Manual. Research Version, Tenth Edition, AT&T Bell Laboratories, Murray Hill, New Jersy, February 1985

26. Aake Wilkstroem, Functional Programming Using ML, Prentice-Hall, Englewood Cliffs, New Jersy. 1987

27. X3 Secretariat: Standard - The C Language. X3J11/90-013, Computer and Business Equipment Manufactures Association, 311 First Street, NW, Suite 500, Washington, DC 20001, USA