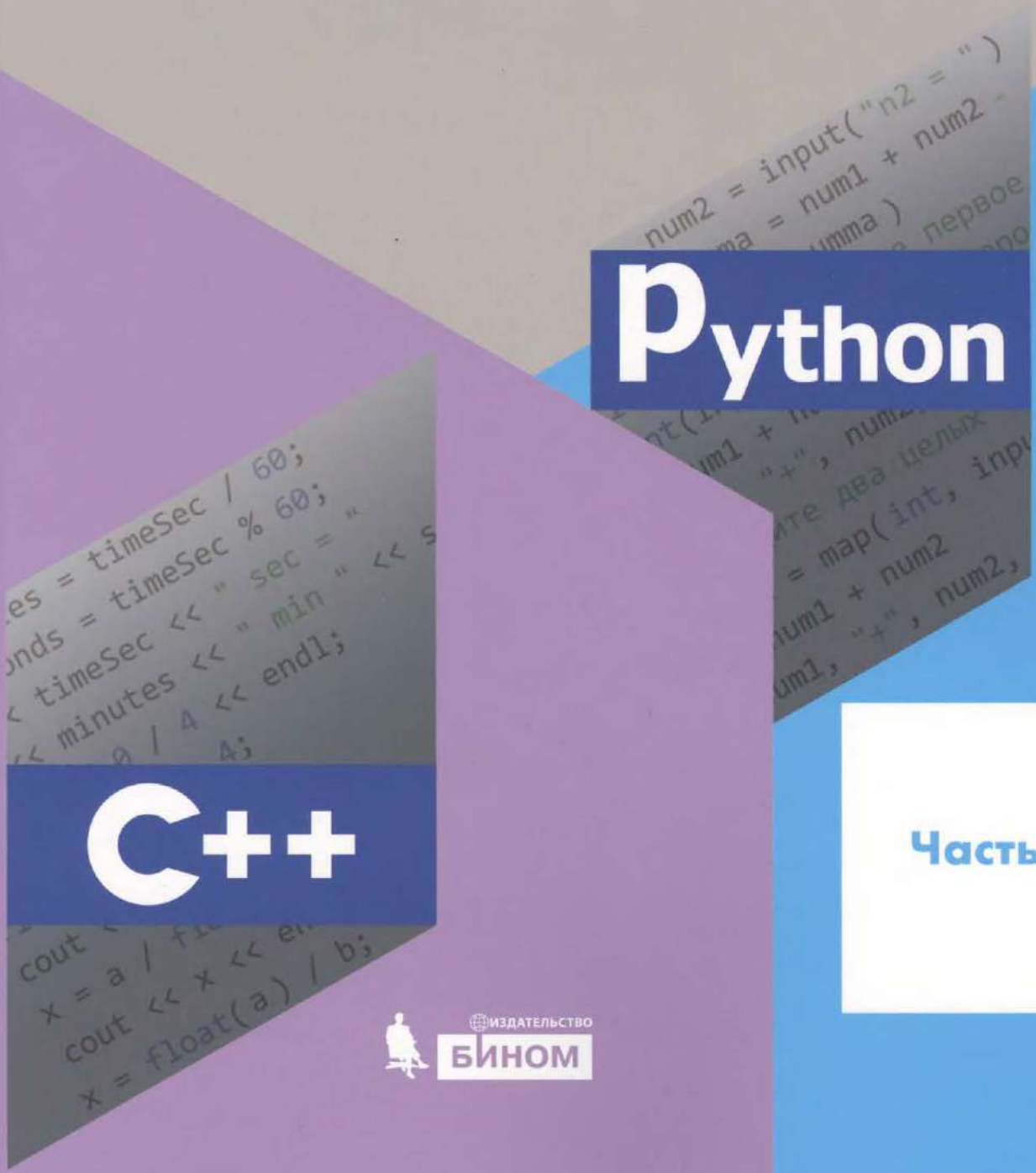
К. Ю. Поляков

ПРОГРАММИРОВАНИЕ

ПРОФИЛЬНАЯ ШКОЛА



Python

C++

Часть 1

ИЗДАТЕЛЬСТВО БИНОМ

К. Ю. Поляков

ПРОГРАММИРОВАНИЕ
Python
C++

Часть 1

Учебное пособие
для общеобразовательных
организаций

Москва
БИНОМ. Лаборатория знаний
2019

УДК 004.9
ББК 32.97
П54

Поляков К. Ю.

П54 Программирование. Python. C++. Часть 1: учебное пособие/ К. Ю. Поляков. — М. : БИНОМ. Лаборатория знаний, 2019. — 144 с. : ил.

ISBN 978-5-9963-4134-4

Книга представляет собой первую часть серии учебных пособий по программированию. В отличие от большинства аналогичных изданий, в ней представлены два языка программирования высокого уровня — Python и C++.

В пособии рассматриваются основы программирования на выбранных языках: ввод и вывод данных, обработка целых и вещественных чисел, управляющие конструкции. Объяснение нового материала строится на примерах его практического применения. Изучаются приёмы разработки программ, использующих компьютерную графику и анимацию.

После каждого параграфа приводится большое число заданий для самостоятельного выполнения разной сложности и вариантов проектных работ.

Пособие предназначено для школьников, начинающих изучать программирование.

УДК 004.9
ББК 32.97

Учебное издание

Поляков Константин Юрьевич

ПРОГРАММИРОВАНИЕ
Python. C++

Часть 1

Учебное пособие

(12+)

Ведущий редактор *О. А. Полежаева*
Концепция внешнего оформления *В. А. Андрианов*
Художественный редактор *Н. А. Новак*
Технический редактор *Е. В. Денюкова*
Корректор *Е. Н. Клитина*
Компьютерная верстка: *В. А. Носенко*

Подписано в печать 06.09.2018. Формат 84x108/16. Усл. печ. л. 15,12
Тираж 3 000 экз. Заказ № м7020.

ООО «БИНОМ. Лаборатория знаний»
127473, Москва, ул. Краснопролетарская, д. 16, стр. 3,
тел. (495)181-53-44, e-mail: binom@Lbz.ru
http://Lbz.ru, http://metodist.Lbz.ru

Отпечатано в филиале «Смоленский полиграфический комбинат»
ОАО «Издательство «Высшая школа». 214020, г. Смоленск, ул. Смольянинова, 1
Тел.: +7 (4812) 31-11-96. Факс: +7 (4812) 31-31-70
E-mail: spk@smolpk.ru http://www.smolpk.ru

ISBN 978-5-9963-4134-4

ПРЕДИСЛОВИЕ

Вы держите в руках первую часть необычного учебного пособия. В нём рассматриваются сразу два современных языка программирования — Python и C++.

С одной стороны, эти языки разные, они используются для разных целей. Python удобен для решения небольших задач, в которых скорость выполнения не очень важна. Он применяется и в серьёзных проектах: для разработки сайтов, решения задач биоинформатики и обработки больших данных. Язык C++ — основной язык для создания игр и операционных систем. Программы на C++ работают значительно быстрее, чем на Python, но их сложнее писать и отлаживать.

С другой стороны, у этих языков есть много общего: они используют одни и те же понятия и конструкции, которые оформляются немного по-разному. Поэтому перейти с одного языка на другой совсем несложно, так же как для полиглота выучить ещё один иностранный язык. Умение программировать на разных языках — это обязательное требование к разработчику программного обеспечения.

Сначала мы познакомимся с языком Python: изучим основные команды для выполнения вычислений, ветвления, циклы, научимся программировать простую компьютерную графику и анимацию.

Вторая глава пособия посвящена языку C++. Вы увидите, что знания, полученные при изучении Python, очень помогают: освоить ещё одну форму записи цикла или ветвления довольно просто, если связать новое с известным материалом.

После каждого параграфа вы найдёте множество заданий для практической работы. Некоторые из них — это проектные работы, выполнение которых может занять длительное время. Сложные задания отмечены звёздочкой, а особо сложные — двумя звёздочками.

Дополнительные материалы к пособию, в том числе файлы с программами, можно загрузить с сайта автора:

http://kpolyakov.spb.ru/school/pycpp.htm.

Автор хочет поблагодарить своих коллег за внимательное чтение рукописи этого пособия и полезные критические замечания, которые позволили устранить неточности и существенно улучшить содержание:

- И. Р. Дединского, преподавателя кафедры информатики МФТИ;
- Т. Ф. Хирьянова, преподавателя кафедры информатики МФТИ;
- Д. В. Богданова, старшего преподавателя кафедры бизнес-информатики МЭИ;
- М. Д. Полежаеву, разработчика веб-сайтов.

Глава 1
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

§ 1
Первые программы

Ключевые слова:

- программа
- транслятор
- линейная программа
- ввод данных
- символьная строка

Что такое программа?

Программа — это набор инструкций, записанных на языке, понятном компьютеру. Компьютер (точнее, его процессор) понимает только один язык — язык машинных кодов, которые записываются в виде цепочек нулей и единиц. Писать программы на таком языке (как делали программисты вычислительных машин первого поколения) очень сложно и долго. Особенно тяжело *отлаживать* их, т. е. находить и исправлять ошибки в таких программах.

Для решения этой проблемы были созданы *языки программирования высокого уровня*, в которых в командах используются слова естественного языка (чаще всего английского). Однако в отличие от естественных языков эти языки — формальные, т. е. в них каждое слово или предложение имеет один точно определённый смысл, и нет никаких исключений.

Но всё дело в том, что компьютер не понимает программы, написанные на языках высокого уровня. Чтобы подготовить программы к выполнению, используется специальная программа — *транслятор*.

Транслятор может, например, перевести программу в машинные коды конкретного процессора — такие трансляторы называются *компиляторами*.

Возможен и другой вариант: транслятор сам выполняет программу или переводит её в *байт-код* — на язык, в котором код каждой команды занимает один байт (это число в диапазоне от 0 до 255). Такой транслятор называется *интерпретатором*. Байт-код выполняется другой программой — *виртуальной машиной* (так, например, работают программы, написанные на языке Java).

Один из самых популярных современных языков программирования называется Python. Его придумал в 1991 году нидерландский программист Гвидо ван Россум. Язык Python непрерывно совершенствуется, и сейчас большинство программистов используют его третью версию — Python 3. Предыдущая версия — Python 2 — устарела, и мы не будем её рассматривать.

Несмотря на то что язык Python очень прост, он обладает огромными возможностями, которые могут применяться в самых разных областях. На Python можно программировать игры и веб-сайты, обрабатывать большие данные, решать задачи искусственного интеллекта.

Транслятор Python — это интерпретатор. Он создаёт байт-код, а затем сам его выполняет. Поэтому для того, чтобы запустить программу на Python, нужно установить на компьютер интерпретатор Python. Во многих операционных системах, например в macOS и Linux, этот интерпретатор входит в стандартную поставку и устанавливается вместе с операционной системой.

В этой главе мы начнём знакомиться с языком Python и писать на нём программы для решения различных задач.

Интерпретатор Python может работать в двух режимах:

- через командную строку (в интерактивном режиме), когда каждая введённая команда сразу выполняется;
- в программном режиме, когда программа сначала записывается в файл (обычно имеющий расширение `.py`) и при запуске выполняется целиком; такая программа на Python называется *скриптом* (от англ. *script* — сценарий).

Мы будем работать в программном режиме.

Сначала мы научимся писать программы, в которых команды выполняются последовательно, одна за другой. Как вы знаете, такие алгоритмы (и программы) называются *линейными*.

Самая простая программа

Давайте посмотрим, что представляет собой пустая программа. Это такая программа, которая не содержит никаких команд, но удовлетворяет всем требованиям языка программирования. Компьютер может выполнить её, но делать она, разумеется, ничего не будет.

Python — один из тех языков программирования, в которых пустая программа — действительно пустая, она не содержит ни одной строки. Мы можем создать пустой файл с расширением `.py`, а затем выполнить его с помощью интерпретатора.

Попробуем добавить в программу такую строку:

```
# пустая программа
```

Символ # обозначает начало комментария — пояснительного текста, который не обрабатывается транслятором. Комментарии служат для того, чтобы автору (и другому программисту) было легче разобраться в программе. При запуске такой программы также ничего не происходит. В программах на Python, в которых используются русские буквы, часто добавляют специальный комментарий:

```
# -*- coding: utf-8 -*-
```

Он говорит о том, что текст программы набран в кодировке UTF-8.

Вывод текста на экран

Теперь научим программу делать что-то полезное, например выводить текст на экран. Пусть она при запуске приветствует вас:

```
Привет!
```

Вот как выглядит такая программа:

```
print( "Привет!" )
```

Чтобы вывести что-то на экран, используется встроенная *функция* (команда) `print`. В кавычках записывается текст для вывода — *символьная строка*, т. е. последовательность символов.

В начале строки (слева от команды `print`) не должно быть пробелов — таково требование языка Python.

Вместо кавычек можно использовать апострофы («одиночные кавычки»):

```
print( 'Привет!' )
```

Это полезно, например, когда необходимо вывести строку с кавычками:

```
print( 'Смотрите фильм "Салют-7"!' )
```

За один раз можно выводить несколько символьных строк: они перечисляются через запятую внутри круглых скобок. Например, по команде

```
print( "Привет,", "Вася!" )
```

на экран выводится фраза

```
Привет, Вася!
```

Пробел между строками (элементами списка вывода) вставляется автоматически, Если он не нужен, при вызове функции нужно добавить ещё один аргумент с именем `sep` (от англ. *separator* — разделитель), равный пустой строке `""`. Команда

```
print( "2", "+", "2", "=", "4", sep="" )
```

выведет все символы без пробелов:

```
2+2=4
```

Теперь попробуем вывести второе приветствие:

```
print( "Привет, Вася!" )
print( "Привет, Петя!" )
```

Такая программа выведет каждую фразу в отдельной строке:

```
Привет, Вася!
Привет, Петя!
```

Это значит, что после вывода всех данных функция print выполняет переход на новую строку — следующий вызов print будет выводить данные в новой строке.

Если нужно, чтобы несколько вызовов функции print выводили информацию в одной строке, можно отменить переход на новую строку, указав аргумент с именем end (по-английски *end* — конец), равный пустой строке "":

```
print( "1", end="" )
print( "23", end="" )
print( "456" )
```

Такая программа выведет:

```
123456
```

Выводы

- Пустая программа на языке Python не содержит ни одной команды.
- Комментарии — это пояснения для человека внутри текста программы. Транслятор их не обрабатывает.
- Символьная строка — это последовательность символов, заключённая в кавычки или апострофы.
- Для вывода данных на экран используют функцию print.
- Элементы списка вывода разделяются запятыми.
- При выводе данные разделяются пробелами. Если это не требуется, нужно добавить именованный аргумент sep=''.
- По умолчанию после вывода всех данных функция print переводит курсор в начало следующей строки. Если это не требуется, нужно добавить именованный аргумент end=''.

Вопросы и задания

1. Зачем пишут комментарии в программах? Подумайте, как комментирование можно использовать при поиске ошибок в программе.
2. Найдите и исправьте ошибки в программе:

```
print( "Акция "Доброе дело"." )
```

3. Найдите и исправьте ошибки в программе:

```
print( "Привет,", Вася! )
```

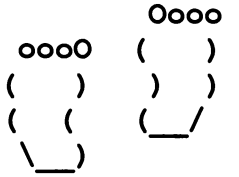
4. Заполните пропуски так, чтобы программа вывела на экран слово Python:

```
print( "Py", ... )
...
print( "on" )
```

5. Напишите программу, которая выводит на экран фразу лесенкой (автор — В. Маяковский):

```
Я волком бы
        выгрыз
              бюрократизм.
```

6. Напишите программу, которая выводит изображение:



Интересные сайты

python.org — официальный сайт поддержки языка Python

pythontutor.ru — бесплатный онлайн-курс программирования на языке Python

sourceforge.net/projects/pyscripter/ — PyScripter: свободная интегрированная среда разработки языка Python для операционной системы Windows

www.pyinstaller.org — программа PyInstaller для преобразования скриптов на языке Python в исполняемые файлы

§ 2
Диалоговые программы

Ключевые слова:

- диалоговая программа
- ввод данных
- список вывода
- переменная
- идентификатор
- оператор присваивания

В этом параграфе мы познакомимся с диалоговыми программами. Они выполняют какие-то действия в *диалоге* с пользователем, который при каждом запуске программы может вводить новые исходные данные и получать новые результаты, зависящие от того, что он ввёл. Ввод данных позволяет нам управлять программой, изменяя результаты её работы.

Как тебя зовут?

Напишем программу, которая спрашивает пользователя, как его зовут, и затем приветствует его. Вот диалог, который мы хотим получить в результате:

```
Как тебя зовут? Василина
Привет, Василина!
```

В отличие от программ, которые мы писали ранее, здесь нужно получить от пользователя данные — его имя, и потом использовать это имя в строке вывода.

Сначала выведем сообщение с подсказкой:

```
print( "Как тебя зовут? ", end="" )
```

Данные, которые мы откуда-то получаем, необходимо сохранить в памяти компьютера. Для этого используются *переменные* — области памяти, к которым можно обращаться по имени.

Для ввода данных с клавиатуры вызываем встроенную функцию (команду) `input` (от англ. *input* — ввод):

```
name = input()
```

Пара скобок говорит о том, что мы вызываем функцию. Их надо писать обязательно, даже если в скобках ничего нет.

При выполнении этой команды программа ожидает ввода пользователя, и после того, как он нажмёт клавишу Enter, введённая символьная строка запишется в переменную с именем `name`. Это значит, что в памяти выделяется область необходимого размера, с ней связывается имя `name`, и в этой области сохраняются все полученные символы.

Можно совместить вывод подсказки и ввод данных, указав текст подсказки в скобках как аргумент функции `input`:

```
name = input( "Как тебя зовут? " )
```

Знак `=` — это *оператор присваивания*. Он служит для присваивания значения переменной, т. е. для связывания имени с некоторым значением.

Оператор — это команда языка программирования.

Когда имя записано в переменную `name`, можно использовать его в программе:

```
print( "Привет, ", name, "!", sep="" )
```

В *списке вывода* три элемента: сначала выводится символьная строка `"Привет, "`, затем — значение переменной `name` (имя переменной записывается без кавычек!), и наконец — символьная строка `"!"`.

Переменные

Переменная — это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.

Имя переменной называют *идентификатором* (от слова идентифицировать — отличать один объект от другого).

В именах переменных в Python можно использовать латинские буквы (строчные и прописные буквы *различаются*), цифры и знак подчёркивания «_». Имя не может начинаться с цифры, это сделано потому, что иначе транслятору было бы сложно определить, где начинается имя, а где — число.

Желательно давать переменным «говорящие» имена, чтобы можно было сразу понять, зачем нужна та или иная переменная. Например, переменная с именем `name`, скорее всего, служит для хранения какого-то имени, а о назначении переменной `abc` догадаться очень сложно.

В отличие от многих языков программирования (Паскаль, C, Java), переменные в языке Python не нужно объявлять. Память для переменной выделяется автоматически тогда, когда переменной присваивается новое значение.

Запись значения в переменную выполняет оператор присваивания. Его можно применять не только при вводе данных с клавиатуры, но и для записи заранее известного значения:

```
name = "Поликарп"
```

Оператор присваивания позволяет изменить значение переменой:

```
name = "Платон"
name = "Сократ"
```

Переменная может хранить только одно значение. При записи в неё нового значения «старое» стирается, и его уже никак не восстановить. В языке Python при изменении значения переменной выделяется новая область памяти и связывается с тем же именем (рис. 1.1).

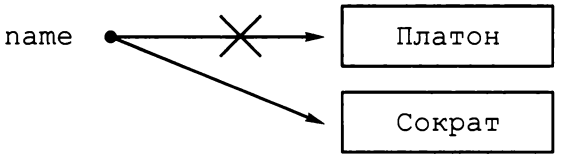


Рис. 1.1

Теперь та область памяти, в которой было записано старое значение (`"Платон"`), уже недоступна, потому что с ней не связано ни одно имя. Эта память будет освобождена *сборщиком мусора* — специальной программой, которая управляет памятью. Поэтому в языке Python невозможно, например, сразу изменить один символ внутри символьной строки (но можно создать новую, изменённую строку).

Заметим, что в большинстве языков программирования (Паскаль, C++, Java) работа с переменными организована иначе: переменные заранее объявляются, и им сразу выделяется место в памяти. После объявления вся работа с переменной происходит в одной и той же области памяти.

В языке Python каждая переменная имеет свой тип. Тип нужен для того, чтобы определить:

- какие значения может принимать переменная;
- какие операции можно выполнять с этой переменной;
- как хранить значения переменной в памяти.

Определить тип переменной можно с помощью встроенной функции `type`:

```
lang = "Котлин"
print( type(lang) )
cost = 123
print( type(cost) )
dist = 45.678
print( type(dist) )
```

Запустив эту программу, мы увидим:

```
<class 'str'>
<class 'int'>
<class 'float'>
```

Это означает, что переменные `lang`, `cost` и `dist` относятся соответственно к типам (классам):

`str` — символьная строка (от англ. *string* — строка),
`int` — целое число (от англ. *integer* — целый),
`float` — вещественное число (от англ. *float* — с плавающей запятой).

Транслятор Python сам определяет тип переменной по тому значению, которое ей присваивается.

Сумма чисел

Теперь научим компьютер складывать два целых числа. Можно, например, сложить два заранее известных числа:

```
print( 12345 + 67890 )
```

Иногда удобно сначала записать эти числа в переменные:

```
num1 = 12345
num2 = 67890
print( num1 + num2 )
```

Недостаток этой программы состоит в том, что она складывает только два заранее известных числа. Если нужно сложить другие числа, придётся менять программу. Чтобы можно было менять исходные данные, не изменяя программу, их вводят с клавиатуры, из файла, с какого-то устройства или через компьютерную сеть.

Напишем программу, которая:

1) запрашивает у пользователя два целых числа;
2) складывает их и сохраняет результат в памяти;
3) выводит результат на экран.

Вместо команд языка Python пока вставим комментарии:

```
# ввести два числа
# найти их сумму
# вывести результат
```

Компьютер не может выполнить эту программу, потому что команд «ввести два числа» и ей подобных, которые записаны в комментариях, нет в его системе команд. Будем постепенно расшифровывать комментарии — записывать вместо них команды языка Python.

Сначала введём два числа и запишем их в переменные num1 и num2:

```
num1 = input()
num2 = input()
```

Заметим, что числа нужно вводить по одному в строке, нажимая клавишу Enter после каждого введённого значения.

После этого вычислим сумму и запишем её в переменную summa:

```
summa = num1 + num2
```

Выведем результат на экран:

```
print( summa )
```

Запустив эту программу, мы увидим неожиданный результат: если ввести, например, числа 12 и 13, то мы получим не 25, а 1213. Дело в том, что функция input не знает заранее, значение какого типа нужно ввести. Поэтому она считает всё, что введено, символьной строкой. Операция сложения для символьных строк тоже определена, она дописывает вторую строку в конец первой. Таким образом, проблема в том, что программа воспринимает введённые нами данные не как числа, а как символьные строки.

Чтобы работать с числами, необходимо явно сказать, что введённые символьные строки нужно преобразовать в числа. Это делает встроенная функция int. Получается такая программа:

```
num1 = int( input() )
num2 = int( input() )
summa = num1 + num2
print( summa )
```

Обратите внимание, что в начале каждой строки не должно быть пробелов.

Недостаток этой программы — плохой диалог с пользователем:

- при вводе данных программа просто ждёт ввода, но что именно нужно вводить, неясно;
- в конце работы программа выводит какое-то число, что оно означает, неясно.

Программу можно легко доработать. Добавим в самом начале приглашение к вводу:

```
print( "Введите два целых числа:" )
```

и оформим вывод, заменив последнюю строку:

```
print( num1, "+", num2, "=", summa, sep = "" )
```

Теперь при вводе чисел 12 и 13 программа выведет:

```
12+13=25
```

Отметим, что в этой задаче можно было обойтись и без переменной summa, потому что выполнять вычисления можно прямо при выводе:

```
print( num1, "+", num2, "=", num1+num2, sep = "" )
```

Транслятор вычислит значение выражения num1+num2 и передаст его функции print для вывода. Однако если это значение понадобится позже, лучше вычислить его один раз и сохранить в переменной, а потом везде использовать значение этой переменной.

Ввод данных в одной строке

В программе, которая показана выше, мы вводили числа по одному: сначала значение переменной num1, затем, после нажатия на клавишу Enter, значение переменной num2. Иногда нужно вводить несколько значений в одной строке.

Рассмотрим случай, когда нужно ввести два целых числа в одной строке и записать их в переменные num1 и num2. В этом случае программа должна:

1) ввести символьную строку, содержащую запись двух чисел;
2) выделить части, разделённые пробелами;
3) каждую часть преобразовать в целое число.

Мы уже знаем, как решить первую задачу:

```
s = input()
```

Введённая строка записывается в переменную s. Применять к ней сразу функцию int нельзя, потому что она содержит не одно, а два числа.

Чтобы выделить две части, применим функцию split (от англ. *split* — расщепить) и, предполагая, что этих частей всего две, запишем их в переменные num1 и num2:

```
num1, num2 = s.split()
```

Здесь используется *множественное присваивание* — в одном операторе присваивания задаются значения двух переменных.

Можно обойтись и без переменной s:

```
num1, num2 = input().split()
```

Теперь нужно применить функцию int к переменным num1 и num2 — преобразовать строки в целые числа:

```
num1 = int( num1 )
num2 = int( num2 )
```

В итоге все эти операции можно заменить одной строкой:

```
num1, num2 = map(int, input().split())
```

Здесь вызывается функция `map`, которая применяет другую функцию (в нашем случае — `int`) к каждой части, полученной после разбиения введённой строки на части по пробелам.

Обратите внимание, что количество имён переменных слева от оператора присваивания должно точно соответствовать количеству введённых чисел: если их будет больше или меньше, программа завершится с ошибкой.

Выводы

- Диалоговая программа — это программа, которая обменивается данными с пользователем.
- Чтобы сохранить данные в памяти, используют переменные.
- Переменная — это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.
- Идентификатор — это имя переменной.
- Каждая переменная относится к некоторому типу данных. Тип переменной в Python определяется автоматически во время присваивания ей значения.
- Оператор — это команда языка программирования.
- При записи нового значения в переменную оно размещается в новой области памяти, а «старое» значение удаляется сборщиком мусора.
- Функция `input` вводит с клавиатуры символьную строку. Для преобразования строки в целое число её обрабатывают функцией `int`.

Вопросы и задания

1. Какие имена переменных недопустимы в языке Python?

а) `1`	и) `СУ-27`
б) `m11`	к) `СУ_27`
в) `1m`	л) `_27`
г) `m 1`	м) `СУ(27)`
д) `Vasya`	н) `@mail_ru`
е) `Петя`	о) `lenta.ru`
ж) `Митин брат`	п) `"Pes barbos"`
з) `Quo vadis`	р) `<Ладья>`

2. Какую роль играет запись `sep=""` при вызове этой функции?

```
print( "Привет, "name, "!", sep="" )
```

3. Чем различаются результаты работы программ *а* и *б*?

а)
```
print( "Как тебя зовут? " )
name = input()
```

б)
```
name = input( "Как тебя зовут? " )
```

4. Что выведет на экран эта программа?

```
name = "Мария"
print( "Привет, ", name, "!" )
print( "Привет, ", "name", "!" )
```

5. Что выведет на экран эта программа?

```
a = 1
print(a)
a = 5
print(a)
```

6. Что выведет эта программа при `a = 4`, `b = 5` и `c = 9`?

```
print( "a", "+b", "=", c )
```

7. Исправьте ошибки в операторе вывода:

```
print( "c", "-b", =, a )
```

так чтобы при `a = 4`, `b = 5` и `c = 9` программа вывела:

```
9-5=4
```

§ 3
Компьютерная графика

Ключевые слова:

- графический режим
- холст
- пиксель
- координаты
- оси координат
- модуль
- импорт модуля

Что такое компьютерная графика?

Много лет назад человек общался с компьютером только в текстовом режиме — вводил с клавиатуры символы и получал ответ компьютера тоже в форме текста.

Мониторы работали в текстовом режиме. Это значит, что весь экран был разбит на прямоугольники-символы (например, 25 строк по 80 символов в каждой) и программа могла управлять каждым символом. При этом было очень сложно что-то нарисовать на экране, потому что рисунок нужно было тоже строить из символов.

Современные мониторы работают в *графическом режиме*. Это значит, что программа может управлять отдельными точками на экране монитора, строить (по точкам) диаграммы и графики, а также выводить на экран растровые (точечные) изображения, например фотографии.

Вы уже научились создавать рисунки в графических редакторах. Теперь мы будем составлять программы, которые рисуют без нашего участия, автоматически. Раздел информатики, связанный с построением и обработкой рисунков на компьютере, называется **компьютерной графикой**.

Графика в Python

Программы, с которыми мы работали в предыдущих параграфах, выводили текст в *консольное окно*, где невозможно рисовать. Если требуется написать программу с графическими возможностями, обычно используют библиотеки, которые скрывают от программиста сложности взаимодействия с операционной системой.

Библиотека — это набор готовых функций (команд), расширяющих возможности языка программирования. Одна из библиотек, позволяющих работать с графикой, — `tkinter` — устанавливается вместе с интерпретатором языка Python.

Библиотека `graph`, с которой мы будем работать, использует возможности `tkinter`. Такая библиотека называется «обёрткой» (англ. *wrapper*) для библиотеки `tkinter`; её задача — упростить работу с `tkinter` для начинающих. С помощью библиотеки `graph` мы будем строить простые изображения и анимацию в отдельном окне.

Библиотека `graph` состоит из одного файла `graph.py`[1]. Такой файл, содержащий функции, в Python называется *модулем*. Функции, входящие в модуль, можно вызывать из своей программы.

Простейшая графическая программа состоит всего из двух строк:

```
from graph import *
run()
```

Первая строка переводится с английского как «из `graph` импортировать `*`». *Импортировать* — значит подключить к программе, знак `*` означает «все функции». Таким образом, первая строка программы подключает к нашей программе все возможности модуля `graph` — графической библиотеки.

Вторая строка — `run()` — запускает рисование и открывает графическое окно. Это вызов функции `run` из модуля `graph`. Такая команда должна всегда завершать программу, использующую модуль `graph` для работы с графикой. Для сокращения записи в тексте пособия мы не будем добавлять эту команду в конец каждой программы, помня о том, что она должна обязательно там быть.

При запуске этой программы откроется пустое окно с белым фоном. На нём можно рисовать, управляя каждым пикселем. Этим мы и займёмся.

Система координат

Поле для рисования в графических программах называется **холстом** (англ. *canvas*). Размер холста совпадает с размером графического окна. Если рисовать что-то за пределами холста, эта часть рисунка будет потеряна.

---

[1] Модуль `graph` может быть загружен с сайта автора **http://kpolyakov.spb.ru/school/probook/python.htm**.

Холст — это прямоугольник, состоящий из отдельных пикселей, т. е. *растровый рисунок*. Каждый пиксель имеет две координаты (x, y):

x — расстояние от пикселя до левой границы холста;

y — расстояние от пикселя до верхней границы холста (рис. 1.2).

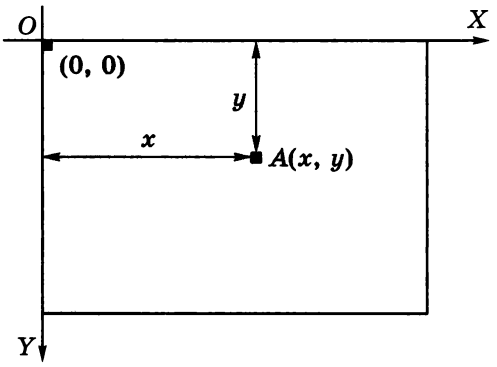


Рис. 1.2

Чаще всего используют прямоугольную систему координат, показанную на рис. 1.2. В отличие от привычной вам «математической» системы координат, здесь ось *OY* направлена не вверх, а вниз. Дело в том, что в памяти видеокарты пиксели холста хранятся построчно: слева направо, сверху вниз. Первый пиксель — это пиксель в левом верхнем углу холста, и очень удобно присвоить ему координаты (0, 0), так как его смещение от начала области холста равно 0. Теперь естественно направить ось *OY* вниз, чтобы координаты точек на холсте не были отрицательными.

Управляем пикселями

Пиксель — это наименьший элемент рисунка, которым мы можем управлять (менять его цвет) независимо от других элементов. Чтобы выкрасить пиксель, например, в синий цвет, нужно сначала установить цвет пера:

```
penColor( "blue" )
```

Название этой функции составлено из английских слов *pen* (перо) и *color* (цвет). Цвет здесь передаётся функции как символьная строка с названием цвета (`"blue"`). Так можно задать все цвета, имеющие имена в библиотеке `tkinter`, например: `white`, `black`, `gray`, `navy`, `blue`, `cyan`, `green`, `yellow`, `red`, `orange`, `brown`, `maroon`, `violet`, `purple`.

Кроме того, любой цвет можно определить кодом в модели RGB (***Red*** — красный, ***Green*** — зелёный, ***Blue*** — синий). Яркость каждой составляющей — это целое число в диапазоне от 0 до 255, например, синий цвет пера можно установить так:

```
penColor( 0, 0, 255 )
```

В коде этого цвета первые две составляющие (красная и зелёная) нулевые, а последняя (синяя) имеет максимально возможно значение 255.

Когда цвет пера выбран, рисуем точку этим пером. Для этого вызывается функция `point` (от англ. *point* — точка), ей передаются координаты пикселя (сначала x, потом — y):

```
point( 10, 20 )
```

В результате пиксель с координатами (10, 20) на холсте станет синим.

Рисуем линии

Линии на холсте состоят из пикселей. Легче всего нарисовать горизонтальную или вертикальную линию. Например, линию (точнее — отрезок) из точки (10, 20) в точку (15, 20) можно построить из шести пикселей:

```
point( 10, 20 )
point( 11, 20 )
point( 12, 20 )
point( 13, 20 )
point( 14, 20 )
point( 15, 20 )
```

Нарисуем наклонную линию. Она, как и весь холст, тоже состоит из пикселей. Например, линия из точки (10, 20) в точку (15, 25) — диагональная, так что при увеличении x-координаты на 1 также увеличивается и y-координата:

```
point( 10, 20 )
point( 11, 21 )
point( 12, 22 )
point( 13, 23 )
point( 14, 24 )
point( 15, 25 )
```

А как нарисовать линию из точки (10, 20), скажем, в точку (15, 28)? Вычислить координаты нужных пикселей во многих случаях непросто. Для этой цели американский учёный Дж. Э. Брезенхэм в 1962 году придумал алгоритм, который используется в функции `line` из модуля `graph`. Нам достаточно просто вызвать эту функцию:

```
line( 10, 20, 15, 28 )
```

Функции передаются четыре аргумента: сначала координаты первого конца отрезка, затем — координаты второго конца. Линия будет иметь тот цвет, который был установлен до этого командой `penColor`.

Толщину линии можно при желании изменить командой `penSize` (от англ. *size* — размер):

```
penSize( 5 )
```

После выполнения этой команды все линии будут иметь толщину 5 пикселей, пока не выполнится новая команды `penSize`.

Линия — это один из *графических примитивов*. Так называются элементы рисунка, которые добавляются с помощью одной команды. В модуле `graph` есть ещё несколько примитивов, которые позволяют легко нарисовать прямоугольник, ломаную и окружность.

Прямоугольники

Теперь нарисуем прямоугольник, координаты углов которого показаны на рис. 1.3.

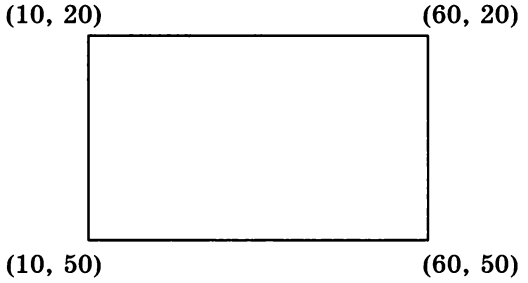


Рис. 1.3

Самый простой вариант — нарисовать его из четырёх отрезков, использовав четыре команды `line`:

```
line( 10, 20, 60, 20 )
line( 60, 20, 60, 50 )
line( 60, 50, 10, 50 )
line( 10, 50, 10, 20 )
```

Их можно объединить в одну команду `polygon`:

```
polygon( [(10,20),(60,20),(60,50),(10,50),(10,20)] )
```

Эта команда строит ломаную линию по точкам. Координаты точек перечисляются через запятую в квадратных скобках[1)], координаты каждой точки сгруппированы в круглые скобки.

Команда `polygon` позволяет построить любую ломаную линию, в том числе и незамкнутую. А для построения прямоугольников в модуле `graph` есть специальная команда `rectangle` (по-английски — прямоугольник):

```
rectangle( 10, 20, 60, 50 )
```

Ей передаются четыре аргумента: координаты левого верхнего угла (сначала x, потом — y) и координаты правого нижнего угла. Эта команда может нарисовать прямоугольник и сразу залить его каким-

[1)] Такая структура данных называется списком или массивом.

то цветом. Цвет заливки нужно заранее задать с помощью команды `brushColor` (от англ. *brush* — кисть и *color* — цвет):

```
penColor( "blue" )        # цвет границы
brushColor( "yellow" )    # цвет заливки
rectangle( 10, 20, 60, 50 )
```

Этот фрагмент программы нарисует прямоугольник с синей границей и жёлтой заливкой.

Отметим, что команда `polygon` тоже выполняет заливку, если ломаная замкнута (координаты первой и последней точек совпадают).

Окружность

Нарисовать окружность можно с помощью команды `circle` (по-английски — окружность):

```
penColor( "red" )
brushColor( "green" )
circle( 200, 150, 50 )
```

Первые два аргумента, переданные функции `circle`, — это координаты её центра (сначала — x, потом — y), а третий аргумент — радиус окружности. Будет построена окружность красного цвета (это цвет пера), а внутренняя часть круга будет залита зелёным цветом (это цвет кисти).

Для вычисления координат пикселей, из которых строится окружность, используется один из вариантов алгоритма Брезенхэма.

Изменение координат

Теперь напишем программу, которая вводит с клавиатуры координаты x и y и рисует квадрат на холсте в точке с заданными координатами. Здесь нужно разобраться в том, что считать координатами квадрата. Обычно выбирают некоторую *базовую* (или *опорную*) *точку* и договариваются, что координаты фигуры — это и есть координаты её опорной точки.

Для квадрата базовой точкой обычно считают его левый верхний угол. Обозначим его координаты через (x, y). Пусть сторона квадрата равна a. Тогда, учитывая, что x-координата увеличивается вправо, а y-координата — вниз, вычисляем координаты остальных углов (рис. 1.4).

Программа рисования квадрата с заданными координатами выглядит так:

```
print( "Введите координаты квадрата" )
x = int( input("x = ") )
y = int( input("y = ") )
a = 20
rectangle( x, y, x+a, y+a )
```

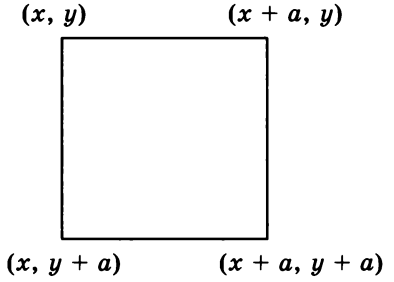


Рис. 1.4

Для рисования использована команда `rectangle` из модуля `graph`. Вообще-то она рисует прямоугольники, но её можно использовать и для квадрата, ведь квадрат — это частный случай прямоугольника. Команде `rectangle` передаются координаты левого верхнего и правого нижнего углов, которые мы определили ранее (см. рис. 1.4).

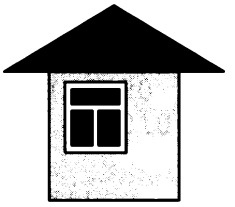
Выводы

- Холст — это область для рисования, которая хранится в памяти как растровый рисунок.
- Из программы можно управлять каждым пикселем холста — менять его цвет независимо от цветов других пикселей.
- Графический примитив — это геометрическая фигура, которая добавляется на рисунок с помощью одной команды.
- Основные графические примитивы — пиксель, линия (отрезок), прямоугольник, ломаная, окружность.
- Модуль — это файл на языке Python, содержащий функции. Для того чтобы использовать эти функции в своей программе, модуль нужно импортировать (подключить) командой `import`.
- Цвет контура фигур устанавливается с помощью команды `penColor`.
- Замкнутые контуры (прямоугольник, окружность, ломаная) заливаются тем цветом, который установлен как цвет заливки командой `brushColor`.

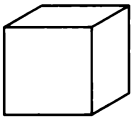
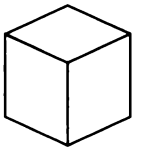
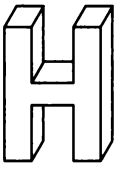
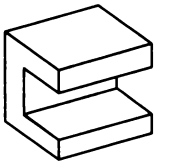
Вопросы и задания

1. Как вы думаете, почему в списке команд модуля `graph` нет команды «треугольник»?
2. Проверьте, что произойдёт, если рисовать за пределами холста.
3. Определите координаты пикселей, из которых состоят отрезки:
   а) (10,20) — (10, 15)
   б) (10,20) — (5, 15)
   в) (10,20) — (15,21)
   г) (10,20) — (15,22)
4. Выясните, что будет, если при рисовании прямоугольника задать координаты другой пары противоположных углов: правого верхнего и левого нижнего.

5. Посмотрите, как нарисованы кнопки в окне какой-нибудь программы. Нарисуйте изображения кнопки в нажатом и отпущенном положениях.
*6. Нарисуйте стандартное окно программы в вашей операционной системе.
7. Придумайте и нарисуйте своего персонажа, используя прямоугольники, окружности и ломаные.
8. Доработайте последнюю программу, приведённую в тексте параграфа, так, чтобы длину стороны квадрата также можно было вводить с клавиатуры.
9. Вычислите координаты всех углов квадрата при условии, что за опорную точку принят его центр.
10. Выберите опорную точку для прямоугольника и вычислите координаты всех его углов.
11. Напишите программы, которые строят такие рисунки (см. цветные рисунки на обороте обложки):

а)  б)  в) 

*12. Сделайте так, чтобы координаты фигуры, построенной при выполнении предыдущего задания, можно было менять, вводя с клавиатуры координаты опорной точки.
*13. Напишите программы, которые строят изображения трёхмерных объектов:

а)  б)  в)  г) 

Интересные сайты

kpolyakov.spb.ru/school/probook/python.htm — модуль `graph` для создания простых графических программ на языке Python

wiki.python.org/moin/TkInter — стандартная графическая библиотека языка Python (на английском языке)

www.riverbankcomputing.com/software/pyqt — библиотека `PyQt` для создания графических приложений на языке Python в разных операционных системах

§ 4
Процедуры

Ключевые слова:

- подпрограмма
- процедура
- рефакторинг
- аргументы
- параметры

Зачем нужны процедуры?

Рассмотрим такую программу, которая рисует домик на экране[1]:

```
penColor( "black" )
brushColor( "green" )
rectangle( 100, 100, 200, 200 )

brushColor( "brown" )
polygon( [(90,100), (150,50),
          (210,100), (90,100)] )

penColor( "white" )
penSize( 3 )
brushColor( "black" )
rectangle( 120, 120, 150, 170 )
line( 120, 140, 150, 140 )
line( 135, 140, 150, 170 )
```

Пустыми строками отделены друг от друга блоки программы, каждый из которых выполняет свою работу: первый рисует основную часть домика, второй — крышу, третий — окно. Запустив эту программу, мы обнаружим, что окно нарисовано неправильно. Как найти ошибку? Понятно, что нужно как-то определить блок рисования окна, но где он? Чтобы облегчить решение этой задачи, в программе пишут комментарии:

```
  # основная часть домика
penColor( "black" )
brushColor( "green" )
rectangle( 100, 100, 200, 200 )
  # крыша
brushColor( "brown" )
polygon( [(90,100),(150,50),
          (210,100),(90,100)] )
  # окно
penColor( "white" )
```

[1] Здесь и далее для сокращения записи мы не будем каждый раз писать команду импорта модуля `graph` и команду `run` в конце программы.

```
penSize( 3 )
brushColor( "black" )
rectangle( 120, 120, 150, 170 )
line( 120, 140, 150, 140 )
line( 135, 140, 150, 170 )
```

Но есть и другой приём: оформить блоки рисования каждой части домика как отдельные вспомогательные алгоритмы (*подпрограммы*, или *процедуры*) и дать им имена, например: `frame` (по-английски — сруб, основная часть дома), `roof` (крыша) и `window` (окно). Получается такая программа:

```
def frame():
  penColor( "black" )
  brushColor( "green" )
  rectangle( 100, 100, 200, 200 )

def roof():
  penColor( "black" )
  brushColor( "brown" )
  polygon( [(90,100), (150,50),
            (210,100), (90,100)] )

def window():
  penColor( "white" )
  penSize( 3 )
  brushColor( "black" )
  rectangle(120, 120, 150, 170 )
  line( 120, 140, 150, 140 )
  line( 135, 140, 150, 170 )

frame()
roof()
window()
```

Сначала посмотрим на последние три строки — это и есть основная программа. Сразу видно, что она состоит из трёх команд — вызовов процедур. Раньше мы использовали команды из встроенной библиотеки языка Python или модуля `graph`, а теперь создали такие команды сами.

Действия, которые компьютер выполняет при вызове этих процедур, определены выше с помощью служебных слов `def` (от английского *define* — определить). После слова `def` записывают имя новой процедуры, круглые скобки (позже мы узнаем, что можно записывать в этих скобках) и двоеточие. Все команды, которые входят в процедуру, записывают в следующих строках с одинаковым сдвигом вправо, например, на 2 или 4 символа.

Основная программа, где вызываются процедуры, записывается без отступа, начиная с первой позиции строки.

Теперь ясно, что, если неправильно рисуется окно, искать ошибку нужно в первую очередь в процедуре window, которая и занимается рисованием окна (исправьте ошибку самостоятельно).

То, что мы сейчас сделали с программой, в программировании называется рефакторингом.

Рефакторинг — это изменение программы, которое не влияет на результат её работы, но делает её более понятной для человека.

Процедура вызывает процедуру

Процедура может вызывать другие процедуры. Например, можно ввести в нашу программу ещё одну процедуру home, которая рисует весь домик:

```
def home():
  frame()
  roof()
  window()
```

Тогда вся основная программа будет представлять собой одну строку — вызов процедуры home:

```
home()
```

Процедуры с параметрами

Если попробовать вызвать только что написанную процедуру home дважды, мы не получим два домика. Дело в том, что эта процедура неуправляема: мы не можем ничего изменить в её работе, координаты и размеры всех фигур жёстко заданы внутри процедур frame, roof и window. А если хочется нарисовать несколько одинаковых домиков, но в разных местах холста?

Решить эту задачу можно, используя метод базовой точки, описанный в предыдущем параграфе. Выбираем одну из точек изображения домика в качестве базовой, обозначаем её координаты символами (например, x и y) и пересчитываем координаты всех остальных точек через x и y.

Получается, что координаты x и y нужно как-то передать процедуре home. Для этого и служат круглые скобки, в которых мы пока ничего не писали. Теперь можно ввести с клавиатуры координаты, записать их в переменные и передать процедуре, указав в скобках при вызове:

```
print( "Введите координаты домика" )
x = int( input("x = ") )
y = int( input("y = ") )
home( x, y )
```

Данные, передаваемые в процедуру, называют *аргументами* (как у функции в математике).

Теперь возникла другая проблема: процедура home ничего не знает о том, что ей передадут две координаты, она не готова их принять. Чтобы исправить ситуацию, нужно изменить определение процедуры home так:

```
def home( x, y ):
  frame( x, y )
  roof( x, y )
  window( x, y )
```

В первой строке указано, что процедура готова принять два значения, обозначенные как x и y. Данные, которые принимает процедура, называют параметрами.

Параметры — это данные, которые принимает процедура. В процедуре они обозначаются именами.

Параметры — это переменные, которые используются только внутри процедуры. Другие процедуры и основная программа не могут к ним обращаться.

Следующие строки процедуры home говорят о том, что значения x и y передаются в процедуры frame, roof и windows, которые вызываются из home. Эти процедуры тоже нужно переделать. Например, процедура frame рисует квадрат со стороной 100 пикселей, поэтому если его верхний левый угол имеет координаты (x, y), то координаты правого нижнего угла получаются равными (x+100, y+100). Вся процедура принимает вид:

```
def frame( x, y ):
  penColor( "black" )
  brushColor( "green" )
  rectangle( x, y, x+100, y+100 )
```

Остальные процедуры переделайте самостоятельно.

Теперь мы можем, вызывая несколько раз процедуру home с разными значениями x и y, рисовать домики в разных местах холста:

```
home( 100, 100 )
home( 300, 100 )
```

После выполнения такой программы вы увидите, что все линии второго домика жирные. Постарайтесь понять, почему так произошло, и исправить ситуацию.

Обратите внимание, что мы можем изменять только координаты домика, потому что только они передаются в процедуру home. Другие свойства (например, ширину домика и цвета) изменить пока нельзя. Если нужно управлять, например, шириной домика, её необходимо передать

в процедуру home как третий аргумент, а процедура home должна передать это значение остальным процедурам (их тоже придётся доработать).

Выводы

- Процедура — это отдельная часть программы, имеющая имя.
- Вызвать процедуру — значит записать её имя с круглыми скобками, в которых перечисляются аргументы — данные, передаваемые процедуре.
- Параметры — это данные, которые принимает процедура. В процедуре они обозначаются именами.
- Рефакторинг — это изменение программы, которое не влияет на результат её работы, но делает её более понятной для человека.

Вопросы и задания

1. Допишите программу для рисования домика, использующую процедуры с параметрами. Постройте с её помощью ряд из четырёх домиков (см. цветной рисунок на обороте обложки).

2. Измените процедуру для рисования домика так, чтобы при вызове можно было бы задать не только координаты, но и ширину сруба.
3. Напишите программы, которые строят такие рисунки с помощью нескольких вызовов одной процедуры (см. цветной рисунок на обороте обложки).

   а) б) в) г)

4. *Напишите программы, которые строят изображения трёхмерных объектов с помощью одной процедуры (см. цветной рисунок на обороте обложки).

   а) б)

§5
Обработка целых чисел

Ключевые слова:

- арифметические операции
- арифметические выражения
- остаток
- частное
- приоритет операций
- форматный вывод

Арифметические выражения

Арифметические выражения могут содержать *константы* (постоянные значения), имена переменных, знаки арифметических операций, круглые скобки (для изменения порядка действий).

Для основных арифметических операций в языке Python используются те же обозначения, что и в других языках программирования:

`+` — сложение; `-` — вычитание;
`*` — умножение; `/` — деление.

Кроме того, есть ещё операция возведения в степень, которая обозначается как `**`. Например, x^3 записывается как `x**3`.

Сначала рассмотрим операции с одной переменной. Оператор присваивания

```
i = i + 1
```

заменяет значение `i` на `i + 1`, т. е. увеличивает значение переменной `i` на 1. Эта операция часто используется для увеличения специальных переменных-счётчиков, с помощью которых считается количество каких-то событий (например, щелчков мышью) или объектов (например, количество собранных призов в игре).

Более сложное присваивание

$$a \leftarrow \frac{c+b-1}{2} \cdot d$$

запишется в программе так:

```
a = (c + b - 1)/2*d
```

Это *линейная запись* арифметического выражения (запись в одну строку). Длинное выражение можно перенести на следующую строку с помощью символа `\`:

```
a = x*(2*c + 4*b - 8*f) \
    + 2*a*c
```

При переносе внутри скобок символ `\` вставлять не обязательно:

```
a = (c + 5
     - 1) / 2*d
```

Эти правила переноса справедливы и для других операторов языка Python.

Порядок выполнения действий определяется *приоритетом* (старшинством) операций:

- сначала выполняются действия в скобках;
- затем — возведение в степень, справа налево, т. е. `2**3**2` — это то же самое, что и `2**(3**2)`;
- потом — умножение и деление, слева направо;
- в конце — сложение и вычитание, слева направо.

Таким образом, умножение и деление имеют одинаковый приоритет, более высокий, чем сложение и вычитание.

Например, в выражении

```
a = (c + b - 1)/2*d
```

сначала выполняются действия в скобках: сложение и затем вычитание; потом — деление на 2 и напоследок — умножение на `d`.

В языке Python разрешено множественное присваивание. Запись

```
a, b = 1, 2
```

равносильна паре операторов присваивания:

```
a = 1
b = 2
```

При этом считается, что эти два действия происходят параллельно, т. е. одновременно. Если двум переменным присваивается одно и то же значение, можно применить множественное присваивание «по цепочке»:

```
a = b = 0
```

Это равносильно паре операторов присваивания

```
b = 0
a = b
```

При изменении значений переменных удобно использовать сокращённую запись арифметических операций.

Полная запись	Сокращённая запись
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`

Если в выражение входят переменные разных типов, в некоторых случаях происходит автоматическое приведение типа к более «широкому». Например, результат умножения целого числа на вещественное —

это вещественное число. Переход к более «узкому» типу автоматически не выполняется. Нужно помнить, что результат деления (операция «/») — это вещественное число, даже если делимое и делитель — целые и делятся друг на друга нацело[1].

Деление нацело

Результат деления (операция «/») может быть нецелым числом, такие числа называются *вещественными*. Часто нужно получить целый результат деления целых чисел и остаток от деления. Например, известен интервал времени в секундах (скажем, 289 секунд) и нужно определить, сколько в нём целых минут и оставшихся секунд (289 с = = 4 мин 49 с). Здесь число минут 4 — это целая часть от деления 289 на 60, а 49 секунд — это остаток от этого деления.

В таких случаях используют специальные операторы:

`//` — деление нацело (с отбрасыванием остатка);
`%` — взятие остатка от деления.

Вот как выглядит программа, которая выделяет целые минуты и секунды (от 0 до 59) из интервала времени в секундах:

```
timeSec = 289
minutes = timeSec // 60    # = 4
seconds = timeSec % 60     # = 49
```

С помощью этих операций удобно работать с отдельными цифрами числа. Остаток от деления числа на 10 — это последняя цифра его десятичной записи[2]:

```
N = 123
lastDigit = N % 10    # = 3
```

Если разделить число на 10 и взять только целую часть, мы «отбросим» последнюю цифру числа: значение `123 // 10` равно 12:

```
N = 123
digits2 = N // 10    # = 12
```

Интересен результат выполнения операций `//` и `%` для отрицательных чисел. Программа

```
print ( -7 // 2 )
print ( -7 % 2 )
```

выводит на экран числа –4 и 1. Дело в том, что с точки зрения теории чисел остаток — это неотрицательное число, поэтому

$$-7 = (-4) \cdot 2 + 1,$$

[1] В некоторых языках (например, в C++) это не так: при делении целых чисел получается целое число и остаток отбрасывается.

[2] А остаток от деления целого числа на N — это последняя цифра записи числа в системе счисления с основанием N.

т. е. частное от деления –7 на 2 равно –4, а остаток равен 1. В языке Python (в отличие от многих других языков, например Паскаля и C++) эти операции выполняются математически правильно.

Вывод данных на экран

Вы уже знаете, что функция `print` вставляет по одному пробелу между элементами списка вывода:

```
a = 12
b = 5
c = 155
print(a, b, c)  # 12 5 155
```

Иногда требуется выводить данные в виде таблицы, выравнивая значения в каждом столбце по правой границе:

```
 12   5 155
211 315   8
```

Предположим, что мы работаем с натуральными числами, которые меньше 1000. Тогда на каждое число можно выделить 4 позиции на экране: три на запись числа и ещё один пробел слева, разделяющий числа. Записывается это так:

```
print( "{:4}{:4}{:4}".format(a, b, c) )
```

Это *форматный вывод*: строка для вывода строится с помощью встроенной функции `format`. Аргументы этой функции — `a`, `b` и `c` в скобках — это те данные, которые выводятся. Символьная строка слева от точки — это *форматная строка*, которая определяет, как именно данные будут представлены на экране.

Фигурные скобки обозначают место для вывода очередного элемента: на первом месте выводится значение `a`, на втором — значение `b` и на третьем — `c`.

Число после двоеточия — это количество позиций, которые отводятся на число. В пределах этого поля число прижимается к правой границе. Например, числа 12, 5 и 155 будут выведены так:

$$\underbrace{\circ\circ 12}_{4}\underbrace{\circ\circ\circ 5}_{4}\underbrace{\circ 155}_{4}$$

Здесь ∘ обозначает пробел.

Количество позиций можно не указывать:

```
print( "{}{}{}".format(a, b, c) )
```

Тогда данные выводятся вплотную друг к другу:

```
125155
```

Между данными из списка можно выводить и другие символы. Например, программа

```
num1 = 12; num2 = 13
print( "{}+{}={}".format(num1, num2, num1+num2) )
```

выведет

```
12+13=25
```

Как видно из первой строки программы, в одной строке можно записывать несколько операторов, разделяя их точками с запятой.

Выводы

- Арифметические выражения могут содержать константы (постоянные значения), имена переменных, знаки арифметических операций, круглые скобки (для изменения порядка действий).
- Порядок выполнения действий определяется *приоритетом* операций: сначала выполняется возведение в степень, затем — умножение и деление, затем — сложение и вычитание. Для изменения порядка действий используются скобки.
- Для деления нацело используется оператор `//`, для взятия остатка от деления — оператор `%`.
- Форматный вывод данных выполняется с помощью функции `format`.

Вопросы и задания

1. Что получится, если рассмотреть запись `i = i + 1` как равенство — уравнение относительно переменной `i`?
2. Чему будет равно значение переменной `i` после выполнения оператора `i = i + 1`, если до этого оно было равно 17?
3. Чему будут равны значения переменных `a` и `b` после выполнения программы
   ```
   a += 1
   b += 1
   a += b
   b += a
   a += 1
   ```
   если вначале они имели значения `a = 4` и `b = 7`?
4. Определите порядок действий компьютера при вычислении выражения:
   ```
   a = c + b - 1 / 2 * 5
   ```
5. Запишите присваивание
   $$z \leftarrow a + \frac{b-5}{c+8},$$
   используя линейную запись на языке Python.

6. Напишите программу, которая меняет местами значения двух переменных в памяти.
7. В предыдущей задаче попробуйте найти решение, которое не использует дополнительные переменные и работает не только в языке Python.
8. Как быстро определить, чему равен остаток от деления числа N на 100?
9. Как с помощью операций `//` и `%` выделить вторую с конца цифру числа?
10. Что будет выведено в результате работы следующей программы?

```
a = 1; b = 2; c = 3; d = 4; e = 5
print( "{:4}".format(a) )
print( "{:3}{:2}".format(b, b) )
print( "{:2}{:4}".format(c, c) )
print( "{}{}{}{}{}{}{}{}{}".format(d, d, d, d, d, d, d) )
print( "4}".format(e) )
```

11. Что будет выведено в результате работы фрагмента программы?

а)
```
a = 5; b = 3
print( a, "=Z(", b, ")", sep="" )
```

б)
```
a = 5; b = 3
print( "Z(a)=", "(b)", sep="" )
```

в)
```
a = 5; b = 3
print( "Z(", a, ")=(", a+b, ")", sep="" )
```

12. Запишите оператор для вывода значений целых переменных `a = 5` и `b = 3` в формате:

а) `3+5=?`
б) `Z(5)=F(3)`
в) `a=5; b=3;`
г) `Ответ: (5;3)`

13. Напишите программу, которая находит сумму, произведение и среднее арифметическое трёх целых чисел, введённых с клавиатуры. Например, при вводе чисел 4, 5 и 7 мы должны получить ответ:

```
4+5+7=16, 4*5*7=140, (4+5+7)/3=5.333333
```

14. Напишите программу, которая возводит введённое число в степень 10, используя только четыре операции умножения.
15. Пусть `a = 26` и `b = 6`. Для каждого из следующих фрагментов программы вычислите вручную значение целочисленной переменной `c`. Затем поверьте результат с помощью программы.

а)
```
c = a % b + b
```

б)
```
c = a // b + a
```

в)
```
b = a // b
c = a // b
```

г)
```
b = a // b + b
c  = a % b + a
```

д)
```
b = a % b + 4
c = a % b + 1
```

е)
```
b = a // b
c = a % (b + 1)
```

ж)
```
b = a % b
c = a // (b + 1)
```

16. Выполните предыдущее задание при `a = -22` и `b = 4`.
17. Напишите программу, которая вводит трёхзначное число и разбивает его на цифры. Например, при вводе числа 123 программа должна вывести: `1, 2, 3`
18. Напишите программу, которая вводит с клавиатуры количество секунд и выводит то же самое время в часах, минутах и секундах.
19. Занятия в школе начинаются в 8-30. Урок длится 45 минут, перерывы между уроками — 10 минут. Напишите программу, которая вводит с клавиатуры номер урока и выводит время его окончания.
20. Напишите программу, которая вычисляет стоимость нескольких пирожков. Программа должна ввести три числа: цену пирожка (два числа: рубли, потом — копейки) и количество пирожков. Требуется вывести сумму, которую нужно заплатить (рубли и копейки).
21. Напишите программу, которая вводит с клавиатуры четырёхзначное натуральное число и переставляет его первую и последнюю цифры, например, из числа 1234 должно получиться число 4231.
22. Напишите программу, которая вводит с клавиатуры четырёхзначное число и «вырезает» из него вторую цифру с начала, например из числа 1234 должно получиться число 134.
23. Напишите программу, которая вводит с клавиатуры четырёхзначное число и удаляет из него первую и последнюю цифры, например из числа 1234 должно получиться число 23.
24. Напишите программу, которая моделирует работу следующего автомата. Автомат получает на вход четырёхзначное число. Затем вычисляются три суммы: сумма первых двух цифр, сумма средних цифр и сумма последних двух цифр. Результат работы автомата — произведение этих сумм.

§6 Обработка вещественных чисел

Ключевые слова:

- вещественное число
- научный формат
- мантисса
- форматный вывод
- округление
- модуль

Что такое вещественное число?

Число, в котором выделяются целая и дробная части, в информатике называется вещественным. Для того чтобы отделить дробную часть от целой, используют точку (а не запятую, как принято в отечественной математической литературе).

Программа

```
x = 4.0
print(type(x))
```

выведет

```
<class 'float'>
```

Это означает, что созданная переменная x относится к типу (классу) float, т. е. хранит вещественное число. Заметьте, что дробная часть этого числа равна нулю, но мы специально указали её наличие с помощью точки. Это был сигнал для транслятора: «нужно создать вещественную переменную».

Очень большие или очень маленькие числа можно записывать в *научном формате*. Например, величину, равную заряду электрона ($1{,}60217662 \cdot 10^{-19}$ Кл), можно записать в переменную так:

```
qElectron = 1.60217662e-19
```

Здесь первую часть (1,60217662) часто называют *мантиссой* или значащей частью числа, а «–19» после буквы e — это порядок числа.

Тот же принцип используется и для записи больших чисел. Расстояние от Земли до Солнца ($1{,}496 \cdot 10^{11}$ м) записывается в переменную distToSun так:

```
distToSun = 1.496e11
```

В памяти компьютера вещественные числа хранятся не так, как целые. Они тоже записываются в двоичном коде и представляются как сумма степеней числа 2. Но, в отличие от целых чисел, в эту сумму включают и отрицательные степени двойки, например:

$$4{,}375 = 2^2 + 2^{-2} + 2^{-3} = 100{,}011_2.$$

Память компьютера не бесконечна, поэтому без ошибки могут храниться только те вещественные числа, которые точно равны сумме конечного количества степеней числа 2.

К сожалению, многие вещественные числа, например 0,1 и 0,9, в двоичном коде записываются как бесконечные дроби. Поэтому при их записи в память остаются только первые биты[1)], а остальные отбрасываются.

Большинство вещественных чисел хранится в памяти компьютера с ошибкой.

Эта проблема связана не с двоичной системой, а с ограниченным количеством бит, выделенных в памяти для хранения вещественного числа.

[1)] На хранение значащей части числа в памяти компьютера в большинстве систем выделяется 53 бита.

Вспомните, что число 1/3 не может быть точно записано в десятичной системе — его дробная часть содержит бесконечное число цифр. В компьютерах происходит то же самое.

Ошибки накапливаются при выполнении операций с вещественными числами и могут стать очень большими (например, при делении на неточное маленькое по модулю число), поэтому везде, где возможно, нужно стараться выполнять все расчёты, используя только целые числа.

Рассмотрим программу, которая складывает числа 0,1 и 0,2 и затем вычитает из суммы 0,3:

```
x = 0.1
y = 0.2
print(x + y)
diff = x + y - 0.3
print(diff)
```

Казалось бы, в математике мы должны получить 0, но программа выводит такой результат:

```
0.30000000000000004
5.551115123125783e-17
```

Первое число — это сумма значений x и y, которая должна быть равна 0,3, но немного отличается от этого значения из-за вычислительных ошибок. Второе число представляет значение $5{,}551115123125783 \cdot 10^{-17}$ — это ошибка вычисления разности.

Ввод и вывод

Как вы знаете, результат ввода с помощью функции input — это символьная строка. Если мы хотим ввести вещественное число, нужно затем преобразовать эту строку с помощью функции float:

```
x = float(input())
```

Теперь x — это вещественная переменная, и команда

```
print(type(x))
```

выведет

```
<class 'float'>
```

Несложно ввести несколько переменных в одной строке, используя тот же приём, что и для целых чисел (см. § 2):

```
x, y, z = map(float, input().split())
```

Здесь вводятся значения трёх переменных, x, y и z.

При выводе вещественных значений по умолчанию (т. е. если не сказано делать иначе) выводится 16 значащих цифр. Например, команда

```
print(16/7)
```

выводит

```
2.2857142857142856
```

Если такой вариант не устраивает, применяют *форматный вывод*, например:

```
x = 16/7
print( ">{:f}".format(x) )   # >2.285714
```

Здесь после двоеточия указан формат `f`, по умолчанию он оставляет 6 цифр в дробной части числа. Перед буквой `f` можно записать два числа через точку: первое задаёт общее количество позиций на вывод числа, а второе — количество цифр в дробной части:

```
print( ">{:10.3f}".format(x) )   # >     2.286
```

В этом варианте на число отводится всего 10 позиций, из них 3 — на дробную часть. Поскольку цифры и точка занимают 5 позиций, слева добавляется ещё 5 пробелов.

Если пробелы слева от числа не нужны, а требуется только ограничить количество знаков в дробной части, число слева от точки не пишут:

```
print( ">{:.3f}".format(x) )    # >2.286
```

Для очень больших или очень маленьких чисел используют *научный формат* (стандартный вид числа). Он обозначается буквой `e` внутри фигурных скобок:

```
x = 1e10/7
print( ">{:12.4e}".format(x) )   # > 1.4286e+09
```

Число слева от точки в строке формата — это общее количество позиций для вывода числа, а второе число — количество знаков в дробной части мантиссы (для всех чисел, кроме числа 0, она больше или равна 1 и меньше 10). Если первое число не указывать, будет использовано наименьшее возможное место.

Операции с вещественными числами

При работе с вещественными числами часто приходится округлять их до ближайших целых чисел. Для этого в языке Python есть две функции:

- `int( x )` — отбрасывание дробной части вещественного числа `x`;
- `round( x )` — округление вещественного числа `x` до ближайшего целого.

Функция `int` отбрасывает дробную часть числа, т. е. из числа 1,2 получается 1, а из числа −3,8 — число −3.

Другие математические функции объединены в модуль math. Для того чтобы вызывать эти функции из своей программы, подключим (импортируем) модуль с помощью команды import:

```
import math
```

После этого можно применять функции из этого модуля:

```
x = math.sqrt( 5 )
```

Здесь из модуля math вызывается функция sqrt, которая вычисляет квадратный корень из числа 5, т. е. находит число, квадрат которого равен 5.

Для обращения к функции модуля используется точечная запись: сначала записывают имя модуля, а затем через точку — название функции.

Возможен и другой вариант, когда подключается не весь модуль, а только некоторые функции из него:

```
from math import sqrt, pi
```

Этой строкой к программе подключены из модуля math уже знакомая вам функция sqrt и константа (постоянная) pi, равная иррациональному числу π (3,1415626...). В этом случае для обращения к ним уже не нужно будет указывать имя модуля:

```
x = sqrt( 5 )
R = 12
circleLen = 2*pi*R
```

Можно подключить сразу все функции из модуля, если написать знак * вместо списка функций:

```
from math import *
```

Именно такой способ мы использовали при работе с графической библиотекой — модулем graph. При этом к функциям также можно обращаться просто по имени, не указывая имя модуля.

Выводы

- Большинство вещественных чисел хранится в памяти компьютера с ошибкой. Она вызвана тем, что для хранения числа выделяется конечное число двоичных разрядов.
- Для преобразования символьной строки в вещественное число используется встроенная функция float.
- Для ввода и вывода вещественных чисел можно использовать научный формат (стандартный вид числа).
- Математические функции объединены в модуль math.
- Функция из модуля вызывается с помощью точечной записи (имя модуля.имя функции).

Вопросы и задания

1. Вычислите значение вещественной переменной `c` при `a = 2` и `b = 3`:
   - **а)** `c = a + 1/3`
   - **б)** `c = a + 4/2*3 + 6`
   - **в)** `c = (a + 4)/2*3`
   - **г)** `c = (a + 4)/(b + 3)*a`
2. Что будет выведено в результате работы следующей программы?

```
x = 172.3658
print( x )
print( "{:10.2f}".format(x) )
print( "{:.8f}".format(x) )
```

3. Программа вывела числа в научном формате:

```
1.2345E+001
2.345E+003
5.6E+005
8.74E+00
1.8752E-01
3.462752E-03
```

   Запишите их в «обычном» виде.

4. Что будет выведено на экран в результате работы следующей программы?

```
a = 1; b = 2; c = 3; d = 7
print( "{:.2f}".format(a/b) )
x = b / c
print( "{:.2f} {:2}".format(x, int(x)) )
print( "{:.2f}".format(x-int(x)) )
x = d / c
print( "{:.2f} {:2}".format(x, int(x)) )
print( "{:.2f}".format(x-int(x)) )
```

5. Напишите программу, которая вычисляет квадратный корень введённого числа. Вычислите с её помощью квадратные корни из чисел 221841; 32005,21 и 15239,9025.
6. Вычислите вручную сумму $X = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} = \frac{n}{d}$ в виде простой дроби. Затем проверьте, что выведет эта программа (вместо многоточий добавьте полученные значения `n` и `d`):

```
n = ...; d = ...
y = 1/2 + 1/3 + 1/4 + 1/5
x = n/d
print( x )
print( y )
print( x-y )
```

   Сделайте выводы.

Интересный сайт

h-schmidt.net/FloatConverter/IEEE754.html — представление вещественных чисел в памяти компьютера (онлайн-калькулятор)

§7 Случайные и псевдослучайные числа

Ключевые слова:

- случайные числа
- псевдослучайные числа
- генератор случайных чисел

Случайные и псевдослучайные числа

В некоторых задачах, в том числе в компьютерных играх, необходимо моделировать случайные явления, например результат бросания игрального кубика: на нём может выпасть число от 1 до 6. Как сделать это на компьютере, который «неслучаен», т. е. строго выполняет заданную ему программу?

Случайные числа — это последовательность чисел, в которой невозможно предсказать следующее число, даже зная все предыдущие.

Чтобы получить истинно случайные числа, можно, например, бросать игральный кубик или измерять какой-то шумовой сигнал (например, шум радиоэфира или сигнал, принятый из космоса). Так раньше составлялись таблицы случайных чисел, которые публиковались в книгах.

Но вернёмся к компьютерам. Ставить сложные электронные приборы на каждый компьютер очень дорого, и повторить эксперимент будет невозможно — завтра все значения шумовых сигналов будут уже другими. Существующие таблицы слишком малы, когда, скажем, нужно получать 100 случайных чисел каждую секунду. Для хранения больших таблиц требуется много памяти.

Чтобы выйти из положения, математики придумали алгоритмы получения *псевдослучайных* («как бы случайных») чисел. Такие алгоритмы называются *генераторами* или *датчиками случайных чисел.*

Для стороннего наблюдателя псевдослучайные числа практически неотличимы от случайных, но они вычисляются по некоторой математической формуле: зная первое число (зерно), можно по формуле вычислить второе, затем третье и т. п.

Пишем свой генератор случайных чисел

Чтобы понять, как это работает, давайте сделаем свой вариант генератора псевдослучайных чисел.

Долгое время был очень популярен *линейный конгруэнтный метод*, предложенный в 1949 году американским математиком Д. Г. Лемером. Следующее псевдослучайное число — X_{n+1} — вычисляется по предыдущему — X_n — с помощью формулы

$$X_{n+1} = (aX_n + c) \% m.$$

Здесь a, c и m — некоторые натуральные числа, а знак % обозначает остаток от деления.

Понятно, что остаток от деления на m — это число на отрезке $[0; m-1]$, т. е. по крайней мере через m шагов последовательность повторится! Задача выбора параметров a, c и m заключается в том, чтобы добиться максимально возможной длины последовательности.

Возьмём, например, $a = 3$, $c = 1$ и $m = 8$. Выберем зерно $X_0 = 0$ и несколько раз применим приведённую выше формулу:

$$X_1 = (3 \cdot 0 + 1) \% 8 = 1$$
$$X_2 = (3 \cdot 1 + 1) \% 8 = 4$$
$$X_3 = (3 \cdot 4 + 1) \% 8 = 5$$
$$X_4 = (3 \cdot 5 + 1) \% 8 = 0$$

Получилось так, что мы снова вышли на зерно 0, т. е. дальше значения будут опять повторяться: 1, 4, 5, 0, 1, 4, 5, 0, ... Другие числа, кроме 0, 1, 4 и 5, вообще в последовательности не появляются. Период такой последовательности — 4.

Если же выбрать $a = 1$, $c = 3$ и $m = 8$, для того же зерна $X_0 = 0$ получаем последовательность

$$0, 3, 6, 1, 4, 7, 2, 5, 0, \ldots$$

в которой есть все числа от 0 до 7 включительно, т. е. она имеет наибольший возможный (при выбранном m) период 8.

Хорошие значения a, c и m, обеспечивающие большой период последовательности, можно найти в литературе или в Интернете. Например, генератор случайных чисел в семействе компиляторов GCC использует $a = 1103515245$, $c = 12345$ и $m = 2^{31}$.

Генератор случайных чисел в Python

Функции для работы с псевдослучайными числами собраны в модуле `random`. В библиотеке Python используется один из наиболее совершенных алгоритмов для генерации псевдослучайных чисел — «вихрь Мерсенна», разработанный в 1997 году.

Для получения псевдослучайных чисел в заданном диапазоне мы будем использовать функции из модуля `random`:

- `randint( a, b )` — случайное целое число из отрезка [a; b];
- `uniform( a, b )` — случайное вещественное число из отрезка [a; b].

Для того чтобы записать в переменную `n` случайное число в диапазоне от 1 до 6 (результат бросания игрального кубика), можно использовать такие команды:

```
from random import randint
n = randint( 1, 6 )
```

В первой строке из модуля `random` импортируется (загружается) функция `randint`, во второй она вызывается для получения случайного числа.

Вещественное случайное число на отрезке [5; 12] получается так:

```
from random import uniform
x = uniform( 5, 12 )
```

Выводы

- Случайные числа — это последовательность чисел, в которой невозможно предсказать следующее число, даже зная все предыдущие.
- При моделировании на компьютерах чаще всего используются псевдослучайные числа — последовательность чисел, которая строится по некоторой формуле.
- В языке Python все функции для работы со случайными числами находятся в модуле `random`.
- Случайное целое число на отрезке [a; b] можно получить с помощью вызова `randint(a, b)`, а случайное вещественное число на отрезке [a; b] строится как `uniform( a, b )`.

Вопросы и задания

1. Напишите программу, которая вводит с клавиатуры два целых числа, a и b ($a < b$), и выводит на экран 5 случайных целых чисел на отрезке $[a; b]$.
2. В игре «Русское лото» из мешка случайным образом выбираются бочонки, на каждом из которых написано число от 1 до 90. Напишите программу, которая выводит наугад первые 5 выигрышных номеров.
3. Доработайте программу «Русское лото» так, чтобы все 5 значений гарантированно были разными.
4. Напишите программу, которая моделирует бросание двух игральных кубиков: при запуске выводит случайное число на отрезке [2; 12].
5. Игральный кубик бросается три раза (выпадают три случайных значения). Из этих чисел составляется целое число, программа должна найти его квадрат.

6. Напишите программу, которая случайным образом выбирает дежурных: выводит два случайных числа на отрезке [1; N], где N — количество учеников вашего класса. Какая проблема может при этом возникнуть?
7. При программировании компьютерной игры вам нужно выбрать случайное направление, откуда появляется новый метеорит. Как использовать для этой цели генератор случайных чисел?
8. Напишите программу, которая выводит на холст в графическом окне 5 кругов со случайным радиусом, случайными координатами центра, случайным цветом контура и случайным цветом заливки.

*9. Напишите вариант программы из предыдущего задания, которая выводит не круги, а домики случайного размера и цвета. Координаты также должны выбираться случайно.

§ 8 Ветвления

Ключевые слова:

- условный оператор
- полная форма условного оператора
- неполная форма условного оператора
- вложенный условный оператор
- логические переменные

Условный оператор

Сейчас мы умеем писать *линейные* программы, в которых операторы выполняются последовательно друг за другом, и порядок выполнения операторов не зависит от входных данных.

В большинстве реальных задач порядок действий может изменяться в зависимости от того, какие данные поступили. Например, программа для системы пожарной сигнализации должна выдавать сигнал тревоги, если максимальное из показаний двух датчиков температуры больше, чем предельно допустимое значение. Для этого нужно сначала найти это максимальное значение, т. е. записать его в какую-нибудь переменную.

Выполним *формализацию* задачи — запишем её «в буквах»:

Требуется записать в переменную `M` *наибольшее из значений переменных* `a` *и* `b`.

Для решения таких задач в языках программирования предусмотрены *условные операторы* (ветвления). Например, для того, чтобы записать в переменную `M` *максимальное* (наибольшее) из значений переменных `a` и `b`, можно использовать оператор:

```
if a > b:
  M = a
else:
  M = b
```

Слово **if** переводится с английского языка как «если», а слово **else** — как «иначе». Если верно (истинно) условие, записанное после слова **if**, то выполняются все команды (также говорят «блок команд»), которые расположены до слова **else**. Если же условие после **if** неверно (ложно), выполняются команды, стоящие после **else**.

В Python, в отличие от других языков, сдвиги операторов относительно левой границы (отступы) влияют на работу программы. Обратите внимание, что слова **if** и **else** начинаются на одном уровне, а все команды внутренних блоков сдвинуты относительно этого уровня вправо на одно и то же расстояние. Для сдвига используют пробелы (обычно не меньше двух) или символы табуляции (которые вставляются при нажатии на клавишу *Tab*).

Кроме знаков < и > в условиях можно использовать другие знаки отношений: <= (меньше или равно), >= (больше или равно), == (равно, два знака «равно» без пробела, чтобы отличить от оператора присваивания) и != (не равно).

Если в блоке всего один оператор, иногда бывает удобно записать блок в той же строке, что и служебное слово **if** (**else**):

```
if a > b: M = a
else:     M = b
```

В приведённых примерах условный оператор записан в полной форме: в обоих случаях (истинно условие или ложно) нужно выполнить некоторые действия.

Неполная форма условного оператора

Программа выбора максимального значения может быть написана иначе:

```
M = a
if b > a: M = b
```

Здесь использован условный оператор в неполной форме, потому что в случае, когда условие ложно, ничего делать не требуется (нет слова `else` и блока операторов после него).

Поскольку операция выбора максимального из двух значений нужна очень часто, в Python есть встроенная функция `max`, которая вызывается так:

```
M = max( a, b )
```

Есть также и аналогичная функция `min`, которая выбирает минимальное из двух или нескольких значений.

Если выбирается максимальное из двух чисел, можно использовать особую форму условного оператора в Python:

```
M = a if a > b else b
```

которая работает так же, как и приведённый выше условный оператор в полной форме: записывает в переменную M значение a, если a > b, и значение b, если это условие ложно.

Часто при истинности какого-то условия нужно выполнить сразу несколько действий. Например, в задаче сортировки значений переменных a и b по возрастанию нужно поменять местами значения этих переменных, если a > b:

```
if a > b:
  temp = a
  a = b
  b = temp
```

Здесь temp — это временная (вспомогательная) переменная (от англ. *temporary* — временный). Все операторы, входящие в блок, сдвинуты на одинаковое расстояние от левого края. Начало и конец блока, который выполняется при истинности условия, определяется именно этими сдвигами. Поэтому операторные скобки — специальные ограничители блоков (как, например, слова begin и end в языке Паскаль или фигурные скобки в С-подобных языках) здесь не нужны.

Заметим, что в Python, в отличие от многих других языков программирования, есть множественное присваивание, которое позволяет выполнить такой обмен значительно проще:

```
a, b = b, a
```

Вложенные условные операторы

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы.

Например, пусть возраст Андрея записан в переменной ageA, а возраст Бориса — в переменной ageB. Нужно определить, кто из них старше. Одним условным оператором тут не обойтись, потому что есть три возможных результата: старше Андрей, старше Борис или оба одного возраста. Решение задачи можно записать так:

```
if ageA > ageB:
  print( "Андрей старше" )
else:
  if ageA == ageB:
    print( "Одного возраста" )
  else:
    print( "Борис старше" )
```

Условный оператор, проверяющий равенство (он выделен фоном), находится внутри блока «иначе» (**else**), поэтому он называется вложенным условным оператором.

Как видно из этого примера, использование вложенных условных операторов позволяет выбрать один из нескольких вариантов (а не только из двух).

Если после **else** сразу следует ещё один оператор **if**, можно использовать так называемое «каскадное» ветвление с ключевыми словами **elif** (сокращение от **else-if**). Если очередное условие ложно, выполняется проверка следующего условия и т. д.:

```
if ageA > ageB:
  print( "Андрей старше" )
elif ageA == ageB:
  print( "Одного возраста" )
else:
  print( "Борис старше" )
```

Обратите внимание на отступы: слова **if**, **elif** и **else** находятся на одном уровне.

В цепочке операторов **if-elif-elif-...** срабатывает первое истинное условие. Например, программа

```
cost = 1500
if cost < 1000:
  print( "Скидок нет." )
elif cost < 2000:
  print( "Скидка 2%." )
elif cost < 5000:
  print( "Скидка 5%." )
else:
  print( "Скидка 10%." )
```

выведет «`Скидка 2%.`», хотя условие `cost < 5000` тоже выполняется.

Логические переменные

Во многих языках можно использовать переменные, в которых хранятся логические значения («да»/«нет», истина/ложь). В языке Python такие переменные могут принимают значения `True` («истина») или `False` («ложь»):

```
b = True
b = False
print(type(b) )
```

Эта программа выведет:

```
<class 'bool'>
```

т. е. переменная b относится к типу **bool**. Такая переменная называется *логической* или *булевой* в честь английского математика Джорджа Буля — создателя алгебры логики.

В логической переменной можно хранить значение какого-то условия и затем использовать его в условном операторе. Пусть нужно выяснить, есть ли среди значений переменных a, b и c хотя бы два равных. Нам нужно проверить равенство в трёх парах: a и b, a и c, b и c. Используем логическую переменную вместо вложенных условных операторов:

```
found = False   # пока не нашли пару равных
if a == b: found = True  # проверяем пару a, b
if a == c: found = True  # проверяем пару a, c
if b == c: found = True  # проверяем пару b, c
```

Первые две строки можно заменить одной:

```
found = (a == b)
```

которая записывает в переменную found значение True, если выполняется условие справа от оператора присваивания, т. е. если значения a и b равны.

Теперь логическую переменную можно использовать вместо условия в условном операторе:

```
if found:
  print( "Нашли равные!" )
else:
  print( "Равных нет." )
```

Экспертная система (проект)

Эксперт — это человек, который обладает глубокими теоретическими знаниями и практическим опытом работы в некоторой области. Например, врач-эксперт хорошо ставит диагноз и лечит потому, что имеет медицинское образование и большой опыт лечения пациентов. Он не только знает факты, но понимает их взаимосвязь, может объяснить причины явлений, сделать прогноз, найти решение в конфликтной ситуации.

Экспертная система — это компьютерная программа, задача которой — заменить человека-эксперта при выработке рекомендаций для принятия решений в сложной ситуации.

Экспертная система содержит базу знаний, в которой хранятся не только факты, но и правила, по которым из этих фактов делаются выводы.

Мы построим простейшую экспертную систему, которая задаёт пользователю вопросы и по его ответам определяет класс животных.

Предположим, что в базу знаний внесены следующие правила:

- если у животного есть перья, то это — птица;
- если животное кормит детёнышей молоком, то это — млекопитающее;
- если животное — млекопитающее и ест мясо, то это — хищник.

Диалог пользователя с экспертной системой может быть, например, таким (ответы пользователя выделены курсивом):

```
Это животное кормит детей молоком? нет
Это животное имеет перья? да
Это птица.
```

Для того чтобы определить последовательность вопросов, эксперт строит *дерево решений*, например такое (рис. 1.5).

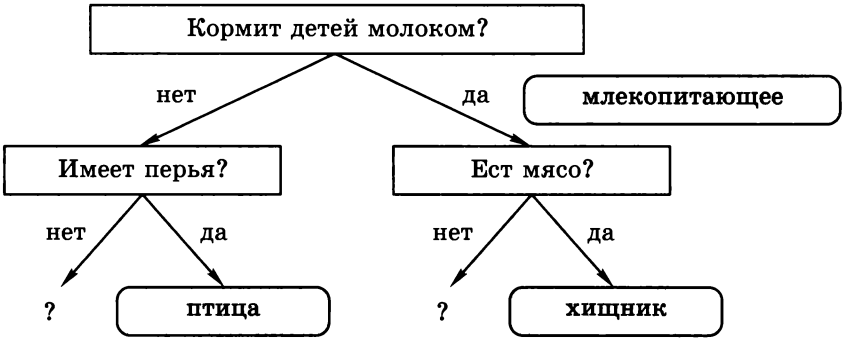


Рис. 1.5

В прямоугольниках записаны вопросы, которые задаёт система пользователю, у стрелок — его возможные ответы («да» или «нет»). Серым фоном выделены **выводы** — результат работы экспертной системы.

Конечно, эта система позволяет определить не все классы животных: в некоторых местах на схеме стоят знаки вопроса. В этих случаях наша программа будет выводить ответ «не знаю».

Человеку удобнее вводить ответ словами («да», «нет»), т. е. в виде символьной строки. Именно в такой форме возвращает результат ввода функция `input`, с которой мы уже знакомы.

Программа начинает диалог с вопроса «Кормит детей молоком?» и в зависимости от ответа выбирает одну из двух ветвей дерева решений (см. рис. 1.5).

```
otvet = input( "Кормит детей молоком? " )
if otvet == "да":
  ... # вариант 1
else:
  ... # вариант 2
```

Конечно, вместо многоточий должны быть добавлены команды, которые нужно выполнить в каждом случае.

Обратите внимание, что символьные строки можно сравнивать с помощью оператора == так же, как и числа.

Разберём дальнейшие действия системы при первом ответе «да». Во-первых, нужно вывести первый результат: Млекопитающее. Во-вторых, задаём второй вопрос и в зависимости от ответа на него сообщаем второй результат:

```
print( "Млекопитающее." )
otvet = input( "Ест мясо? " )
if otvet == "да":
  print( "Хищник." )
else:
  print( "Не знаю." )
```

Вторую ветвь главного условного оператора (вариант 2) вы можете написать самостоятельно.

Выводы

- Условный оператор **if-else** позволяет выбрать один из двух вариантов действий в зависимости от выполнения некоторого условия.
- Условие, которое нужно проверить, записывается после слова **if**. Записывать обратное условие после **else** не нужно.
- Если условие верно, выполняются все команды, записанные после строки с оператором **if** (со сдвигом вправо).
- Если условие неверно, выполняются все команды, записанные после строки с оператором **else** (со сдвигом вправо).
- В обеих частях условного оператора можно использовать любые операторы, в том числа и другие (вложенные) условные операторы.
- В логической переменной можно хранить логическое значение: True (истина) или False (ложь).
- Экспертная система — это компьютерная программа, задача которой — заменить человека-эксперта при выработке рекомендаций для принятия решений в сложной ситуации.

Вопросы и задания

1. Какие задачи невозможно решить с помощью линейных алгоритмов?
2. Как вы думаете, хватит ли линейных алгоритмов и ветвлений для разработки любой программы?
3. Почему нельзя выполнить обмен значений двух переменных в два таких шага?
   ```
   a = b
   b = a
   ```
4. Можно ли переставлять операторы в алгоритме обмена значений двух переменных, приведённом в параграфе? Если нет, приведите контрпример, когда перестановка даст неверный результат.

5. Как вы думаете, можно ли обойтись только неполной формой условных операторов?
6. Чем различаются операторы = и ==?
7. Объясните, чем различаются следующие фрагменты программ:

```
if a > b: a = b
a = c
```

и

```
if a > b: a = b
else:     a = c
```

Приведите примеры исходных данных, для которых результаты выполнения обеих программ (значение переменной a) будут одинаковыми, и примеры данных, для которых они будут различными.

8. Объясните, чем различаются следующие фрагменты программ:

```
if a > b: a = b
if a > c: a = c
```

и

```
if a > b: a = b
elif a > c: a = c
```

Приведите примеры исходных данных, для которых результаты выполнения обеих программ (значение переменной a) будут одинаковыми, и примеры данных, для которых они будут различными.

9. Программист написал программу для выбора наименьшего из двух чисел так:

```
if a < b: m = a
if b < a: m = b
```

В каких случаях эта программа будет работать неправильно? Запишите программу правильно, используя один условный оператор в полной форме.

10. Можно ли в этой программе два условных оператора в неполной форме заменить на один оператор в полной форме? Почему?

```
if a < 5:  a = 5
if a > 10: a = 10
```

Что делает эта программа?

11. Напишите программу, которая определяет модуль введённого целого числа.
12. Напишите программу, которая вводит целое число и выводит ответ «да», если оно чётное. В этой и аналогичных задачах нужно вывести ответ «нет», если нужное свойство не выполняется.
13. Напишите программу, которая вводит трёхзначное целое число и выводит ответ «да», если в его записи есть цифра 0. Попробуйте обойтись одним условным оператором.
14. Покажите, что приведённая программа не всегда верно определяет максимальное из трёх чисел, записанных в переменные a, b и c:

```
if a > b: M = a
else:     M = b
if c > b: M = c
else:     M = b
```

Приведите контрпример, т. е. значения переменных, при которых в переменной M будет получен неверный ответ. Как нужно доработать программу, чтобы она всегда работала правильно?

15. Напишите программу, которая выбирает максимальное и минимальное из трёх (четырёх, пяти) введённых чисел (не используя встроенные функции min и max).

16. Что выведет эта программа при x = -3? x = 0? x = 123?

```
if x >= 0:
  if x > 0:
    print(1)
  else:
    print(0)
else:
  print(-1)
```

17. Напишите другой вариант решения задачи из параграфа, в которой сравнивается возраст Андрея и Бориса (с другой последовательностью проверки условий). Сколько всего вариантов можно придумать?

18. Напишите программу, которая вводит с клавиатуры возраст трёх человек (Антона, Бориса и Виктора) и определяет, кто из них старше.

19. Напишите программу, которая моделирует работу следующего автомата. Автомат получает на вход трёхзначное натуральное число и строит новое число следующим образом:

    1) вычисляются суммы первой и второй, затем — второй и третьей цифр;
    2) эти суммы записываются в порядке невозрастания.

    Например, для числа 639 получаем суммы: 6 + 3 = 9; 3 + 9 = 12. Результат: 129. Ваша программа должна принимать введённое с клавиатуры трёхзначное число и определять число, которое будет получено в результате работы автомата.

20. Напишите программу, которая моделирует работу следующего автомата. Автомат получает на вход четырёхзначное натуральное число и строит новое число следующим образом:

    1) вычисляются суммы первой и второй; второй и третьей; третьей и четвёртой цифр;
    2) из полученных сумм отбрасывается наименьшая;
    3) остальные суммы записываются в порядке неубывания.

    Например, для числа 1284 получаем суммы: 1 + 2 = 3; 2 + 8 = 10; 8 + 4 = 12. Наименьшая сумма 3 отбрасывается, результат 1012.

Ваша программа должна принимать введённое с клавиатуры четырёхзначное число и определять число, которое будет получено в результате работы автомата.

21. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение `True` (истина), если среди введённых чисел есть хотя бы одно, большее 100.
22. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение `True` (истина), если все эти числа чётные.
23. *Проект.* Постройте свою небольшую экспертную систему по интересующей вас тематике.

§9 Сложные условия

Ключевые слова:

- сложное условие
- операция И
- операция ИЛИ
- операция НЕ
- равносильные условия
- порядок выполнения операций

Операция И

Предположим, что ООО «Кнут и Пряник» набирает сотрудников, возраст которых от 25 до 40 лет включительно. Нужно написать программу, которая запрашивает возраст претендента и выдаёт ответ: подходит он или не подходит по этому признаку.

Какое же условие должно быть истинно для того, чтобы человека приняли на работу? Одного условия `возраст >= 25` не хватает, это условие соблюдается и для людей старше 40 лет. Вместе с тем условия `возраст <= 40` тоже недостаточно, так как оно выполняется и для школьников. В этой задаче нужно, чтобы два **условия выполнялись одновременно:** `возраст >= 25` и `возраст <= 40`.

Эту задачу можно решить с помощью вложенного условного оператора, но решение получается некрасивым: оно запутанное и, кроме того, один и тот же ответ `"не подходит"` приходится выводить в двух местах программы.

Почти во всех языках программирования в условном операторе можно использовать такое условие:

```
if age >= 25 and age <= 40:
  print( "подходит" )
else:
  print( "не подходит" )
```

Решение получилось короткое и понятное. В условном операторе мы записали **сложное условие**

```
age >= 25 and age <= 40
```

составленное из двух простых с помощью логической операции И. В языке Python эта операция обозначается словом **and**.

Операция И (and) означает одновременное выполнение двух или нескольких условий.

В программе на языке Python можно сразу проверить выполнение двойного неравенства:

```
if 25 <= age <= 40:
  print("подходит")
else:
  print("не подходит")
```

но во многих популярных языках программирования такая запись считается ошибкой.

Пример

Предположим, что нам надо убедиться, что значение целой переменной a — трёхзначное число, которое делится на 7. Для этого нужно, чтобы одновременно выполнились три условия:

1) число не меньше 100;
2) число меньше 1000;
3) число делится на 7, т. е. остаток от его деления на 7 равен нулю.

В условном операторе эти три простых условия должны быть связаны с помощью двух операций И:

```
if 100 <= a and a < 1000 and a % 7 == 0:
  print("Да!")
else:
  print("Нет.")
```

Для того чтобы было легче разбираться в программах со сложными условиями, можно использовать логические переменные, например так[1]:

```
digits3 = (100 <= a and a < 1000)
div7 = (a % 7 == 0)
if digits3 and div7:
  print("Да!")
else:
  print("Нет.")
```

[1] Значение, которое записывается в логическую переменную, не обязательно брать в скобки, но так запись становится более понятной.

Здесь мы сначала проверили, верно ли, что a — трёхзначное число, и записали результат в логическую переменную digits3. Затем выяснили, делится ли оно на 7, и записали результат в логическую переменную div7. Условие в условном операторе объединяет два рассмотренных условия с помощью операции И. Программа получилась немного более длинной, зато стала понятнее. Здесь важно, что выбраны «говорящие» названия логических переменных: по-английски *digits* означает «цифры», а *div* — сокращение от слова *division* — деление.

Операция ИЛИ

Рассмотрим ещё одну задачу. Самолёт из Санкт-Петербурга в Барнаул летает только по понедельникам и четвергам. В переменной day записан номер дня недели (1 — понедельник, 7 — воскресенье). Программа должна определить, полетит ли самолёт в этот день.

Если мы напишем условие day = 1 **and** day = 4, то это будет неверно, потому что это условие требует, чтобы значение переменной day было *одновременно* равно и 1, и 4. Такого быть не может, поэтому это условие всегда будет ложно. Значит, операция И не подходит. Вместо неё нужно применить другую операцию — ИЛИ, которая требует выполнения хотя бы одного из набора условий.

Операция ИЛИ означает выполнение хотя бы одного из двух или нескольких условий.

Решение нашей задачи выглядит так:

```
if day == 1 or day == 4:
  print( "Полетит!" )
else:
  print( "Нет рейса." )
```

В языке Python операция ИЛИ обозначается словом or (по-английски — «или»).

Можно решить задачу с помощью логической переменной:

```
fly = ( day == 1 or day == 4 )
if fly:
  print( "Полетит!" )
else:
  print( "Нет рейса." )
```

Операция НЕ

Существует ещё одна операция, которую можно использовать в сложных условиях — НЕ, в Python она обозначается словом **not** (по-английски — «не»).

Операция НЕ означает обратное условие (противоположное исходному).

Если исходное условие истинно, то обратное (противоположное) ему — ложно, и наоборот.

Например, решение задачи с самолётом можно было записать так:

```
if not(day == 1 or day == 4):
  print( "Нет рейса." )
else:
  print( "Полетит!" )
```

Используя операцию НЕ, можно записывать условия по-разному, как нам удобнее в каждом случае. Например, условия `a==b` и `not(a!=b)` истинны для одних и тех же значений `a` и `b`, поэтому одно из них можно заменить на другое. Такие условия называются *равносильными*.

Приведём ещё примеры равносильных условий. Условие

```
not(x>=0 and x<=10)
```

означает «`x` не находится внутри отрезка [0; 10]». Это значит, что значение `x` на числовой оси расположено левее нуля *или* правее, чем 10. Поэтому его можно записать иначе, без использования операции НЕ:

```
x<0 or x>10
```

Обратите внимание, что в исходном выражении простые условия были связаны с помощью операции И, а в равносильном — с помощью ИЛИ.

Условие `not(x==2 or x==5)` означает, что значение `x` не равно ни двум, ни пяти, т. е. верно условие

```
x!=2 and x!=5
```

Здесь при переходе к равносильному условию без НЕ логическая операция ИЛИ была заменена на И.

Порядок выполнения операций

Операции И, ИЛИ и НЕ — это **логические операции**, т. е. они применяются к логическим значениям («да»/«нет», «истина»/«ложь»).

Если в сложном условии встречается несколько разных операций, они выполняются в следующем порядке (во всех случаях — слева направо):

1) операции в скобках;
2) операции НЕ;
3) операции И;
4) операции ИЛИ.

Изменить порядок действий можно с помощью круглых скобок.

Выводы

- Операция И (**and**) означает одновременное выполнение двух или нескольких условий.
- Операция ИЛИ (**or**) означает выполнение хотя бы одного из двух или нескольких условий.
- Операция НЕ (**not**) означает обратное условие (противоположное исходному).
- При определении истинности условия сначала выполняются действия в скобках, потом — операции НЕ, затем — операции И и в самом конце — операции ИЛИ.

Вопросы и задания

1. Что будет выведено на экран после выполнения программы при `cats = 2`?

```
if cats = 1 and cats = 2:
  print( "Да! Получилось!" )
else:
  print( "Нет. Не вышло." )
```

2. Найдите другой вариант решения задачи из параграфа о вылете самолёта из Санкт-Петербурга в Барнаул: с использованием операции И.
3. Запишите равносильные условия, не используя операцию НЕ:
   а) **not**`(a < 6)`
   б) **not**`(b == c+d)`
   в) **not**`(c != 15)`

*4. Запишите равносильные условия, не используя операцию НЕ:
   а) **not**`(7 < a` **and** `a < 12)`
   б) **not**`(b != c` **or** `d < 5)`

5. Определите порядок выполнения операций при определении истинности условия:
   not`(a > 10)` **or not**`(a < 10)` **and** `(a < b)`
   Определите, истинно или ложно это выражение при `a = 5, b = 10`.
6. Для выражения в предыдущем задании запишите равносильное выражение без использования операции НЕ. После этого расставьте одну пару скобок так, чтобы значение выражения при `a = 5, b = 10` изменилось на обратное.
7. Напишите программу, которая вводит целое число и выводит ответ «да», если оно трёхзначное.
8. Напишите программу, которая получает три числа — рост трёх спортсменов, и выводит сообщение «По росту!», если числа введены по возрастанию, или сообщение «Не по росту!», если они введены в другом порядке.

9. Напишите программу, которая вводит трёхзначное целое число и выводит ответ «да», если все цифры в его записи расположены в порядке возрастания.
10. Напишите программу, которая вводит трёхзначное целое число и выводит ответ «да», если все его цифры чётные.
11. Напишите программу, которая вводит трёхзначное целое число и выводит ответ «да», если все его цифры одинаковые.
12. Напишите программу, которая вводит трёхзначное целое число и выводит ответ «да», если среди его цифр есть хотя бы две одинаковые.
13. Напишите программу, которая вводит трёхзначное целое число и выводит ответ «да», если среди его цифр нет одинаковых.
14. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение `True` (истина), если среди введённых чисел есть хотя бы одно трёхзначное.
15. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение `True` (истина), если все эти числа двузначные.
16. Напишите программу, которая вводит номер месяца и выводит название времени года. При вводе неверного номера месяца должно быть выведено сообщение об ошибке.
17. Напишите программу, которая вводит с клавиатуры номер месяца и определяет, сколько дней в этом месяце. При вводе неверного номера месяца должно быть выведено сообщение об ошибке. Считайте, что год невисокосный.

*18. Напишите программу, которая вводит с клавиатуры номер месяца и день и определяет, сколько дней осталось до Нового года. При вводе неверных данных должно быть выведено сообщение об ошибке.

19. Напишите программу, которая размещает случайным образом две ладьи на шахматной доске и определяет, бьют ли эти ладьи друг друга.
20. Напишите программу, которая размещает случайным образом двух слонов на шахматной доске и определяет, стоят ли они на полях одного цвета.

*21. Напишите программу, которая вводит с клавиатуры координаты двух ферзей на шахматной доске и определяет, бьют ли эти ферзи друг друга.

*22. Напишите программу, которая вводит с клавиатуры координаты двух коней на шахматной доске и определяет, бьют ли эти кони друг друга.

23. Напишите программу, которая определяет, принадлежит ли число x отрезку $[a; b]$. Все числа вещественные, значения x, a и b вводятся с клавиатуры. Разработайте два варианта программы: с использованием вложенных условных операторов и со сложным условием.
24. Напишите программу, которая решает линейное уравнение $a \cdot x = b$. Значения a и b известны (вводятся с клавиатуры), а x нужно найти. Все числа вещественные. Подумайте, зачем в этой задаче нужны ветвления.

§ 10
Циклы с условием

Ключевые слова:

- цикл
- итерация
- счётчик шагов цикла
- зацикливание
- заголовок цикла
- тело цикла
- цикл с предусловием
- алгоритм Евклида

До этого мы писали программы, в которых каждая команда выполнялась только один раз или не выполнялась вообще. Но мощь компьютера состоит ещё и в том, что короткая программа может выполнять очень сложную и длительную обработку данных, применяя некоторые команды многократно.

Цикл — это многократное выполнение одинаковых действий.

Поскольку цикл связан с повторением, циклические алгоритмы называют *итерационными* (от лат. *iteratio* — повторение). Каждое выполнение тела цикла называют *итерацией*.

Как организовать цикл?

Допустим, мы хотим вывести 5 раз на экран слово «привет». Можно, конечно, записать 5 одинаковых команд:

```
print( "привет" )
print( "привет" )
print( "привет" )
print( "привет" )
print( "привет" )
```

Но если нужно будет сделать какие-то действия 1000 или 1 000 000 раз? В этом случае можно организовать цикл. Простейший цикл, нужный нам в этой задаче, мы хотели бы записать так:

```
# сделай 5 раз:
  print( "привет" )
```

К сожалению, в Python (как и во многих других языках программирования) такого цикла нет[1]. Однако можно запрограммировать те же действия немного по-другому. Давайте разберёмся, как можно организовать цикл на языке Python.

Вы знаете, что программа выполняется автоматически. И при этом в любой момент нужно знать, сколько раз уже выполнен цикл и сколько ещё осталось выполнить. Для этого необходимо использовать ячейку памяти (переменную). В ней можно, например, запоминать количество завершённых итераций цикла. Такая переменная целого типа часто называется *счётчиком*.

Сначала в переменную-счётчик записывают ноль (ни одной итерации не сделано), а после каждой итерации цикла увеличивают значение на единицу:

```
count = 0
while count < 5:    # заголовок цикла
  print( "привет" )
  count += 1        # увеличение счётчика
```

В этой программе используется новое служебное слово **while** (в переводе с английского — «пока»), после которого записано условие.

Все операторы, которые выполняются в цикле (они называются *телом цикла*), сдвигаются вправо на одинаковое число позиций, так же как и в условном операторе. Этот приём позволяет обойтись без операторных скобок, ограничивающих тело цикла в других языках программирования.

Нам нужно выполнять цикл 5 раз, т. е. пока счётчик не станет равен 5. Об этом говорит *заголовок цикла*

```
while count < 5:
```

Его можно прочитать как «делай, пока `count < 5`».

После каждой итерации цикла переменная `count` увеличивается на 1 — цикл выполнен ещё один раз. Если программист забудет написать этот оператор, произойдёт *зацикливание*: программа никогда не остановится, потому что условие `count < 5` никогда не станет ложным.

Цикл можно построить и по-другому: сразу записать в счётчик нужное количество итераций, и после каждой итерации цикла *уменьшать* счётчик на 1. Тогда цикл должен закончиться при нулевом значении счётчика:

```
count = 5
while count > 0:    # заголовок цикла
  print( "привет" )
  count -= 1        # уменьшение счётчика
```

Этот вариант несколько лучше, чем предыдущий, поскольку счётчик сравнивается с нулём, а такое сравнение выполняется в процессоре автоматически.

[1] Такой цикл есть, например, в школьном алгоритмическом языке системы КуМир.

Циклы с предусловием

Цикл, в котором проверка условия выполняется при входе (перед выполнением очередной итерации) называется *циклом с предусловием*, т. е. циклом с предварительной проверкой условия. Перед тем как начать выполнение цикла, мы проверяем, нужно ли это делать вообще.

Все циклы, записанные в начале параграфа, — это циклы с предусловием. У них есть два важных свойства:

- условие проверяется при входе в цикл, поэтому тело цикла не выполнится ни разу, если условие в самом начале ложно;
- когда при очередной проверке условия в заголовке цикла выясняется, что это условие ложно, работа цикла заканчивается.

Пример

Рассмотрим ещё одну задачу, которая решается с помощью цикла с условием. Требуется ввести с клавиатуры натуральное число и найти сумму цифр его десятичной записи. Например, если ввели число 123, программа должна вывести сумму 1 + 2 + 3 = 6.

Сначала составим алгоритм решения этой задачи. Предположим, что число записано в переменной `N`. Нам нужно как-то разбить число на отдельные цифры.

Текущую цифру числа будем хранить в переменной `digit`.

Вспомним, что остаток от деления числа на 10 равен последней цифре его десятичной записи. Запишем эту цифру в переменную `digit`:

```
digit = N % 10
```

Сумму цифр будем хранить в целой переменной `summa`. В самом начале, пока ни одну цифру ещё не обработали, значение этой переменной равно нулю:

```
summa = 0
```

Для того чтобы добавить к предыдущей сумме новую цифру, нужно заменить значение переменной `summa` на `summa+digit`, т. е. выполнить присваивание

```
summa += digit
```

Для того чтобы получить следующую цифру числа, надо затем отсечь последнюю цифру числа `N`. Для этого разделим `N` на 10 (основание системы счисления):

```
N = N // 10
```

Эти три операции — выделение последней цифры числа, увеличение суммы и отсечение последней цифры — нужно выполнять несколько раз, пока все цифры не будут обработаны (и отсечены!) и в переменной `N` не останется ноль:

```
N = int( input( "Введите число: " ) )
```

```
summa = 0
while N != 0:
  digit = N % 10
  summa += digit
  N = N // 10
print("Сумма цифр", summa)
```

Выполним трассировку (ручную прокрутку) программы при N = 15. В столбцы таблицы будем записывать изменение значений всех переменных:

Оператор	Условие	N	digit	summa
		15		
sum = 0				0
N != 0	да			
digit = N % 10			5	
sum += digit				5
N = N // 10		1		
N != 0	да			
digit = N % 10			1	
sum += digit				6
N = N // 10		0		
N != 0	нет			

Для введённого числа 15 программа выведет ответ 6 (последнее значение переменной summa).

В отличие от предыдущего примера здесь количество итераций цикла заранее неизвестно, ведь оно определяется введённым числом (точнее — количеством цифр в его десятичной записи).

Докажем, что эта программа не зациклится, т. е. не будет работать бесконечно. Цикл завершается, когда переменная N становится равна нулю, поэтому нужно доказать, что это обязательно случится. По условию заданное число — натуральное, на каждой итерации цикла оно делится на 10 (остаток отбрасывается). В результате после очередного деления оно обязательно станет равно нулю.

Алгоритм Евклида

Алгоритм Евклида — это один из самых древних известных алгоритмов, который используется по сей день. Его автор — греческий математик Евклид — жил в III веке до нашей эры. Алгоритм Евклида позволяет найти наибольший общий делитель (НОД) двух натуральных чисел.

Алгоритм Евклида для натуральных чисел: заменять большее из двух заданных чисел на их разность до тех пор, пока они не станут равны. Полученное число и есть их НОД.

Этот вариант алгоритма работает довольно медленно, если одно из чисел значительно меньше другого, например, для пары чисел 2 и 2014. Значительно быстрее выполняется улучшенный (модифицированный) алгоритм Евклида.

Модифицированный алгоритм Евклида для натуральных чисел: заменять большее из двух заданных чисел на остаток от деления большего на меньшее, пока этот остаток не станет равен нулю. Тогда второе число и есть их НОД.

Мы видим, что здесь тоже нужно выполнять некоторые операции несколько раз, причём сколько раз — заранее неизвестно. Но нас выручит цикл с условием, ведь мы знаем, когда нужно остановиться — когда какое-нибудь из двух чисел станет равно нулю. Значит, нужно выполнять цикл, пока оба значения (одновременно!) ненулевые:

```
while a != 0 and b != 0:
  if a > b:
    a = a % b
  else:
    b = b % a
```

Остаётся вывести результат — ненулевое значение переменной a или b. Можно было бы написать вывод так:

```
if a != 0:
  print( a )
else:
  print( b )
```

Однако можно использовать тот факт, что если одно из двух чисел равно нулю, то их сумма равна второму (ненулевому) числу. Приходим к такому варианту:

```
print( a+b )
```

Количество итераций такого цикла заранее неизвестно и зависит от исходных данных. Дополним программу так, чтобы она считала ещё и количество сделанных итераций цикла. Для этого нужно ввести переменную-счётчик целого типа. Перед началом цикла счётчик обнуляется (в него записывается ноль), и после каждой итерации цикла значение счётчика увеличивается на единицу:

```
count = 0
while a != 0 and b != 0:
  ...
  count += 1
print( a+b )
print( "Шагов:", count )
```

Вместо многоточия нужно вставить условный оператор, как и в предыдущем варианте программы. Обратите внимание, что оператор, увеличивающий значение счётчика, записывается с отступом — он находится в теле цикла.

Обработка потока данных

Рассмотрим такую задачу: на вход программы поступает *поток данных* — последовательность целых чисел, которая заканчивается нулём. Требуется найти сумму элементов этой последовательности.

В этой задаче не нужно сохранять все данные в памяти, мы можем добавлять их к сумме по одному. Будем использовать две целые переменные: в переменной `x` будем хранить последнее введённое число, а в переменной `summa` — накапливать сумму.

Сначала запишем основной цикл программы, «скрыв» шаги алгоритма в комментариях:

```
while x != 0:
  # добавить x к сумме
  # прочитать следующее число
```

Однако перед таким циклом нужно прочитать первое число, иначе неясно, откуда возьмётся значение `x` при первой проверке условия. В итоге получается такая программа:

```
summa = 0
x = int( input() )
while x != 0:
  summa += x
  x = int( input() )
print( "Сумма", summa )
```

Бесконечные циклы

В некоторых задачах используются циклы, условие которых всегда истинно (многоточие обозначает тело цикла):

```
while True:
  ...
```

Поскольку условие `True` всегда истинно, такой цикл никогда не завершится. Но завершить его всё-таки можно, если применить специальный оператор **break**. Например, недостаток предыдущей программы

обработки потока состоит в том, что команда ввода очередного числа в программе записана дважды: перед началом цикла и в конце тела цикла. А можно сделать так:

```
sum = 0
while True:
  x = int(input())
  if x == 0: break  # досрочный выход из цикла
  sum += x
print( "Сумма", sum )
```

Мы заранее (до цикла) ничего не вводим, и организуем цикл по другому. Вводится очередное число и сразу проверяется: если оно равно 0, цикл завершается с помощью оператора **break**. Если это не ноль, число добавляется к сумме и цикл работает дальше.

Выводы

- С помощью циклов в программе можно выполнять повторяющиеся действия.
- Цикл с условием выполняется до тех пор, пока некоторое условие (условие продолжения работы цикла) не станет ложным.
- Если условие в заголовке цикла ложно с самого начала, тело цикла не выполнится ни разу.
- Если условие в заголовке цикла всегда остаётся истинным, цикл работает бесконечно — программа зацикливается.
- Модифицированный алгоритм Евклида для вычисления НОД двух натуральных чисел: заменять большее из чисел на остаток от деления большего на меньшее, пока этот остаток не станет равен нулю. Тогда второе число и есть их НОД.
- Для досрочного выхода из цикла используют оператор **break**.

Вопросы и задания

1. В каком случае программа, содержащая цикл с условием, может зациклиться?
2. В каком случае тело цикла с условием не выполняется ни разу?
3. Программист написал программу с циклом:

```
count = 0
while count < 20:
  print( "привет" )
  count += 1
```

Определите, как изменится работа программы, если:

а) заменить условие на `count != 20`;
б) переставить две строки в теле цикла;
в) заменить условие на `count <= 20`;
г) заменить условие на `count > 20`;

д) программист забудет написать первую строку (запись начального значения счётчика);

е) программист забудет написать строку `count += 1` (увеличение значения счётчика)?

4. Найдите и исправьте ошибку в программе:

```
k = 0
while k < 10:
  print( "привет" )
```

5. Ответьте на вопросы о приведённой в параграфе программе, которая считает сумму цифр числа (ответы обоснуйте).

а) Какова может быть сумма цифр двузначного числа? Трёхзначного? *K*-значного?

б) Сколько раз выполнится цикл, если ввести однозначное число? Двузначное? *K*-значное? Число 0?

в) Как можно обойтись без переменной `digit`?

6. Определите, сколько раз выполнится цикл, и чему будут равны значения переменных `a` и `b` после его завершения.

а)
```
a = 4; b = 6
while a < b: a += 1
```

б)
```
a = 4; b = 6
while a < b: a += b
```

в)
```
a = 4; b = 6
while a > b: a += 1
```

г)
```
a = 4; b = 6
while a < b: b = a - b
```

д)
```
a = 4; b = 6
while a < b: a -= 1
```

7. Что будет выведено на экран в результате работы цикла?

а)
```
k = 1
while k < 5:
  print( k, end="" )
  k += 1
```

б)
```
k = 4
while k < 10:
  print( k*k, end="" )
  k += 2
```

в)
```
k = 12
while k > 3:
print( 2*k-1, end="" )
k -= 1
```

г)
```
k = 10
while k > 2:
  print( k*k, end="" )
  k -= 2
```

д)
```
k = 15
while k > 6:
  print( k-1, end="" )
  k -= 1
```

8. Напишите программу, которая вводит с клавиатуры количество повторений и выводит столько же раз какое-нибудь сообщение.
9. Напишите программу, которая вводит с клавиатуры натуральное число и определяет количество цифр в его десятичной записи.
10. Напишите программу, которая выводит на экран в столбик все цифры числа в обратном порядке (начиная с последней).
*11. Напишите программу, которая выводит на экран в столбик все цифры числа, начиная с первой.
12. Напишите программу, которая вводит с клавиатуры натуральное число и определяет количество чётных цифр в его десятичной записи.
13. Напишите программу, которая вводит с клавиатуры натуральное число и определяет, сколько раз в его десятичной записи встречается цифра 1.
*14. Напишите программу, которая вводит с клавиатуры натуральное число и находит наибольшую цифру в его десятичной записи.
*15. Напишите программу, которая вводит с клавиатуры натуральное число и определяет, есть ли в его десятичной записи две одинаковые цифры, стоящие рядом.
*16. Напишите программу, которая определяет, верно ли, что введённое число состоит из одинаковых цифр (как, например, 222).
17. Напишите программу, которая вводит с клавиатуры натуральное число и вычисляет целую часть квадратного корня из него — наибольшее число, квадрат которого не больше данного числа.
18. Напишите программу, которая предлагает ввести пароль и не переходит к выполнению основной части, пока не введён правильный пароль. Основная часть — вывод на экран «секретных сведений» (придумайте их сами).
*19. Напишите программу, которая вводит с клавиатуры натуральное число и определяет, простое оно или нет. Для этого нужно делить число на все натуральные числа, начиная с 2, пока не получится деление без остатка.
20. Напишите программу, которая вводит с клавиатуры два натуральных числа и находит их НОД с помощью алгоритма Евклида. Программа должна подсчитать количество шагов цикла.
21. Напишите программу, которая вводит с клавиатуры два натуральных числа и сравнивает количество шагов для вычисления их НОД с помощью «обычного» и модифицированного алгоритмов Евклида.

22. Попробуйте найти в Интернете более короткую запись алгоритма Евклида на языке Python.
23. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит количество чисел в потоке.
24. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит произведение чисел в потоке (не считая 0).
25. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит минимальное и максимальное из чисел в потоке (не считая 0).
26. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит среднее арифметическое чисел в потоке (не считая 0).
27. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит сумму положительных чисел в потоке.
28. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит количество чисел в потоке, которые делятся на 3.
29. Поток данных заканчивается вводом числа 0. Напишите программу, которая находит количество чисел в потоке, которые оканчиваются на 5.

Интересный сайт

mccme.ru/free-books/shen/shen-progbook.pdf — книга А. Шеня «Программирование. Теоремы и задачи» (распространяется свободно)

§ 11
Анимация

Ключевые слова:

- анимация
- кадр
- смена кадров
- координаты объекта
- событие
- обработчик события

Принципы анимации

Слово «анимация» произошло от латинского слова *animatio*, что означает «оживление». При анимации быстрая смена мало различающихся рисунков (*кадров*) создаёт иллюзию движения. Если кадры меняются чаще, чем 16 раз в секунду, человеческий глаз не успевает реагировать на каждое изменение и видит плавное движение.

Допустим, мы нарисовали четыре кадра одинакового размера (рис. 1.6) и меняем их на экране, скажем, через каждую секунду: сначала выводим кадр *а*, затем, через 1 секунду — кадр *б*, ещё через 1 секунду — кадр *в* и т. д.

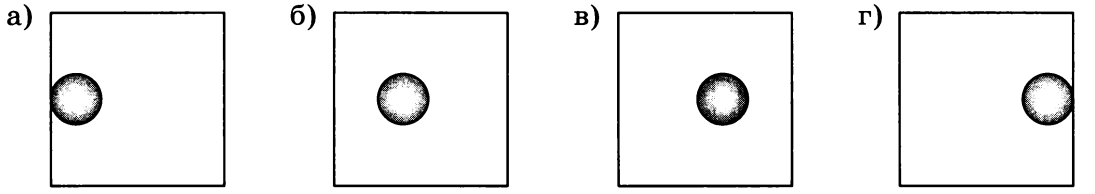


Рис. 1.6

Что мы увидим? Движение шарика слева направо. Конечно, оно не будет плавным, потому что выбран большой интервал времени. Но если менять кадры чаще, чем 16 раз в секунду, глаз перестаёт замечать смену изображений и человеку кажется, что движение непрерывное.

Предположим, что нам нужно изобразить движение объекта (например, шарика) на каком-то фоне. Фон может быть одноцветный или в виде картинки. Для перемещения шарика нам нужно:

1) «стереть» шарик, т. е. восстановить фон в том месте, где сейчас нарисован шарик;
2) изменить положение шарика (его координаты на холсте);
3) запомнить участок фона, который будет испорчен при выводе шарика в новом месте;
4) вывести изображение шарика поверх фона.

Если фон одноцветный, то задача упрощается, потому что не нужно запоминать изображение, перекрытое шариком. Чтобы стереть шарик, достаточно залить это место цветом фона.

В графической библиотеке языка Python все элементы рисунка (линии, прямоугольники, ломаные, круги) — это объекты, которые умеют сами себя перерисовывать. Поэтому нам не нужно заботиться о том, чтобы запоминать и восстанавливать фоновое изображение под объектом, достаточно просто скомандовать «передвинь объект в точку с координатами (x, y)».

Начальное положение

Сначала нарисуем фон — синий квадрат со стороной 400 пикселей:

```
brushColor( "blue" )
rectangle( 0, 0, 400, 400 )
```

Шарик будем изображать в виде круга жёлтого цвета, контур тоже сделаем жёлтым. Базовой точкой, определяющей *текущие координаты* шарика (т. е. координаты в данный момент времени) удобно считать центр круга, обозначим его координаты через `xCenter` и `yCenter`.

Шарик будем двигать слева направо по середине холста на уровне `yCenter = 200` (рис. 1.7).

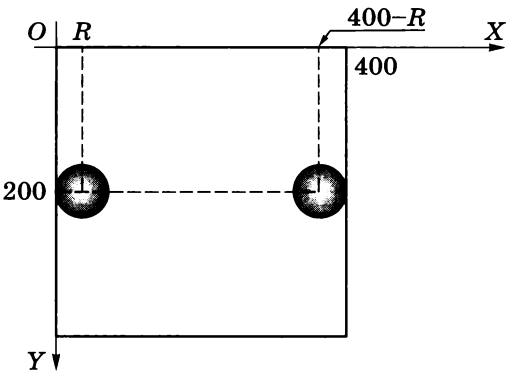


Рис. 1.7

Когда шарик находится вплотную к левой границе, x-координата его центра равна радиусу (обозначим его через R), а у правой границы xCenter = 400 – R.

Теперь можно нарисовать сам шарик в начальном положении:

```
R = 10
xCenter = 10
yCenter = 200
penColor( "yellow" )
brushColor( "yellow" )
obj = circle( xCenter, yCenter, R )
```

Обратите внимание на последнюю строку: мы не только рисуем круг с помощью команды circle, но и записываем в переменную obj результат работы этой функции. А она возвращает специальный код объекта-круга, с помощью которого им можно управлять, например перемещать по холсту.

Анимация движения

Для того чтобы анимация заработала (шарик «поехал» вправо), нам нужно периодически через малый интервал времени увеличивать x-координату шарика, таким образом смещая его в новое место.

Перемещением объекта на экране занимается функция moveObjectBy (от английских слов *move* — переместить, *object* — объект, *by* — предлог, указывающий на величину смещения). Команда

```
moveObjectBy( obj, dx, dy )
```

переместит объект так, чтобы его x-координата увеличилась на dx, а y-координата — на dy. Например, перемещение на 5 пикселей вправо (без изменения y-координаты) задаётся командой

```
moveObjectBy( obj, 5, 0 )
```

Объединим команды, задающие движение шарика, в процедуру с именем update (в переводе с английского — обновить):

```
def update():
  global xCenter    # будем изменять xCenter
  xCenter += 5      # новая x-координата центра
  moveObjectBy( obj, 5, 0 )
```

Обратите внимание, что в процедуре участвуют переменные obj и xCenter, которые введены в основной программе. Такие переменные называются **глобальными**, их значения могут использовать все процедуры программы. Однако тогда, когда мы не просто читаем значение глобальной переменной, а изменяем его, нужно уточнить, что мы будем обращаться именно к глобальной переменной — это делает команда global.

Процедуру update нужно выполнять многократно с небольшим интервалом, например каждые 20 миллисекунд. В графической библиотеке Python есть команда, которая позволяет решить эту задачу:

```
onTimer( update, 20 )
```

Команда onTimer запускает таймер, который будет срабатывать каждые 20 мс, и при каждом срабатывании будет вызываться указанная нами процедура update. Таким образом, говоря на языке программистов, мы установили *обработчик события*.

Событие — это изменение состояния программы или действие пользователя. В нашем случае событие — это срабатывание запущенного нами таймера, а обработчик события — наша процедура update.

Теперь можно запустить программу, и шарик «поедет» вправо. Однако его ничто не останавливает — он будет перемещаться, пока мы не закроем графическое окно (конечно, он уйдёт за пределы холста!).

Давайте остановим шарик на границе синего квадрата. Как определить этот момент? По изменению переменной xCenter: если её значение станет больше, чем 400, это будет означать, что центр шарика вышел за границу области. Учитывая, что радиус шарика равен R, касание границы происходит при xCenter = 400 − R. В этот момент мы можем закрыть графическое окно и остановить работу программы. Добавим такую строку в конец процедуры update:

```
if xCenter >= 400-R: close()
```

Теперь работа программы заканчивается, когда шарик выходит за границу синего квадрата. Вот полная программа:

```
from graph import *

def update():
  global xCenter    # будем изменять xCenter
  xCenter += 5      # новая x-координата центра
```

```
  moveObjectBy( obj, 5, 0 )
  if xCenter >= 400-R: close()

brushColor( "blue" )
rectangle( 0, 0, 400, 400 )

R = 10
xCenter = 10
yCenter = 200
penColor( "yellow" )
brushColor( "yellow" )
obj = circle( xCenter, yCenter, R )

onTimer( update, 20 )

run()
```

Обработка нажатия клавиши

Теперь попробуем остановить движение шарика (и завершить работу программы) при нажатии на клавишу *Escape (Esc)* на клавиатуре.

Нажатие клавиши — это событие, для которого можно задать свой обработчик (так же, как и для срабатывания таймера):

```
onKey( keyPressed )
```

Эту команду нужно поместить в основную программу, сразу после (или перед) командой `onTimer`. Обработчик, установленный командой `onKey`, срабатывает тогда, когда нажата какая-то (любая!) клавиша на клавиатуре. При этом вызывается процедура, которую мы указали — `keyPressed`. Этой процедуры ещё нет, нам предстоит её написать. Например, она может выглядеть так:

```
def keyPressed( event ):
  if event.keycode == VK_ESCAPE:
    close()
```

Когда вызывается эта процедура, она получает специальный блок данных `event` с информацией о том событии, которое произошло. Этот блок объединяет несколько переменных (полей), в которых записаны данные. Этот блок объединяет несколько переменных (полей), в которых записаны данные. Такие блоки программисты называют *структурами* или *записями* (англ. *record*).

В одном из полей структуры `event` записан код нажатой клавиши, прочитать который можно с помощью точечной записи: `event.keycode`. Это означает «поле `keycode` блока данных `event`». Мы читаем значение этого поля и, если оно совпадает с кодом клавиши *Escape* (он обозначается как `VK_ESCAPE`), завершаем работу программы.

Выводы

- Анимация на компьютере — это быстрая смена рисунков, которая создаёт иллюзию движения. Каждый из таких рисунков называют кадром.
- Если менять кадры чаще, чем 16 раз в секунду, глаз перестаёт замечать смену изображений и человеку кажется, что он видит непрерывное движение.
- В графической библиотеке языка Python все элементы рисунка — это объекты, которые умеют сами себя перерисовывать. Поэтому анимация сводится к постепенному перемещению этих объектов.
- Переменные, которые введены в основной программе, можно использовать во всех процедурах. Они называются глобальными.
- Глобальные переменные, которые нужно изменять в процедуре, необходимо перечислить в команде `global`.
- Для создания анимации нужно установить обработчик события «таймер»: указать процедуру, которая будет вызываться при каждом срабатывании таймера.
- Если программа должна реагировать на нажатие клавиш, нужно установить обработчик события «клавиша нажата». Внутри этой процедуры определяется код нажатой клавиши и выполняются необходимые действия.

Вопросы и задания

1. Как вы думаете, можно ли заранее нарисовать все кадры анимации для мультфильма? В компьютерных играх? Обоснуйте свой ответ.
2. Как можно изменить программу анимации, чтобы ускорить движение шарика? Предложите два варианта решения этой задачи и обсудите ограничения каждого из них.
3. Измените программу анимации так, чтобы после касания шариком правой границы синего квадрата он останавливался, но окно не закрывалось.
4. Доработайте программу анимации так, чтобы шарик после выхода за границы синего квадрата вновь появлялся с другой стороны холста.
5. Выполните рефакторинг (переработку кода) программы анимации, избавившись от «магических чисел» (5, 400): запишите их в глобальные переменные в самом начале основной программы и далее везде используйте эти переменные.
6. Напишите программу, в которой через поле (синий прямоугольник) движутся два квадрата навстречу друг другу.

*7. Доработайте предыдущую программу так, чтобы квадраты отталкивались от границ поля и начинали движение в противоположном направлении.

*8. Напишите программу, в которой шарик управляется с помощью клавиш-стрелок (их коды обозначаются как VK_LEFT, VK_RIGHT, VK_UP и VK_DOWN). Шарик не должен выходить за границы поля.

*9. Напишите программу, в которой шарик движется непрерывно и меняет направление при нажатии на клавиши-стрелки.

*10. Доработайте программу из предыдущего задания так, чтобы шарик отталкивался от границ поля.

Интересный сайт

pygame.org — библиотека PyGame для программирования игр на языке Python

§ 12
Циклы по переменной

Ключевые слова:

- цикл по переменной
- переменная цикла
- шаг изменения переменной цикла

Сделать *N* раз

Вернёмся снова к задаче, которую мы обсуждали в одном из параграфов — вывести на экран несколько раз слово «привет». Фактически нам нужно организовать цикл, в котором блок операторов выполнится заданное число раз. Для этого можно применить ещё один вид цикла — *цикл по переменной* (или *цикл с параметром*). На языке Python он записывается так:

```
for i in range(10):
  print( "привет" )
```

Здесь слово for означает «для», переменная i (её называют *переменной цикла*) изменяется в диапазоне (in range) от 0 до 10, не включая 10 (т. е. от 0 до 9 включительно). Таким образом, цикл выполняется для i = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 — ровно 10 раз. Переменная i — это счётчик выполненных итераций цикла. Можно было записать этот цикл и по-другому:

```
for i in [0,1,2,3,4,5,6,7,8,9]:
  print( "привет" )
```

В квадратных скобках через запятую перечислены все значения переменной, при которых выполняется цикл. Если их много, такой способ неудобен, лучше использовать встроенную функцию range.

Обратите внимание, что последовательность, которую строит функция range, не бесконечна, т. е. цикл по переменной всегда заканчивается, программа не может зациклиться.

От цикла while к циклу for

Рассмотрим ещё один пример. В информатике важную роль играют степени числа 2 (2, 4, 8, 16 и т. д.). Давайте выведем на экран все степени двойки от 2^1 до 2^{10}. Для решения этой задачи мы можем написать программу, использующую цикл с условием:

```
power = 1
N = 2
while  power <= 10:
  print(N)
  N *= 2
  power += 1
```

Вы, наверно, заметили, что переменная power используется трижды (см. выделенные блоки): в операторе присваивания начального значения, в условии выполнения цикла и в теле цикла (увеличение на 1).

Чтобы собрать все действия с ней в один оператор, применим цикл по переменной. Нам нужно выполнить тело цикла при power = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Чтобы получить такой набор значений, нужно вызвать функцию range с двумя аргументами: первый — это начальное значение (1), а второй — ограничитель, не входящий в последовательность (11):

```
N = 2
for power in range(1,11):
  print(N)
  N *= 2
```

Запись цикла получилась проще, и поэтому меньше шансов сделать ошибку.

Однако не любой цикл с условием может быть переписан как цикл по переменной. Если количество повторения цикла неизвестно и не может быть найдено заранее (как в задаче с вычислением суммы цифр числа), цикл по переменной использовать не удаётся.

Вместе с тем, любой цикл по переменной можно заменить на равносильный цикл с условием: вместо вызова функции range придётся задать отдельно начальное значение переменной цикла, условие продолжения цикла и правило изменения переменной цикла.

Рассмотрим ещё одну задачу — найдём сумму всех натуральных чисел от 1 до 1000. Для накопления суммы будем использовать переменную (назовём её summa). В цикле другая переменная (скажем, i) изменяется от 1 до 1000, и на каждом шаге этого цикла к сумме добавляется очередное число:

```
summa = 0
for i in range(1,1001):
  summa += i
```

Шаг изменения переменной цикла

По умолчанию функция range строит последовательность, в которой каждое следующее число на 1 больше предыдущего. Но это правило можно изменить, если при вызове функции range указать третий аргумент — шаг изменения переменной цикла. Следующая программа выводит квадраты натуральных чисел от 10 до 1 в порядке убывания:

```
for k in range(10,0,-1):
  print(k*k)
```

В этом примере шаг равен –1, т. е. каждое следующее число на 1 меньше предыдущего. Заметим, что конечное значение 0 не входит в последовательность.

Пусть, например, нам нужно перебрать в цикле все значения переменной i от 0 до 100, кратные пяти: 0, 5, 10, ..., 100. Для этого нужно взять шаг изменения переменной, равный 5:

```
for i in range(0, 101, 5):
  ... # что-то делать с i
```

Второй аргумент функции range равен 101 для того, чтобы последнее значение переменной i было равно 100. Значение-ограничитель должно быть больше, чем 100 (чтобы число 100 появилось в последовательности), но меньше, чем 106 (чтобы следующее число, 105, не появилось).

Выводы

- Цикл по переменной применяют тогда, когда количество итераций цикла заранее известно или может быть вычислено до начала цикла.
- Переменная, изменение которой определяет работу цикла, называется переменной цикла.
- Цикл по переменной всегда заканчивается (программа не может зациклиться).
- Цикл с условием, для которого количество итераций известно или может быть вычислено заранее, можно заменить на равносильный цикл по переменной.
- Любой цикл по переменной можно заменить на цикл с условием, который выполняет те же действия.
- При вызове функции range указывают начальное значение, значение-ограничитель (не входящее в последовательность) и шаг изменения переменной цикла.

Вопросы и задания

1. В каких случаях цикл по переменной не выполнится ни разу?
2. Может ли цикл по переменной работать бесконечно? Почему?
3. Сравните цикл по переменной и цикл с условием. Какие преимущества и недостатки есть у каждого из них?

4. Верно ли, что любой цикл по переменной можно заменить циклом с условием? Верно ли обратное утверждение?
5. Напишите программу, которая выводит на экран чётные степени числа 2, от 2^{10} до 2^2, в порядке убывания.
6. Определите, сколько раз выполнится цикл и чему будет равно значение переменной a после его завершения.

а)
```
a = 1
for i in range(3):
  a += 1
```

б)
```
a = 1
for i in range(3,1):
  a += 1
```

в)
```
a = 1
for i in range(1,3,-1):
  a += i
```

г)
```
a = 1
for i in range(3,1,-1):
  a += i
```

7. Что будет выведено на экран в результате работы следующего цикла?

а)
```
k = 1
for i in range(5):
  print( i, end="" )
```

б)
```
k = 1
for i in range(5):
  print( i+k, end="" )
```

в)
```
k = 1
for i in range(5):
  print( k*k, end="" )
  k += 2
```

г)
```
k = 8
for i in range(5,0,-1):
  print( i, end="" )
  k -= 2
```

д)
```
k = 8
for i in range(5,0,-1):
  print( 2*i-k, end="" )
  k -= 2
```

8. Напишите программу, которая вводит с клавиатуры значения a и b ($a \le b$) и выводит все целые числа от a до b в порядке возрастания.
9. Напишите программу, которая вводит с клавиатуры значения a и b ($a \le b$) и выводит квадраты всех целых чисел от a до b в порядке возрастания.
10. Напишите программу, которая вводит с клавиатуры натуральное число и определяет, простое оно или нет.

11. Напишите программу, которая вводит с клавиатуры два целых числа и вычисляет их произведение, используя только операции сложения. Учтите, что числа могут быть отрицательными.
12. Ипполит задумал трёхзначное число, которое при делении на 15 даёт в остатке 11, а при делении на 11 даёт в остатке 9. Напишите программу, которая находит все такие числа.
13. С клавиатуры вводится натуральное число N. Программа должна найти факториал этого числа (обозначается как $N!$) – произведение всех натуральных чисел от 1 до N. Например:
$$5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120.$$
14. Напишите программу, которая вводит с клавиатуры натуральное число N и находит сумму всех натуральных чисел от 1 до N. В одном варианте программы используйте цикл с условием, в другом — цикл по переменной.
15. Напишите программу, которая вводит с клавиатуры натуральное число N и выводит первые N чётных натуральных чисел.
16. Напишите программу, которая вводит с клавиатуры натуральные числа a и b ($a \le b$), и выводит сумму квадратов всех натуральных чисел на отрезке $[a; b]$.
17. Напишите программу, которая вводит с клавиатуры натуральное число N и выводит на экран N случайных целых чисел. Запустите её несколько раз, объясните результаты опыта.

*18. Напишите программу, которая строит последовательность из N случайных вещественных чисел на полуинтервале [0; 1) и определяет, сколько из них попадает в полуинтервалы [0; 0,25), [0,25; 0,5), [0,5; 0,75) и [0,75; 1). Сравните результаты, полученные при N = 10, 100, 1000, 10000.

19. *Автоморфные числа.* Натуральное число называется автоморфным, если его запись — это последние цифры его квадрата. Например: $25^2 = 625$. Напишите программу, которая вводит с клавиатуры натуральное число N и выводит на экран все автоморфные числа, не превосходящие N.
20. Напишите программу, которая вводит с клавиатуры натуральные числа A и N и вычисляет A^N без использования операции возведения в степень.
21. Напишите программу, которая вводит с клавиатуры натуральное число N и определяет сумму всех его делителей, меньших самого числа. Например, для числа 8 эта сумма равна 1 + 2 + 4 = 7.

*22. В магазине продаётся мастика в ящиках по 15 кг, 17 кг, 21 кг. Напишите программу, которая определяет:
   1) как купить ровно 185 кг мастики, не вскрывая ящики;
   2) сколькими способами можно это сделать?

23. Тройкой пифагоровых чисел называются натуральные числа (a, b, c), для которых выполняется равенство $a^2 + b^2 = c^2$. Напишите программу, которая находит все тройки пифагоровых чисел, в которых каждое число не превышает значения N, введённого с клавиатуры.

24. Напишите программу, которая вводит натуральное число N и находит все тройки натуральных чисел (a, b, c), для которых выполняется равенство $N = a^2 + b^2 + c^2$. Учтите, что таких троек может не быть.
25. Напишите программу, которая в последовательности натуральных чисел определяет количество чисел, кратных 6. Программа получает на вход количество чисел в последовательности, а затем — сами числа.
26. Напишите программу, которая в последовательности натуральных чисел определяет количество чисел, которые оканчиваются на 5. Программа получает на вход количество чисел в последовательности, а затем — сами числа.
27. Напишите программу, которая в последовательности натуральных чисел определяет количество чисел, которые оканчиваются на 1 и делятся на 3. Программа получает на вход количество чисел в последовательности, а затем — сами числа.
28. Напишите программу, которая в последовательности натуральных чисел определяет количество двузначных чисел, которые оканчиваются на 3 и делятся на 7. Программа получает на вход количество чисел в последовательности, а затем — сами числа.
29. Напишите программу, которая моделирует работу следующего автомата. Автомат получает на вход четырёхзначное натуральное число и строит новое число следующим образом:
    1) вычисляются суммы первой и второй; второй и третьей; третьей и четвёртой цифр;
    2) из полученных сумм отбрасывается наибольшая;
    3) остальные записываются в порядке невозрастания.

    Например, для числа 1284 получаем суммы: 1 + 2 = 3; 2 + 8 = 10; 8 + 4 = 12. Наибольшая сумма 12 отбрасывается, результат 103. Ваша программа должна вводить с клавиатуры желаемый результат работы автомата и выводить все четырёхзначные числа, при обработке которых автомат выдаст этот результат.

§ 13
Циклы в компьютерной графике

Ключевые слова:

- цикл
- узор
- вложенный цикл
- процедура

Узоры

Узор — это рисунок, основанный на повторении одинаковых элементов. В этом параграфе мы будем применять циклы для построения разнообразных узоров.

Задача 1

Построим ряд из пяти окружностей радиуса 5 пикселей (рис. 1.8).

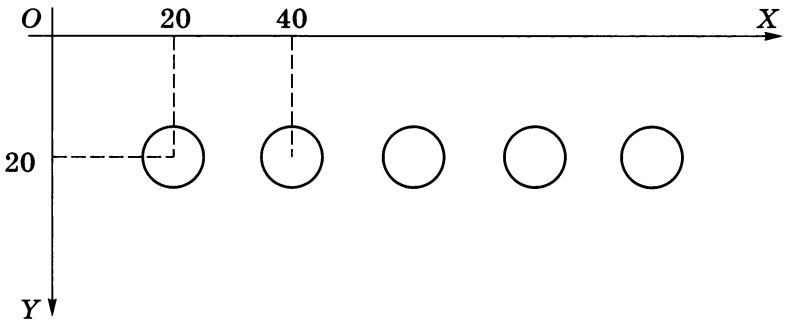


Рис. 1.8

Их можно нарисовать с помощью отдельных вызовов команды `circle`:

```
circle( 20, 20, 5 )
circle( 40, 20, 5 )
circle( 60, 20, 5 )
circle( 80, 20, 5 )
circle( 100, 20, 5 )
```

Можно заметить, что у окружностей на рис. 1.8 изменяется только x-координата центра. Её можно сделать переменной и назвать `x`. Эта переменная будет меняться от 20 до 100 с шагом 20, поэтому можно использовать такой цикл по переменной:

```
for x in range(20,101,20):
  circle( x, 20, 5 )
```

Поскольку значение-ограничитель диапазона — второе число, переданное функции `range`, — не включается в последовательность, оно должно быть больше, чем 100, но меньше, чем 121 (число 100 должно войти в последовательность, а число 120 — не должно).

Вложенные циклы

Задача 2

Построим три одинаковых ряда окружностей, расположенных на расстоянии 20 пикселей по высоте друг от друга (рис. 1.9).

Три горизонтальных ряда на рис. 1.9 различаются лишь координатой y. Поэтому можно применить такой цикл:

```
for y in range(20,61,20):
  # построить ряд с выбранным y
```

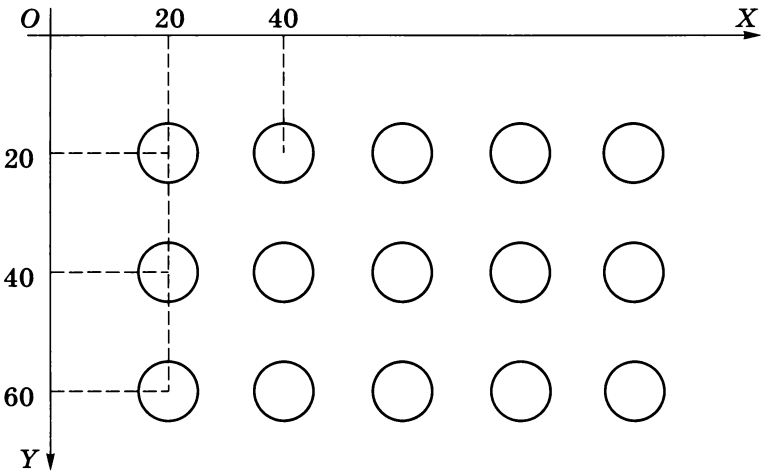


Рис. 1.9

Вспомним, что в начале параграфа мы уже решали такую задачу: строили такой же ряд окружностей для `y = 20`. Теперь заменяем в предыдущем решении 20 на `y` и записываем это решение в тело цикла вместо комментария:

```
for y in range(20,61,20):
  for x in range(20,101,20):
    circle( x, y, 5 )
```

Поскольку построение одного ряда окружностей — это тоже цикл, мы получили цикл внутри другого цикла, т. е. *вложенный цикл*.

Вложенный цикл — это цикл, находящийся внутри другого цикла.

Рефакторинг

Оформим рисование одного ряда окружностей в виде процедуры `Row`, которая принимает один параметр — *y*-координату центра окружностей:

```
def Row(y):
  for x in range(20,101,20):
    circle( x, y, 5 )
```

Тогда основная программа — это цикл с вызовом процедуры `Row`:

```
for y in range(20,61,20):
  Row( y )
```

Можно и дальше улучшать процедуру `Row`, делая её более универсальной, например, передавать ей радиус окружностей, начальную координату, шаг между окружностями и т. д. Но нужно помнить, что процедура с большим количеством параметров (больше, чем 3–4) становится менее понятной.

Пример

Задача 3

Построим на экране узор из ромбов (рис. 1.10).

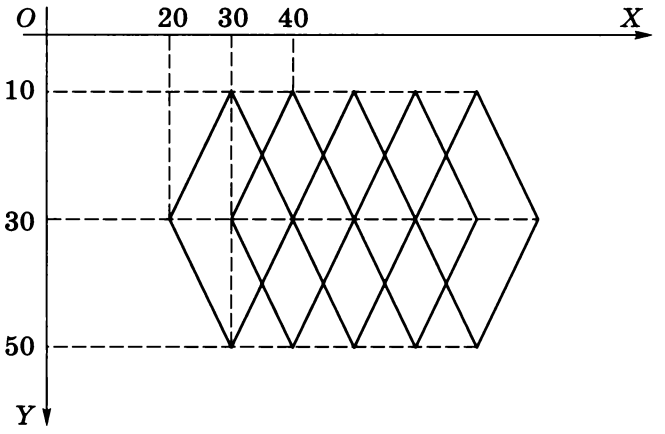


Рис. 1.10

По рисунку видно, что узор состоит из пяти одинаковых ромбов. Мы оформим алгоритм рисования ромба в виде процедуры, а затем будем вызывать её несколько раз.

Выберем за базовую точку левый угол ромба, его координаты обозначим через (x, y). Напишем процедуру `Romb`, которая рисует ромб заданного размера, левый угол которого находится в точке с заданными координатами (x, y). По рисунку определяем, что углы такого ромба расположены в точках (x, y), (x+10, y–20), (x+20, y) и (x+10, y+20). Эти точки нужно соединить ломаной — вызвать функцию `polygon`:

```
def Romb( x, y ):
  polygon( [(x,y), (x+10,y-20), (x+20,y),
            (x+10,y+20), (x,y)] )
```

Хотя y-координата у всех ромбов одинаковая, мы включили её в список параметров процедуры, чтобы сделать процедуру более универсальной.

По рисунку видим, что x-координаты левых углов ромбов образуют последовательность 20, 30, 40, 50, 60:

```
Romb( 20, 30 )
Romb( 30, 30 )
Romb( 40, 30 )
Romb( 50, 30 )
Romb( 60, 30 )
```

Тогда основную программу можно написать в виде цикла по переменной `x`:

```
for x in range(20, 61, 10):
  Romb( x, 30 )
```

Штриховка

Во многих задачах компьютерной графики нужно заштриховать какие-то области. Например, штриховкой обозначаются сечения на чертежах, болота на картах местности.

Штриховка строится из параллельных линий, которые удобно рисовать в цикле. Выполним вертикальную штриховку прямоугольника, разделив его на N полос (рис. 1.11).

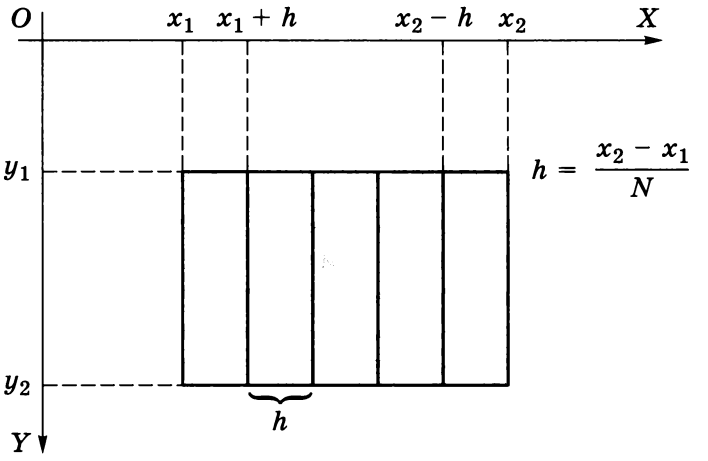


Рис. 1.11

Решим задачу в общем виде для любого прямоугольника. Будем считать, что его верхний левый угол находится в точке с координатами (x_1, y_1), а правый нижний — в точке с координатами (x_2, y_2). Сначала нарисуем контур прямоугольника:

```
x1 = 100; x2 = 300
y1 = 100; y2 = 200
rectangle(x1, y1, x2, y2)
```

Конечно, вы можете выбрать и другие координаты углов.

В результате штриховки нужно разделить прямоугольник на N полос, так что шаг штриховки (расстояние между соседними линиями) вычисляется по формуле

$$h = \frac{x_2 - x_1}{N}.$$

Координаты концов отрезка при вызове команды `line` (рисование линии) должны быть целыми числами, поэтому мы будем использовать только целые значения шага (в пикселях). Поэтому величина `h` округляется до ближайшего целого значения:

```
h = round((x2-x1)/N)
```

Теперь посмотрим на рис. 1.11. Первую слева линию штриховки можно нарисовать так:

```
line( x1+h, y1, x1+h, y2 )
```

Вторая смещена на h вправо по оси x, т. е. обе x-координаты увеличиваются на h:

```
line( x1+2*h, y1, x1+2*h, y2 )
```

Видим, что y-координаты обоих концов отрезка не изменяются, а x-координата меняется с шагом h, так что можно написать цикл по переменной x:

```
for x in range(x1+h, x2, h):
  line( x, y1, x, y2 )
```

Штриховка: второй вариант

Изучение работы программы рисования штриховки при различных значениях N (например, при N = 6 или N = 9 для наших данных) показывает, что последняя полоса может оказаться больше или меньше, чем остальные. Это происходит из-за того, что значение h округлено, и при увеличении x в цикле с каждым шагом эта ошибка накапливается.

Для того чтобы улучшить результат, можно не округлять значение шага h при вычислении, а изменять x как вещественную переменную, округляя её значение только при передаче функции line.

Однако при этом нельзя использовать функцию range, которая работает только с последовательностями целых чисел. Придётся применить цикл с условием: вручную установить начальное значение x и вручную изменять его, увеличивая на h в конце каждого повторения цикла:

```
h = (x2-x1) / N        # без округления
x = x1 + h
while x < x2:
  line( round(x), y1, round(x), y2 )
  x += h
```

Ещё раз обратим внимание на то, что переменные x и h в этой программе — вещественные, но при передаче функции line значение x округляется до ближайшего целого с помощью встроенной функции round.

Выводы

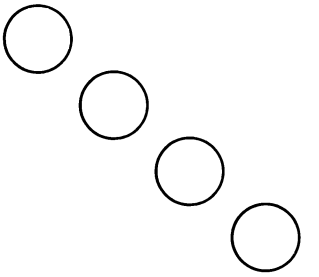
- При выполнении узоров (повторяющихся рисунков) и штриховки удобно использовать цикл по переменной.
- Для того чтобы легче было построить цикл, можно записать команды для рисования нескольких элементов узора, в том числе первого и последнего. После этого определить, какие величины

изменяются и как именно (чему равны начальное и конечное значения переменной цикла, шаг).

- Штриховка обычно строится из параллельных линий, расположенных на одинаковом расстоянии друг от друга.
- Для вычисления шага штриховки нужно разделить длину отрезка, который разбивается на равные части, на количество полос.
- При вычислении координат отрезков штриховки лучше использовать вещественные числа, округляя координаты при вызове функции рисования линий.

Вопросы и задания

1. Что нужно изменить в решении задачи 1, чтобы окружности выстроились по диагонали?

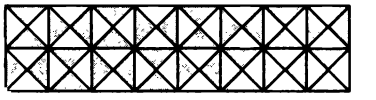


Нужен ли для этого вложенный цикл?

2. Сравните два варианта решения задачи 2:
   а) вызов процедуры в цикле;
   б) вложенные циклы.

   Какой из них вам больше нравится? Почему?
3. Сравните разные варианты выбора базовой точки для рисования ромба в задаче 3 из параграфа. Какой из них лучше? Как вы рассуждали?
4. Постройте изображение шахматной доски размером 8×8 клеток.
5. Постройте узоры (см. цветные рисунки на обороте обложки).

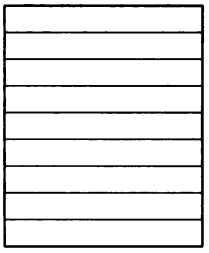
а) б) в)

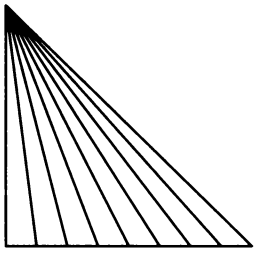


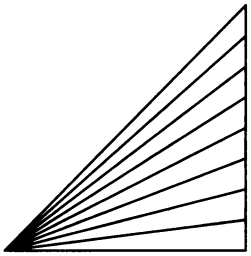
г) д) е)

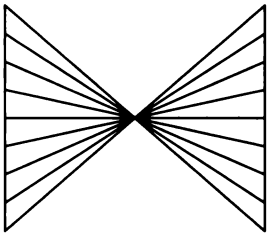
ж)

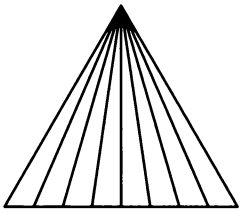
6. Напишите программу, которая выполняет штриховку прямоугольника горизонтальными линиями.
7. Постройте следующие рисунки (число линий храните в переменной N).

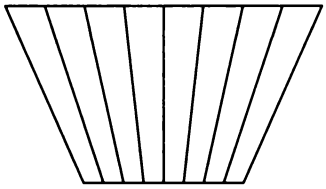
а) 

б) 

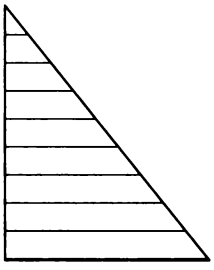
в) 

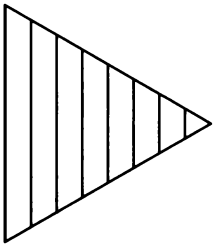
г) 

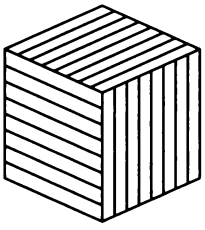
д) 

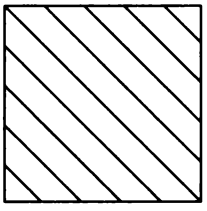
е) 

*8. Выполните штриховку (число линий храните в переменной N):

а) 

б) 

в) 

г) 

**9. Напишите программу, которая выполняет штриховку круга линиями с наклоном 45 градусов.

Глава 2
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C++

§ 14
Первые программы

Ключевые слова:

- компилятор
- препроцессор
- раздельная компиляция
- вывод данных
- поток вывода
- поток ввода
- пространство имён
- символьная строка

Язык C++

В начале 1970-х годов Деннис Ритчи, сотрудник компании Bell Labs, разработал язык C, который в последующие годы стал одним из самых востребованных языков программирования и остаётся на лидирующих позициях до настоящего времени[1].

С одной стороны, язык C — это язык высокого уровня, независимый от процессора, на котором выполняется программа. С другой стороны, он позволяет напрямую работать с аппаратурой, заменяя чрезвычайно сложное программирование в машинных кодах и на языке ассемблера.

В начале 1980-х годов Бьёрн Страуструп, сотрудник той же компании Bell Labs, дополнил язык C новыми возможностями, которые были нужны ему для моделирования сложных систем. Так был создан язык C++, который стал одним из самых популярных языков программирования. Отметим, что почти все программы на «чистом» C будут работать и в C++.

Одно из достоинств современного языка C++ — обширная стандартная библиотека: набор готовых подпрограмм для решения часто встречающихся задач (например, для обработки символьных строк).

На языках C и C++ написано большинство современных операционных систем и программ, которыми вы пользуетесь ежедневно, в том числе абсолютное большинство компьютерных игр. Синтаксис C и C++ (правила написания команд) использовали при создании языков C#, Objective-C, Java, JavaScript, Swift и других, поэтому, изучив C++, вы легко можете перейти на эти языки при необходимости.

Перед запуском программа на C++ обрабатывается *компилятором* — специальным видом транслятора, который строит исполняемую

[1] В рейтинге популярности языков программирования от компании TIOBE в январе 2018 года язык C занимал второе место после Java (**https://www.tiobe.com/tiobe-index/**).

программу (*приложение*) в машинных кодах для конкретного процессора и операционной системы. Чтобы потом запускать такую программу, транслятор уже не нужен (в отличие от программ на языке Python).

Существует много компиляторов C++, самые известные из них — Microsoft Visual C++, Intel C++ Compiler, GCC g++, Clang. Последние два компилятора относятся к свободному программному обеспечению, и их можно использовать бесплатно.

Благодаря компиляции, программа на C++ работает значительно (в 100–150 раз) быстрее, чем аналогичная программа на языке Python.

Файлы с программами на C++ обычно имеют расширение `.cpp`, такие файлы называются исходными кодами или на жаргоне — «исходниками». Компилятор в результате обработки исходных файлов строит для каждого из них *объектный файл*, в котором записаны машинные команды — *исполняемый код*. Объектные файлы обычно имеют расширения `.o` или `.obj`. Затем другая программа — *сборщик*, или *редактор связей* (англ. *linker*), — собирает все объектные файлы вместе в один исполняемый файл, добавляя все используемые функции из библиотек.

Такой подход называется раздельной компиляцией. Если, например, программист внёс изменения только в один исходный файл, компилятор должен обработать (*компилировать*) заново только этот файл, а потом остаётся собрать новую версию программы с помощью сборщика.

В этой главе мы познакомимся с простыми возможностями C++, используя теоретические знания, полученные в предыдущей главе при изучении языка Python.

Самая простая программа

Пустая программа — это программа, которая ничего не делает, но удовлетворяет требованиям языка C++. Она полезна, прежде всего, для того, чтобы понять общую структуру программы.

```
int main()
  {
  // это основная программа
  /* здесь записывают
     операторы  */
  }
```

В отличие от языка Python основная программа всегда оформляется как отдельный блок — *функция*. Главная функция всегда имеет имя `main` (от англ. *main* — основной, главный). Пустые круглые скобки после имени функции означают, что у неё нет параметров, т. е. она не принимает никаких дополнительных данных от пользователя[1)].

Слово **`int`** перед именем `main` говорит о том, что программа в конце работы передаёт операционной системе результат своей работы —

[1)] Вообще говоря, параметры у основной программы могут быть, но пока мы не будем их использовать.

целое число (от английского *integer* — целый). По этому числу операционная система определяет, завершилась ли программа успешно или при её выполнении были ошибки.

Тело (основная часть) программы ограничено фигурными скобками. Между ними записывают операторы — команды языка программирования. Часть программы, ограниченная фигурными скобками, называется *блоком*.

Всё, что следует после символов // до конца строки, это *комментарии* — пояснения, которые не обрабатываются компилятором. Комментарий можно ограничивать парами символов /* и */, в этом случае комментарий может занимать несколько строк[1].

Вывод текста на экран

Программа, которая выводит на экран текст, выглядит так:

```
#include <iostream>
using namespace std;
int main()
  {
  cout << "I am a program!";
  cin.get();
  }
```

Перед заголовком основной программы добавилась строка, начинающаяся с символов **#include** (от англ. *include* — включить). Это команда для *препроцессора* — специальной программы, которая обрабатывает текст нашей программы до того, как за него «примется» компилятор. С помощью команд **#include** к программе подключаются стандартные библиотеки языка программирования, а также разные дополнительные библиотеки.

К нашей программе подключена библиотека `iostream` (от англ. *input–output streams* — потоки ввода и вывода). *Поток вывода* — это последовательность символов на выходе программы[2]. Стандартный поток вывода в языке C++ называется `cout` (от англ. *character output stream* — поток вывода символов). *Поток ввода* — последовательность символов, вводимая с клавиатуры, — имеет имя `cin` (от англ. *character input stream* — поток ввода символов).

При выводе в выходной поток справа от оператора << записывают данные, которые нужно вывести на экран; в нашей программе — это символьная строка в кавычках[3].

---

1) Эти символы удобно набирать на цифровой части клавиатуры.

2) Строго говоря, поток в C++ — это специальный объект, который занимается вводом или выводом данных. Но пока можно думать о нём как о последовательности символов на входе или выходе программы.

3) Конечно, можно выводить и текст на русском языке, но при этом могут возникнуть проблемы с кодировками, и мы пока не будем этого делать.

Каждый оператор в C++ заканчивается (обязательно — в отличие от языка Python) точкой с запятой.

Можно записывать несколько операторов << в одной команде, например:

```
cout << "Hello, " << "Dad!";
```

Для перехода на новую строку используют специальную команду endl:

```
cout << "Hi!" << endl << "Who are you?";
```

Одиночный символ можно заключать в апострофы («одиночные кавычки»):

```
cout << "Hello" << '!' << endl;
```

Последняя команда cin.get() в программе на C++ обеспечивает задержку до нажатия на клавишу *Enter*. Если этого не сделать, в некоторых системах окно программы закроется сразу после вывода текста, и мы не успеем увидеть результат. Здесь cin — это входной поток. С помощью оператора «точка» вызывается команда (метод) get для этого потока, которая ждёт ввода одного символа. Но этот символ никуда не записывается, так что эта команда означает «получить символ из входного потока и ничего с ним не делать». Мы используем cin.get() только для организации паузы[1)], поэтому в следующие программы эту строку включать не будем.

Строка

```
using namespace std;
```

говорит о том, что будет использоваться пространство имён std, в котором определена стандартная библиотека языка C++. *Пространство имён* — это область видимости объектов программы, в том числе переменных. С помощью пространств имён программисты отделяют друг от друга разные библиотеки, чтобы имена переменных и функций в них не пересекались и не путались.

В области std описаны стандартные потоки ввода и вывода с именами cin и cout. Если мы не объявили бы используемое пространство имён, то нужно было бы явно указывать его при любом обращении к этим потокам, вот так:

```
#include <iostream>
int main()
  {
  std::cout << "Привет!";
  std::cin.get();
  }
```

[1)] В операционной системе Windows можно вместо этого использовать команду system("pause");

Можно с помощью команды using объявить только те объекты из пространства имён, которые будут использоваться в программе:

```
using std::cin;
using std::cout;
```

Так обычно делают в больших профессиональных программах, в которых подключается большое количество библиотек.

Выводы

- Любая программа на языке C++ содержит основную программу — функцию с именем main.
- Символами // начинается комментарий, который действует до конца строки. Комментариями считается также всё, что записано между парами символов /* и */.
- Каждый оператор заканчивается точкой с запятой.
- Символьная строка — это последовательность символов, заключённая в кавычки. Апострофы используются для записи одиночных символов (не строк).
- Стандартные (и не только стандартные) библиотеки подключаются к программе с помощью команды #include. Её выполняет препроцессор, обрабатывающий текст программы перед компилятором.
- Поток — это последовательность символов, которая поступает на вход или на выход программы. Для вывода данных на экран используют выходной поток cout, для ввода с клавиатуры — поток cin.
- Пространство имён — это область видимости объектов программы, в том числе переменных. Потоки ввода и вывода принадлежат пространству имён std.

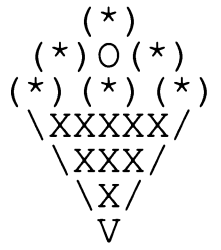
Вопросы и задания

1. Почему раздельная компиляция позволяет экономить время программиста при отладке программы?
2. Найдите ошибки в записи операторов вывода:
```
cout << "Смотрите фильм "Салют-7"!";
cout << "C" "++";
cout << 'Hello, ' << "world!";
```
Как их можно исправить?
3. Найдите ошибки в программе.
```
cout << "Привет," <<  Вася!;
```
4. Заполните пропуск так, чтобы программа вывела на экран слова
```
Computer
science
```
(каждое слово в отдельной строке):
```
cout << "Comp" << ...;
cout << "nce";
```

5. Напишите программу, которая выводит на экран фразу «лесенкой» (автор — В. В. Маяковский):

```
Я хотел бы
        жить
           и умереть в Париже,
если б не было
            такой земли —
                       Москва...
```

6. Напишите программу, которая рисует вкусное мороженое:



Интересные сайты

informatics.mccme.ru — сайт для подготовки к олимпиадам по информатике с автоматической проверкой решений
cppstudio.com — программирование на C++ для начинающих
www.cplusplus.com — сайт, посвящённый языку C++
cpp.sh — онлайн-компилятор C++
ideone.com — среда для онлайн-разработки и отладки программ на разных языках

§ 15
Диалоговые программы

Ключевые слова:

- ввод данных
- переменная
- объявление переменной
- входной поток

Как тебя зовут?

Напишем, как и раньше на языке Python, программу, которая спрашивает пользователя, как его зовут, и затем приветствует его.

Имя, полученное от пользователя, будем хранить в переменной. Это символьная строка, в языке C++ она относится к типу **string** (по-английски — строка). В отличие от языка Python переменные в C++ необходимо *объявлять*. Объявление переменной — это запись вида

```
string name;
```

которая говорит транслятору, что переменная name относится к типу **string**, т. е. это символьная строка (а не число, например).

Язык C++ использует *статическую* (неизменяемую) *типизацию*: переменная name будет строкой до конца своей «жизни», т. е. до завершения работы основной программы (точнее — до завершения работы блока в фигурных скобках, где она объявлена).

При объявлении переменной ей выделяется место в памяти, и она постоянно в течение своей «жизни» будет занимать именно это место.

Сначала выведем сообщение с подсказкой:

```
cout << "What is your name? ";
```

Затем вводим данные из входного потока cin и записываем в переменную name:

```
cin >> name;
```

Остаётся вывести приветствие в выходной поток:

```
cout << "Hello, " << name << "!";
```

Приведём программу полностью:

```
#include <iostream>
using namespace std;
int main()
  {
  string name;
  cout << "What is your name? ";
  cin >> name;
  cout << "Hello, " << name << "!";
  }
```

Переменные

Имена переменных (*идентификаторы*) строятся по тем же правилам, что и в языке Python: можно использовать латинские буквы, цифры и знак подчёркивания «_». Строчные и прописные буквы *различаются*, т. е. a и A — разные переменные. Имя не может начинаться с цифры.

Нужно выбирать «говорящие» имена переменных, которые облегчают понимание программы. Например, запись

```
animals = dogs + cats;
```

значительно понятнее, чем

```
a = d + c;
```

(по-английски *animals* — животные, *cats* — кошки, *dogs* — собаки).

Тип переменной в программе на C++ определяется явно при её объявлении. Мы будем использовать следующие основные типы данных:

string — символьная строка;
int — целое число;
float — вещественное число (может иметь дробную часть);
bool — логическое значение;
char — одиночный символ.

Для записи значения в переменную используется оператор присваивания, который, так же как и в языке Python, обозначается знаком «равно»:

```
name = "Dennis Ritchie";
```

В переменную нельзя записывать значение другого типа (не совпадающего с типом переменной). Пусть переменные объявлены так:

```
int houses;
float weight;
string name;
char grade;
```

Тогда эти операторы присваивания ошибочны:

```
houses = 4.5;       // пропадёт дробная часть
houses = "Muhtar"; // ошибка компиляции
weight = "Murka";  // ошибка компиляции
grade = "C++";      // ошибка компиляции
```

При записи вещественного числа в целую переменную пропадёт дробная часть, а символьную строку нельзя сохранить ни в целой, ни в вещественной, ни в символьной переменной.

А вот так (сюрприз!) делать можно:

```
weight = 45;  // (1)
houses = 'A'; // (2)
name = 'Э';   // (3)
grade = 65;   // (4)
```

Что при этом произойдёт?

(1) При записи целого числа в вещественную переменную значение будет преобразовано в равное вещественное число: 45,0 (с нулевой дробной частью).
(2) Символ `'A'` хранится в памяти как числовой код 65, поэтому в переменную `houses` будет записано число 65. На самом деле такую хитрую запись применяют редко.
(3) При записи символа в символьную строку будет построена новая строка из одного символа.
(4) При записи целого числа в символьную переменную это число будет воспринято как код символа.

Сумма чисел

Сложить два заранее известных числа и вывести сумму на экран можно так:

```
cout << 12345 + 67890;
```

или так:

```
int num1 = 12345, num2 = 67890;
cout << num1 + num2;
```

Заметьте, что здесь в первой строке объявлены сразу две переменные: num1 и num2. Переменным присвоены начальные значения прямо при объявлении. Если это не сделано, в переменных находятся случайные данные — «мусор» и использовать эти значения бессмысленно (и даже опасно — это может привести к серьёзной ошибке!).

Теперь напишем программу, которая складывает два числа, введённых с клавиатуры (вспомните, как мы делали это в языке Python). Программа на C++ выглядит так:

```
int num1, num2, sum;
cin >> num1 >> num2;
sum = num1 + num2;
cout << sum;
```

Здесь и далее мы будем для экономии места писать только содержательную часть основной программы.

В отличие от языка Python числа можно вводить по-разному: в одной строке через пробел, по одному в каждой строке и даже пропуская несколько строк между числами.

Теперь добавим приглашение к вводу и оформим вывод результата, чтобы было понятно, что делает программа:

```
int num1, num2, sum;
cout << "Give me two numbers: ";
cin >> num1 >> num2;
sum = num1 + num2;
cout << num1 << "+" << num2 << "=" << sum;
```

В этой задаче можно было обойтись и без переменной sum, потому что выполнять вычисления можно прямо при выводе ответа на экран:

```
cout << num1 << "+" << num2 << "=" << num1+num2;
```

Программа вычислит значение выражения num1+num2 и передаст его в поток вывода. Однако если значение суммы понадобится далее, лучше сохранить его в переменной, а потом везде использовать эту переменную, не вычисляя результат заново.

Выводы

- Переменные в языке C++ нужно объявлять. При объявлении определяется тип переменной и выделяется память, в которой её значение размещается до конца своей «жизни».
- В переменную нельзя записывать значение другого типа, не совместимого с типом переменной.
- Основные типы данных в C++: **int** — целое число; **float** — вещественное число; **char** — символ; **string** — символьная строка; **bool** — логическое значение.
- Тип переменной в C++ нельзя изменить во время работы программы.
- При объявлении переменной лучше сразу задать ей начальное значение. Если начальное значение не задано, в переменной находится «мусор» — неопределённые данные, использовать их бессмысленно и опасно).

Вопросы и задания

1. Чем различаются результаты работы двух операторов?

```
cout << "May!" << "Peace!" << "Cats!";
```

и

```
cout << "May!" << endl << "Peace!"
     << endl << "Cats!";
```

2. Что выведет на экран эта программа?

```
string name = "Bond";
cout << name << ". James " << name << ".";
cout << endl;
cout << "name" << ". James " << "name" << ".";
```

3. Что выведет эта команда при `a = 4, b = 5`?

```
cout << "a" << "+b" << "=" << a+b;
```

4. Исправьте ошибки в команде вывода:

```
cout << "c" << "-b" << = << sum;
```

так чтобы при `a = 4, b = 5` и `sum = 9` программа вывела: `9-5=4`

Интересный сайт

stepik.org/course/363/ — онлайн-курс «Введение в программирование на C++»

§ 16
Компьютерная графика

Ключевые слова:

- окно
- координаты
- оси координат
- пиксель
- цвет контура
- цвет заливки
- код цвета
- прозрачный цвет

Библиотека TX Library

Писать простые графические программы на C++ не всегда просто. Для вывода графики требуется с помощью операционной системы создать специальное окно, обеспечить его взаимодействие с пользователем и т. п.

Для того чтобы упростить вход в мир графического программирования, были разработаны специальные библиотеки. На взгляд автора, наиболее удобна библиотека `TX Library`[1], разработанная И. Р. Дединским. Для работы с графическими примерами её вначале нужно установить на свой компьютер.

После установки библиотеки на рабочем столе появится «Ярлык для TX», он открывает папку со справочной системой, примерами и самой библиотекой. Справочная система расположена в файле `TXLib Help`, советуем найти там раздел «Простейший пример» и прочитать его. В разделе «Рисование» приведены описания графических функций библиотеки.

Простейшая графическая программа, использующая эту библиотеку, состоит всего из нескольких строк:

```
#include "TXLib.h"
int main()
  {
  txCreateWindow (800, 600);
  }
```

Первая строка подключает библиотеку `TX Library` — файл `TXLib.h`, который был скопирован на ваш компьютер во время установки библиотеки.

Единственная строка основной программы создаёт окно для вывода графики. Для этого вызывается команда `txCreateWindow`, ей передаются размеры окна — ширина 800 пикселей и высота 600 пикселей. Холст — область рисования — заливается чёрным цветом.

Больше никаких команд не выполняется, окно закрывается при нажатии любой клавиши.

---

[1] Библиотеку `TX Library` можно загрузить с главной страницы сайта И. Р. Дединского **ded32.ru** или **txlib.ru**.

Управляем пикселями

В TX Library используется стандартная для большинства графических библиотек система координат: пиксель с координатами (0, 0) находится в левом верхнем углу холста, ось *OX* направлена вправо вдоль верхней границы, а ось *OY* — вниз вдоль левой границы холста (а не вверх, как в математике!).

Для управления пикселями используется команда txSetPixel (от английских слов *set* — установить, *pixel* — пиксель). Ей нужно передать координаты пикселя и цвет, который мы хотим ему присвоить. Например, так:

```
txSetPixel(10, 20, TX_LIGHTGREEN);
```

или так:

```
txSetPixel(10, 20, RGB(0,255,128));
```

Эта команда меняет цвет пикселя с координатами (10, 20) на светло-зелёный. Цвет задаётся либо названием (TX_LIGHTGREEN), либо в модели RGB: красная составляющая равна 0, зелёная — 255, а синяя — 128.

Названия цветов определены как константы (постоянные величины) в библиотеке TX Library. Например, TX_BLACK — это чёрный цвет, TX_RED — красный, TX_YELLOW — жёлтый. Полный список цветов, которым даны имена, можно найти в справочной системе библиотеки.

Функция txGetPixel (англ. *get* — получить) позволяет узнать цвет пикселя с заданными координатами. Например, добавив в программу строку

```
cout << txGetPixel(10, 20);
```

мы увидим на экране число 8453888 — код заданного нами цвета. Попробуем вывести его в шестнадцатеричной системе счисления, установив специальный формат hex (от англ. *hexadecimal* — шестнадцатеричный):

```
cout << hex << txGetPixel(10, 20);
```

Получаем: 80ff00. Как это расшифровать? Каждая составляющая цвета может принимать значения от 0 до 255, так что для её хранения нужен один байт. Байт записывается как две шестнадцатеричные цифры. Для светло-зелёного цвета, как мы видели выше, красная составляющая равна 0, зелёная — 255 и синяя — 128. Становится понятно, что старший байт числа $80_{16} = 128$ — это синяя составляющая, следующий байт $FF_{16} = 255$ — зелёная, и третий (младший) байт $00_{16} = 0$ — красная.

Линии и фигуры

Цвет линий устанавливается командой txSetColor:

```
txSetColor( TX_PINK );
```

Можно вызывать функцию и с двумя аргументами: второй задаёт толщину линий:

```
txSetColor( TX_GREEN, 5 );
```

Для рисования линий используется команда txLine:

```
txLine( 10, 20, 100, 150 );
```

Ей передаются четыре аргумента: сначала координаты первого конца отрезка, затем — координаты второго конца.

Линии рисуются тем цветом, который был задан до вызова txLine, поэтому цвет надо задавать заранее, до рисования.

Замкнутые фигуры

Для замкнутых фигур нужно заранее определить не только цвет линии, но и цвет заливки внутренней области. Цвет заливки задаёт команда txSetFillColor (по-английски *fill* — заполнение, заливка):

```
txSetFillColor( TX_YELLOW );
```

После этого все фигуры будут заливаться жёлтым цветом, пока новая команда txSetFillColor не изменит режим заливки.

Если заливка не нужна, необходимо установить прозрачный цвет TX_TRANSPARENT:

```
txSetFillColor( TX_TRANSPARENT );
```

Нарисовать замкнутую ломаную линию можно с помощью команды txPolygon:

```
POINT contour[3] = {{0,0}, {100,100}, {0,50}};
txPolygon( contour, 3 );
```

Здесь мы сначала определяем координаты точек, по которым строится ломаная: координаты каждой точки записываем в фигурных скобках через запятую, а все пары координат вместе взяты ещё в одну пару фигурных скобок. У нас получилась группа данных, которая в информатике называется «массив». Имя массива — contour, он состоит из пар координат — объектов типа POINT, а число 3 в квадратных скобках — это количество точек ломаной линии.

При вызове команды txPolygon мы передаём ей массив точек и его длину (количество точек). Ломаная автоматически замыкается: последняя точка соединяется с первой.

Прямоугольник, стороны которого параллельны осям координат, рисуется так же, как и в графической библиотеке языка Python:

```
txRectangle( 50, 50, 100, 130 );
```

Команде нужно передать четыре аргумента: координаты левого верхнего угла прямоугольника (сначала x-координата, потом — y-координата), а затем — координаты его правого нижнего угла.

Существует и команда для рисования окружности:

```
txCircle( 100, 100, 50 );
```

Первые два её аргумента — x-координата и y-координата центра, третий задаёт радиус окружности.

Как и для замкнутой ломаной, цвет контуров прямоугольников и окружностей определяется командой `txSetColor`, а цвет заливки — командой `txSetFillColor`.

Выводы

- Для создания графических программ в этом пособии используется библиотека `TX Library`.
- Цвет контура фигуры устанавливается с помощью команды `txSetColor`.
- Замкнутые контуры (прямоугольник, окружность, замкнутая ломаная) заливаются тем цветом, который установлен как цвет заливки командой `txSetFillColor`.

Вопросы и задания

1. Используя справочную систему или исходный код функции `txCreateWindow`, выясните, какие ещё параметры она может принимать и что они означают.
2. Используя справочную систему или исходный код библиотеки `TX Library`, выясните, можно ли передавать функции `txSetPixel` вещественные (дробные) координаты пикселя. Как она их обрабатывает?
3. Определите красную, зелёную и синюю составляющие цвета `TX_MAGENTA`.
4. Выясните, каким цветом выполняется заливка по умолчанию (если не вызывать команду `txSetFillColor`).
5. Выясните, как хранится в памяти цвет `TX_TRANSPARENT`. Можно ли задать его с помощью модели RGB?
6. Внимательно прочитайте в справочной системе библиотеки `TX Library` описание команды `txTriangle` и примените её. Соответствует ли её действие описанию?
7. Проверьте, что произойдёт, если рисовать за пределами области холста.

8. Напишите программу, которая рисует какую-нибудь невозможную фигуру, например треугольник Пенроуза (используйте информацию из дополнительных источников).
9. Придумайте и нарисуйте своего персонажа (например, собачку или котика). Используйте функции библиотеки TX Library, которые не описаны в этом параграфе. Их описание ищите в справочной системе библиотеки.

*10. Сделайте так, чтобы координаты персонажа, построенного при выполнении предыдущего задания, можно было менять, вводя с клавиатуры координаты опорной точки (например, центра фигуры).

Интересный сайт

ded32.ru (txlib.ru) — сайт профессиональной проектной работы по информатике для школьников, там же расположена библиотека TX Library

§ 17 Процедуры

Ключевые слова:

- подпрограмма
- процедура
- рефакторинг
- аргументы
- параметры
- базовая точка

Длинная программа

Рассмотрим такую программу, которая рисует на экране машинку:

```
POINT a[4] = {{40, 70}, {65, 55},
              {70, 55}, {70, 70}};
POINT b[4] = {{75, 55}, {95, 55},
              {110, 70}, {75, 70}};
POINT c[8] = {{20, 95}, {20, 70},
              {30, 70}, {60, 50},
              {100, 50}, {120, 70},
              {160, 80}, {160, 95} };
txSetFillColor( TX_LIGHTBLUE );
txPolygon( c, 8 );
txSetFillColor( TX_BLACK );
txPolygon( a, 4 );
txPolygon( b, 4 );
txSetFillColor( TX_GRAY );
txCircle( 50, 115, 13 );
txCircle( 120, 95, 13 );
```

Если её запустить, мы увидим, что одно из колес машинки стоит не на месте. Чтобы найти ошибку, нужно как-то определить строки, где рисуются колёса. Сделать это не так просто, потому что эта программа написана плохо:

- трудно понять, где рисуются части машинки: кузов, колеса, окна;
- имена переменных a, b и c ничего не говорят о том, что в них хранится.

Первое, что можно сделать, — разбить программу на смысловые блоки, отделив их друг от друга пустыми строками:

```
POINT c[8] = {{20, 95}, {20, 70},
              {30, 70}, {60, 50},
              {100, 50}, {120, 70},
              {160, 80}, {160, 95} };
txSetFillColor( TX_LIGHTBLUE );
txPolygon( c, 8 );

txSetFillColor( TX_BLACK );
POINT a[4] = {{40, 70}, {65, 55},
              {70, 55}, {70, 70}};
txPolygon( a, 4 );
POINT b[4] = {{75, 55}, {95, 55},
              {110, 70}, {75, 70}};
txPolygon( b, 4 );

txSetFillColor( TX_GRAY );
txCircle( 50, 115, 13 );
txCircle( 120, 95, 13 );
```

В первый блок мы объединили строки, которые используют набор (массив) точек c, во втором блоке задействованы переменные a и b, в третьем вообще нет переменных.

Теперь попытаемся понять, что делает каждый блок. Сначала можно заключить в комментарии все блоки, кроме первого, и запустить программу. Получится только кузов машины, поэтому делаем вывод: первый блок рисует кузов. Тем же способом выясняем, что второй блок рисует окна, а третий — колёса. Значит, ошибку нужно искать в третьем блоке.

Рефакторинг

Выполним *рефакторинг*, т. е. сделаем программу более структурной и понятной, не изменяя результатов её работы. В первую очередь нужно дать переменным «говорящие» имена.

Набор точек с именем c назовём carBody (по-английски — кузов машины), а наборы точек a и b — соответственно backWindow (по-английски — заднее окно) и frontWindow (переднее окно).

Все команды, связанные с рисованием кузова, выделим в отдельный блок — подпрограмму (процедуру), которую можно назвать drawCarBody (по-английски — «нарисовать кузов машины»):

```
void drawCarBody()
  {
  POINT carBody[8] = {{20, 95}, {20, 70},
                      {30, 70}, {60, 50},
                      {100, 50}, {120, 70},
                      {160, 80}, {160, 95} };
  txSetFillColor( TX_LIGHTBLUE );
  txPolygon( carBody, 8 );
  }
```

Слово **void** (по-английски — пустой) означает, что эта подпрограмма не возвращает никакого результата. Такие подпрограммы называют *процедурами*.

Круглые скобки после имени служат для того, чтобы передавать подпрограмме данные (параметры), но в этой процедуре параметров нет.

Всё тело процедуры взято в фигурные скобки, так же как и основная программа (main).

Аналогично можно выделить ещё одну процедуру: рисование окон. Назвать её можно, например, drawCarWindows (нарисовать окна машины):

```
void drawCarWindows()
  {
  txSetFillColor( TX_BLACK );

  POINT backWindow[4] = {{40, 70}, {65, 55},
                         {70, 55}, {70, 70}};
  txPolygon( backWindow, 4 );

  POINT frontWindow[4] = {{75, 55}, {95, 55},
                          {110, 70}, {75, 70}};
  txPolygon( frontWindow, 4 );
  }
```

Пустые строки в процедуре выделяют смысловые части: выбор цвета заливки, рисование заднего окна, рисование переднего окна.

Процедуру, которая рисует колёса, можно назвать drawCarWheels, вы можете написать её самостоятельно.

Теперь основная программа состоит из команды создания окна и вызовов трёх процедур, а сами процедуры нужно расположить выше основной программы для того, чтобы компилятор уже знал о них, когда начнёт обрабатывать основную программу:

```
void drawCarBody()
  {
  ... // команды рисования кузова
  }
void drawCarWindows()
  {
  ... // команды рисования окон
  }
void drawCarWheels()
  {
  ...  // команды рисования колёс
  }
int main()
  {
  txCreateWindow( 500, 400 );
  drawCarBody();
  drawCarWindows();
  drawCarWheels();
  }
```

Вместо многоточий нужно подставить команды, которые входят в эти процедуры.

Как и в языке Python, процедура может вызывать другие процедуры. Например, можно оформить рисование всей машинки в отдельную процедуру (для удобства её вызова):

```
void drawCar()
  {
  drawCarBody();
  drawCarWindows();
  drawCarWheels();
  }
```

В этом случае уже не придётся рисовать машинку тремя строками — достаточно будет одного вызова функции `drawCar`.

Процедуры с параметрами

Пока процедуры в нашей программе неуправляемы — мы не можем изменить положение машинки на экране, при каждом вызове она рисуется в том же месте (это довольно скучно!). Чтобы устранить этот недостаток, нужно передавать в процедуру координаты машинки (точнее, её *базовой точки*), например так:

```
drawCar( 100, 100 );
```

Данные, передаваемые в процедуру при вызове, называют *аргументами*. В процедуре они обозначаются именами и называются *параметрами*:

```
void drawCar( int x, int y )
  {
  ...
  }
```

В отличие от языка Python в C++ перед каждым параметром в заголовке функции нужно указать его тип: в нашем случае обе величины целые: они относятся к типу **int**.

Значения координат нужно передать в три процедуры, которые рисуют разные части машинки:

```
drawCarBody( x, y );
drawCarWindows( x, y );
drawCarWheels( x, y );
```

В эти три процедуры тоже придётся внести изменения, пересчитав координаты всех точек через базовую точку.

Выберем в качестве базовой точки левый нижний угол кузова. Как же пересчитать координаты? Несложно понять, что левый нижний угол кузова — это первая точка в массиве `carBody`. Она сейчас имеет координаты (20, 95), именно их мы обозначаем через `(x, y)` и будем считать базовой точкой всей машинки.

Возьмём следующую точку, (20, 70). Её x-координата совпадает с x-координатой базовой точки, а y-координата (70) на 25 пикселей меньше, чем y-координата базовой точки. Поэтому для этой точки получаем относительные координаты (относительно базовой точки машинки) `(x, y-25)`.

Используя такой подход для всех остальных точек, получаем набор координат точек для рисования кузова:

```
POINT carBody[8] = {{x, y}, {x, y-25},
       {x+10, y-25}, {x+40, y-45},
       {x+80, y-45}, {x+100, y-25},
       {x+140, y-15}, {x+140, y}};
```

Координаты окон и колёс пересчитайте самостоятельно. Стоит только помнить, что базовая точка (x, y) выбирается для всей машинки, а не для каждой из трёх частей независимо, иначе эти части будут неверно размещены на рисунке.

В процедуру `drawCar` можно добавить и другие параметры, например цвет кузова. Его нужно передать только в процедуру `drawCarBody`, а в остальные процедуры передавать не нужно, так как их работа от цвета кузова не зависит. Для объявления цвета используется специальный тип данных **COLORREF**:

```
void drawCar( int x, int y, COLORREF color )
  {
  ...
  }
```

Выводы

- Процедура — это блок кода, имеющий имя и оформленный как отдельная часть программы. Процедуру можно вызывать из других частей программы.
- Вызвать процедуру — значит запустить её на исполнение, записав её имя с круглыми скобками, в которых перечисляются аргументы — данные, передаваемые процедуре.
- Параметры — это данные, которые принимает процедура. В процедуре к параметрам можно обращаться по именам. В языке C++ каждый параметр имеет свой тип.
- Рефакторинг — это изменение программы, которое не влияет на результат её работы, но делает её более понятной для человека и удобной для внесения изменений.

Вопросы и задания

1. В программу рисования машинки добавьте строки, которые рисуют двери.
2. Измените программу рисования машинки так, чтобы можно было рисовать машинки в разных местах окна программы.
3. Доработайте процедуру `drawCar` так, чтобы можно было изменять цвет машинки. С помощью этой процедуры нарисуйте четыре машинки разных цветов в разных местах холста.
4. *Проект*. Придумайте своих персонажей (например, героев вашей будущей компьютерной игры) и напишите процедуру, которая позволяет рисовать персонажей в заданном месте холста, изменять их цвета и т. п.

§ 18
Обработка целых чисел

Ключевые слова:

- арифметические выражения
- частное
- остаток
- форматный вывод
- случайные числа
- зерно

Ограниченность значений целых чисел

Одно из достоинств языка Python — практически неограниченный диапазон значений целых переменных. Этот язык использует так называемую «длинную арифметику»: когда нужно, интерпретатор автоматически («прозрачно» для программиста) расширяет область памяти, которая выделена для хранения числа. Это позволяет нам свободно

работать с большими числами, не особенно задумываясь о том, как именно они хранятся и обрабатываются, но это достигается ценой огромного замедления работы программы.

Совсем другая ситуация в языке C++. Место для хранения переменной жёстко фиксировано и зависит от типа данных и транслятора. Например, для хранения целой переменной типа **int** обычно выделяется 4 байта памяти, т. е. 32 бита. Это позволяет закодировать всего 2^{32} различных чисел, из которых $2^{31} - 1$ положительных, один ноль и 2^{31} отрицательных. Таким образом, наибольшее положительное целое число, которое может быть записано в переменную типа **int**, равно $2^{31} - 1 = 2147483647$. Зато такие числа напрямую обрабатываются процессором компьютера, что позволяет оперировать ими с огромной скоростью. Это может быть важно в научных расчётах или компьютерных играх.

Если этого диапазона не хватает (нужно работать с бóльшими числами), можно применить тип **int64_t**. Такие переменные занимают в памяти 8 байт, и диапазон их возможных значений намного шире, чем у типа `int`.

Определить размер переменной в байтах можно с помощью команды `sizeof` (от англ. *size* — размер):

```
int i;
cout << sizeof(i);
```

или даже так:

```
cout << sizeof(int);
```

Арифметические выражения

Арифметические выражения в C++ строятся по тем же правилам, что и в других языках программирования. Поэтому мы не будем повторяться, а остановимся, прежде всего, на особенностях языка C++ и его отличиях от Python.

Для возведения в степень в C++ используется не оператор, как в Python, а функция. Чтобы возвести число `x` в степень `y`, нужно подключить библиотеку `cmath`:

```
#include <cmath>
```

а затем вызвать функцию `pow` (от английского *power* — степень), определённую в этой библиотеке:

```
cout << pow(x, y);
```

Так же как и в Python, можно использовать каскадное присваивание:

```
x = y = 3;
```

Здесь в обе переменные записывается значение 3.

Есть и сокращённая запись операций: вместо

```
i = i + z;
```

можно написать

```
i += z;
```

и т. п. Кроме того, в C++ есть ещё две операции для увеличения и уменьшения значения переменной на 1:

```
i++;        // то же, что и i += 1
i--;        // то же, что и i -= 1
```

Чаще всего такая запись используется при увеличении или уменьшении переменных в циклах.

Длинные (и не очень) арифметические выражения (и другие операторы) можно переносить на следующую строку:

```
a = x*(2*c + 4*b – 8*f)
    + 2*a*c;
```

Никакого «знака переноса» ставить не нужно, ведь транслятор C++ определяет конец оператора присваивания по точке с запятой, которой этот оператор всегда заканчивается.

Если в выражение входят переменные разных типов, во многих случаях происходит их автоматическое преобразование (приведение типа). Например, результат умножения целого числа на вещественное — это вещественное число. Запись вещественного числа (его целой части) в целую переменную возможна, но транслятор выдаст предупреждение (англ. *warning*), ведь часть информации (по крайней мере, дробная часть числа) при этом будет потеряна.

Деление и остаток

В языке C++ деление целых чисел выполняется особым образом: результат деления целого числа на целое число — это всегда целое число, а остаток при этом отбрасывается. Сам остаток от деления можно найти с помощью операции %, как и в языке Python. Например, такая программа:

```
cout << 7 / 4 << " " << 7 % 4 << endl;
cout << 4 / 7 << " " << 4 % 7;
```

выведет

```
1 3
0 4
```

Программа, которая выделяет целые минуты и секунды (от 0 до 59) из интервала времени в секундах, выглядит почти так же, как на языке Python:

```
int timeSec = 289;
int minutes = timeSec / 60;     // 4
int seconds = timeSec % 60;     // 49
```

Если нам нужен вещественный результат деления (с дробной частью), нужно преобразовать делимое или делитель к вещественному типу. Это можно сделать, например, домножив число на 1.0 (именно так, с нулевой дробной частью) или указав нулевую дробную часть в самом числе. Оператор

```
cout << 7.0 / 4;
```

выведет значение 1,75. Обратите внимание: в программе для отделения дробной части числа от целой части используется точка, а не запятая. Если же значения записаны в переменных, нужно преобразовать делимое или делитель в вещественное число с помощью функции float:

```
int a = 3, b = 4;
float x;
x = a / b;                  // 0
x = a / float(b);           // 0.75
x = float(a) / b;           // 0.75
x = float(a) / float(b);    // 0.75
```

Вывод данных на экран

Выходной поток можно форматировать, т. е. задавать количество позиций, отводимых на вывод каждого значения. В C++ для этого используют специальные команды форматирования (*манипуляторы*) вывода, определённые в библиотеке iomanip. Её нужно подключить к программе с помощью строки

```
#include <iomanip>
```

В результате выполнения фрагмента программы

```
int a = 12, b = 5;
cout << setw(4) << a << setw(4) << b;
```

на экран выводятся два значения, на каждое из которых отводится по 4 позиции (знак ∘ обозначает пробел):

```
∘∘12∘∘∘5
```

По умолчанию (т. е. «обычно», если явно не сказано делать иначе) данные выравниваются вправо. Если нужно выравнивание влево, используется команда left:

```
cout << left << setw(4) << a << setw(4) << b;
```

Теперь дополнительные пробелы будут выводиться справа от чисел:

```
12∘∘5∘∘∘
```

Случайные числа

В стандартную библиотеку языка C++, random, входят функции для генерации случайных (точнее, псевдослучайных) чисел. Эту библиотеку нужно подключить командой

```
#include <random>
```

Функция rand возвращает случайное целое число на отрезке [0; RAND_MAX], где RAND_MAX — это константа (постоянная), определённая в библиотеке cstdlib. Во многих системах значение RAND_MAX равно 32767. Программа

```
cout << rand() << endl;
cout << rand() << endl;
cout << rand() << endl;
```

выводит три разных больших случайных числа.

Как же получить случайное целое число на заданном отрезке [a; b]? Самый простой способ — использовать операцию взятия остатка от деления. Действительно, остаток от деления какого-то числа на N всегда находится на отрезке [0; N–1], так что оператор

```
k = rand() % N;
```

записывает в переменную k случайное число на этом отрезке. Если добавить к этой величине значение a, то весь отрезок смещается так, чтобы его левая граница была в точке a:

```
k = a + rand() % N;
```

Это число расположено уже на отрезке [a; $a + N - 1$]. Для того чтобы отрезок совпал с отрезком [a; b], нужно выбрать N так, чтобы выполнялось условие $b = a + N - 1$. Решив это уравнение относительно N, находим: $N = b - a + 1$. Таким образом, нам нужен оператор

```
k = a + rand() % (b-a+1);
```

Если несколько раз запускать программу, использующую функцию rand, мы увидим, что последовательность случайных чисел получается всегда одна и те же. Дело в том, что она строится с помощью математической формулы: по предыдущему числу рассчитывается следующее, так что вся последовательность полностью определяется начальным числом (*зерном*). А это зерно при каждом запуске программы по умолчанию устанавливается одно и то же. Сделано это специально, для упрощения отладки, чтобы можно было легко воспроизвести в программе ошибочную ситуацию.

Изменяет значение зерна функция srand из библиотеки random. Чтобы зерно каждый раз было разным, можно выбирать в качестве зерна текущее время, которое возвращает функция time из библиотеки ctime:

```
#include <random>
#include <ctime>
...
srand( time(0) );
cout << rand() << endl;
```

Теперь при каждом запуске мы будем строить новую последовательность случайных чисел.

Выводы

- Память для хранения переменных в C++ выделяется при объявлении переменных.
- На переменную типа **int** обычно выделяется 4 байта, она может принимать значения от -2^{31} до $2^{31}-1$.
- Переменная типа **int64_t** занимает 8 байт в памяти.
- Операция деления / для целых чисел всегда даёт в результате целое число (остаток отбрасывается). Если нужно получить вещественный результат, делимое или делитель надо явно преобразовать в вещественное число.
- Для взятия остатка от деления используют оператор %.
- Форматный вывод данных выполняется с помощью команд из библиотеки iomanip.
- Случайные числа можно получить с помощью функции rand из библиотеки random.

Вопросы и задания

1. Выполните следующую программу:
```
int i = 2147483647;
cout << i;
cout << endl << i+1;
```
Объясните результаты. Чтобы изучить вопрос более глубоко, найдите в Интернете информацию о хранении отрицательных чисел в памяти компьютера.
2. Оцените минимальное и максимальное значения для переменной типа long long.
3. Исследуйте, как выполняются операции деления и взятия остатка для отрицательных целых чисел (сравните результаты с работой интерпретатора языка Python).
4. Запишите эти операции в сокращённой форме:
```
a = a + 1;
b = (25 + a)*b;
c = c - 1;
b = b - 3*d;
```

5. Напишите программу, которая меняет местами значения двух переменных в памяти.
6. В предыдущей задаче попробуйте найти решение, которое не использует дополнительные переменные.
7. Как с помощью операций / и % выделить вторую с конца цифру числа?
8. Что будет выведено в результате работы следующей программы?

```
int a=1, b=2, c=3, d=4, e=5;
cout << setw(3) << a << endl;
cout << setw(2) << b << setw(2) << b << endl;
cout << c << setw(4) << c << endl;
cout << setw(2) << d << setw(2) << d << endl;
cout << e << setw(2) << e
     << setw(2) << e << endl;
```

9. Что будет выведено в результате работы следующих фрагментов программ?

а)
```
int a=5, b=3;
cout << a << "=Z(" << b << ")";
```

б)
```
int a=5, b=3;
cout << "Z(a)=" << "(b)";
```

в)
```
int a=5, b=3;
cout << "Z(" << a << ")=(" << a+b << ")";
```

10. Запишите оператор для вывода значений целых переменных a = 5 и b = 3 в формате:
    а) 3+5=?
    б) Z(5)=F(3)
    в) a=5; b=3;
    г) Ответ: (5;3)
11. Напишите программу, которая вводит целое число x из отрезка [8; 15] и выводит его на экран в двоичной системе счисления.
12. Напишите программу, которая вводит двоичную запись целого числа x ($0 \le x \le 31$), и выводит это число на экран в десятичной системе счисления.
13. Напишите программу, которая вводит целое число x ($0 \le x \le 511$) и выводит его на экран в восьмеричной системе счисления.
14. Напишите программу, которая вводит восьмеричную запись целого числа x ($0 \le x \le 511$) и выводит это число на экран в десятичной системе счисления.

*15. Напишите программу, которая вводит целое число x ($0 \le x \le 255$) и выводит его на экран в шестнадцатеричной системе счисления.

*16. Напишите программу, которая вводит шестнадцатеричную запись целого числа x ($16 \le x \le 255$) и выводит это число в десятичной системе счисления.

17. В некоторой стране используют недесятичную денежную систему: 1 хрямзик = 8 грымзиков, 1 грымзик = 5 йцукенов. Напишите программу, которая вводит сумму денег в йцукенах и определяет, как заплатить эту сумму наименьшим количеством монет. Например: 99 йцукенов = 2 хрямзика + 3 грымзика + 4 йцукена.

Интересный сайт

ru.stackoverflow.com — сайт вопросов и ответов для программистов

§ 19
Обработка вещественных чисел

Ключевые слова:

- вещественное число
- научный формат
- мантисса
- форматный вывод
- округление

Вещественные числа в языке C++

При записи вещественных чисел дробная часть, как и во всех языках программирования, отделяется точкой:

```
float x = 4.321;
```

Очень большие или очень маленькие числа можно записывать в *научном формате*. Например, величину, равную заряду электрона ($1{,}60217662 \cdot 10^{-19}$ Кл), можно записать в переменную так:

```
float qElectron = 1.60217662e-19;
```

Здесь первую часть `1.60217662` часто называют *мантиссой* или значащей частью числа, а `-19` после буквы `e` — это порядок числа.

Научный формат используется и для записи больших чисел. Расстояние от Земли до Солнца ($1{,}496 \cdot 10^{11}$ м) записывается в переменную `distToSun` так:

```
float distToSun = 1.496e11;
```

Как вы уже знаете, практически все вещественные числа не могут храниться в памяти с идеальной точностью. Это происходит из-за того, что для хранения числа выделяется ограниченное количество битов. Поэтому и вычисления с вещественными числами чаще всего неточны, причём ошибка накапливается при выполнении новых операций.

Запустим следующую программу, которая складывает числа 0,1 и 0,2 и затем вычитает из этой суммы число 0,3:

```
float x = 0.1, y = 0.2;
float sum = x + y;
cout << sum << endl;
diff = sum - 0.3;
cout << diff;
```

На экран выводится результат:

```
0.3
1.19209e-008
```

Первое число — сумма значений x и y — (вроде бы) равна 0,3. Второе число представляет значение $1{,}19209 \cdot 10^{-8}$ — это ошибка вычисления разности. На самом деле и сумма тоже получилась неточной (она немного отличается от 0,3). Дело в том, что при выводе по умолчанию программа оставляет 6 значащих цифр числа, а следующие 5 цифр после цифры 3 получились нулевыми, и программа их не вывела. Вторая выведенная строка показывает, что ошибка появляется в восьмом–девятом знаке после запятой.

Чем меньше места в памяти выделяется для хранения переменной, тем больше будет эта ошибка. Поэтому для того, чтобы уменьшить вычислительные ошибки при работе с вещественными числами, в C++ часто вместо типа **float** используют тип **double** (по-английски — двойной) — вещественное число с двойной точностью, которое занимает в памяти не четыре байта, как **float**, а восемь.

Ввод и вывод

Ввести три вещественных числа из входного потока и записать их в переменные x, y и z можно так:

```
float x, y, z;
cin >> x >> y >> z;
```

Первое введённое число попадает в переменную x, второе — в y, третье — в z. Числа при вводе можно разделять пробелами или символами «новая строка» (нажимая клавишу Enter после ввода числа).

При выводе вещественных значений по умолчанию выводятся 6 значащих цифр. Причём формат вывода — научный или с фиксированной запятой — тоже выбирается автоматически.

Вывод можно настроить так, как вам нужно, используя специальные средства средства библиотеки iomanip — манипуляторы, управляющие выводом. Сначала эту библиотеку нужно подключить:

```
#include <iomanip>
```

Если мы хотим использовать формат с фиксированной запятой (а не научный), нужно применить команду fixed (по-английски — фиксиро-

ванный), а затем определить количество цифр в дробной части с помощью команды setprecision:

```
cout << fixed << setprecision(9);
```

Мы потребовали вывести 9 цифр в дробной части, и после выполнения фрагмента

```
float x = 0.1, y = 0.2;
float sum = x + y;
float diff = sum - 0.3;
cout << sum << endl << diff;
```

получаем такой результат:

```
0.300000012
0.000000012
```

Можно вывести числа и в научном формате. Для этого применим команду scientific (по-английски — научный):

```
cout << scientific << setprecision(9);
```

Для научного формата setprecision задаёт количество цифр мантиссы (значащей части) после десятичной точки. Поэтому тот же оператор вывода сработает иначе:

```
3.000000119e-001
1.192092913e-008
```

Как и для целых чисел, с помощью команды setw можно задать общее число позиций, отводимых на вывод числа («ширину» поля вывода):

```
cout << fixed << setprecision(2);
cout << setw(6) << x+y << endl;
cout << setw(8) << diff << endl;
```

В пределах заданной ширины значение прижимается к правому краю.

Заметим, что команда setw действует только на ближайшее выводимое значение, потом её действие отменяется. Если нужно задать ширину поля для следующего числа, команду надо повторить.

Операции с вещественными числами

В стандартную математическую библиотеку, которая подключается с помощью команды

```
#include <cmath>
```

включено много математических функций, в том числе:

- `abs( x )` — модуль числа x;
- `sqrt( x )` — квадратный корень числа x;
- `sin( x )` — синус угла x, заданного в радианах;
- `cos( x )` — косинус угла x, заданного в радианах;
- `pow( x, y )` — возведение числа x в степень y.

При работе с вещественными числами часто приходится округлять их до ближайших целых чисел. При явном преобразовании типа просто отсекается дробная часть:

```
float x = 1.6, y = -1.6;
int intX, intY;
intX = int( x );  // = 1
intY = int( y );  // = -1
```

Кроме того, можно использовать стандартные функции из математической библиотеки:

`floor(x)` — наибольшее целое, меньшее или равное `x` (округление «вниз»);

`ceil(x)` — наименьшее целое, большее или равное `x` (округление «вверх»).

Вот примеры использования этих функций:

```
float x = 1.6, y = -1.6;
int intX, intY;
intX = floor(x);  // = 1
intY = floor(y);  // = -2
intX = ceil(x);   // = 2
intY = ceil(y);   // = -1
```

В библиотеке `cmath` введена константа `M_PI`, равная числу π (3,1415626...).

Случайные числа

Случайное вещественное число можно получить с помощью функции `rand` из библиотеки `random`.

Пусть нам нужно случайное вещественное значение на отрезке $[a, b]$. Результат работы `rand` — целое число на отрезке от 0 до `RAND_MAX`. Преобразовав его в вещественное число и разделив на `RAND_MAX`, мы получим значение на отрезке [0, 1]:

```
float xRand;
xRand = float( rand() )/RAND_MAX;
```

Если это значение умножить на `W`, получается число на отрезке [0, `W`]:

```
xRand = W*float( rand() )/RAND_MAX;
```

Теперь остаётся сдвинуть начало отрезка так, чтобы он начинался в точке `a`:

```
xRand = a + W*float( rand() )/RAND_MAX;
```

и выбрать значение `W` так, чтобы `b = a + W`. Окончательно

```
xRand = a + (b-a)*float( rand() )/RAND_MAX;
```

Выводы

- Для повышения точности вычислений с вещественными числами используют переменные типа **double** (с двойной точностью).
- Вещественные числа можно записывать и выводить на экран в формате с фиксированной запятой или в научном формате.
- Вычисления с вещественными числами, как правило, выполняются неточно, с погрешностью.
- Для управления выводом вещественных чисел удобно использовать команды из библиотеки `iomanip` — манипуляторы. С их помощью можно задать формат (с фиксированной запятой или научный), точность, общее количество знаков для вывода числа.
- Для использования математических функций нужно подключить библиотеку `cmath`.

Вопросы и задания

1. Умный Петя заменил во всех своих программах тип вещественных переменных с **float** на **double**. Обсудите достоинства и недостатки такого решения.
2. Что будет выведено в результате работы следующей программы?
```
float x = 172.36589012345678;
cout << x << endl;
cout << fixed << setprecision(2);
cout << setw(10) << x;
```
3. Напишите программу, которая возводит значение x в степень y. С её помощью вычислите $2^{7,25}$ и $3^{1,5}$. Постарайтесь заранее (без вычислений) примерно оценить диапазоны, в которых находится каждое из этих значений.
4. Экспериментально определите точность вычисления суммы 0,1 + 0,2 для переменных типа `double`.

*5. Напишите программу, которая округляет вещественное число до ближайшего целого, не используя стандартные функции.

6. Напишите программу, которая вводит три числа и вычисляет их среднее арифметическое и среднее геометрическое (узнайте из дополнительных источников, что это такое).
7. Напишите программу, которая вычисляет, на какую высоту поднимется теннисный мячик, брошенный вертикально вверх с заданной скоростью, если не учитывать сопротивление воздуха. Необходимые формулы и данные найдите в дополнительных источниках.
8. Напишите программу, которая вводит координаты вершин треугольника и определяет его площадь.
9. Напишите программу, которая вводит координаты вершин тяжёлого треугольника и определяет координаты его центра тяжести.

10. Напишите программу, которая вводит размеры и толщину прямоугольного листа латуни и вычисляет массу этого листа. Плотность латуни найдите в дополнительных источниках.
11. Доработайте программу из предыдущего задания так, чтобы пользователь мог вводить плотность материала, из которого изготовлен лист.
12. Напишите программу, которая вводит диаметр и толщину круглой серебряной монеты и вычисляет массу этой монеты.
13. Напишите программу, которая вводит толщину золотой монеты и её массу и вычисляет её диаметр.

*14. Напишите программу, которая вводит диаметр стального шарика и вычисляет массу этого шарика.

*15. Биллиардные шары делают из материала с плотностью 1900 кг/м^3. Напишите программу, которая вводит массу шара и определяет его диаметр.

16. Напишите программу для расчёта массы прямого отрезка трубы (без изгибов). Подумайте, какие данные нужны для расчёта.

*17. Напишите программу, которая вычисляет период колебаний математического маятника по его параметрам. Формулу для расчёта найдите в дополнительных источниках.

18. Владислав покупает апельсины оптом и хранит их на складе. Каждый день их масса за счёт испарения воды уменьшается на 1%. Напишите программу, которая вводит начальную массу апельсинов и определяет оставшуюся массу через месяц хранения на складе.

*19. Мария захотела открыть счёт в банке. Напишите программу, которая вводит начальный размер вклада и ставку (в процентах) и вычисляет доход за ближайшие 5, 10 и 15 лет. Учтите *эффект капитализации* — начисления процентов на проценты. Для простоты считайте, что проценты дохода по вкладу начисляются ежегодно, а не ежемесячно.

*20. Серафима Петровна захотела открыть счёт в банке. Напишите программу, которая вводит начальный размер вклада и желаемый доход и определяет минимальную ставку (в процентах), которая позволит получить Серафиме Петровне такой доход за ближайшие 10 лет.

*21. Герасим решил взять ипотеку. Известен размер займа, ставка по ипотеке (в процентах) и период (количество лет), на который берётся ипотека. Напишите программу, которая определяет величину переплаты при ежегодных платежах.

22. Напишите программу, которая вводит радиус окружности, центр которой находится в начале координат, и определяет координаты случайной точки на этой окружности.

23. Напишите программу, которая вводит координаты центра и радиус окружности и определяет координаты случайной точки на этой окружности.

*24. Напишите программу, которая вводит координаты двух противоположных углов прямоугольника и определяет координаты случайной точки внутри этого прямоугольника.

*25. Напишите программу, которая вводит координаты вершин треугольника и определяет координаты случайной точки внутри этого треугольника.

§20 Ветвления

Ключевые слова:

- условный оператор
- полная форма
- неполная форма
- составной оператор
- вложенный условный оператор
- логические переменные

Условный оператор

Ветвления применяют тогда, когда последовательность выполнения команд зависит от исходных данных. Рассмотрим самую простую задачу: записать в переменную `M` наибольшее из значений переменных `a` и `b`.

Условный оператор в языке C++ записывается так:

```
if ( a > b )
  M = a;
else
  M = b;
```

Отметим отличия от языка Python:

- всё условие после слова **if** обязательно берётся в скобки, после него двоеточие не ставится;
- после слова **else** двоеточие не ставится;
- отступы не влияют на работу программы, но влияют на работу программиста: делают программу более понятной.

Иногда бывает удобно записывать короткие выполняемые операторы в той же строке, что и служебные слова **if (else)**:

```
if ( a > b ) M = a;
else         M = b;
```

До этого момента мы видели условные операторы *в полной форме*: одна часть (после слова **if**) выполняется, когда условие истинно, вторая (после **else**) — когда условие ложно.

Условный оператор *в неполной форме* не имеет части, которая начинается словом **else**:

```
M = a;
if ( b > a ) M = b;
```

Когда условие ложно, ничего делать не требуется.

Если при выполнении какого-то условия нужно выполнить сразу несколько действий, все нужные операторы объединяют в блок (*составной оператор*) с помощью фигурных скобок:

```
if ( a > b ) {
  int temp = a;
  a = b;
  b = temp;
  }
```

Вспомогательная переменная `temp` (от англ. *temporary* — временный) объявлена в теле условного оператора, внутри блока в фигурных скобках. Время её «жизни» ограничено этим блоком, обращаться к ней в другом месте программы нельзя.

Все операторы, входящие в блок, обычно сдвинуты на одинаковое расстояние от левого края. Язык C++ не требует такого оформления, но в программе с правильными отступами разбираться значительно легче.

После слова **else** также может быть не один оператор, а составной оператор в фигурных скобках.

Кроме знаков < и > в условиях можно использовать другие знаки отношений: <= (меньше или равно), >= (больше или равно), == (равно, два знака «равно» без пробела, чтобы отличить от оператора присваивания) и != (не равно).

Вложенные условные операторы

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы. Это позволяет делать выбор не только из двух, но и из нескольких возможных вариантов.

Например, напишем программу, которая управляет трёхрежимной системой охлаждения двигателя. При температуре до 75° работает режим 1, от 75° до 90° — режим 2, а более 90° — режим 3. Температура измеряется датчиком и записывается в переменную `temp`. Программа должна вывести номер режима охлаждения, который нужно включить.

Одним условным оператором тут не обойтись, потому что есть три возможных результата — три режима. Один из вариантов решения — сначала выяснить, не нужно ли включить режим 1, а потом (если температура не меньше 75°) выбирать между оставшимися двумя режимами:

```
if( temp < 75 )
  cout << "Режим 1";
else
  if( temp < 90 )
    cout << "Режим 2";
  else
    cout << "Режим 3";
```

Условный оператор, выделенный фоном, находится внутри блока «иначе» (**else**), поэтому он называется вложенным условным оператором.

Иногда такие условные операторы записывают так же, как и цепочку операторов **elif** в языке Python:

```
if( temp < 75 )
  cout << "Режим 1";
else if( temp < 90 )
  cout << "Режим 2";
else
  cout << "Режим 3";
```

Все показанные выше фрагменты кода оформлены с помощью отступов так, чтобы их было легко читать. Если отступов не делать, программа всё равно будет работать, но разбираться в ней будет очень сложно. Посмотрите на этот плохо оформленный (но работающий!) вариант:

```
if( temp < 75 ) cout << "Режим 1"; else if
( temp < 90 ) cout <<
"Режим 2"; else  cout << "Режим 3";
```

Рассмотрим ещё один пример, в котором используются вложенные условные операторы:

```
if( temp < 10 )
  if( temp < 0 )
    cout << "Холодно!";
  else
    cout << "Прохладно!";
```

Здесь слово **else** относится к ближайшему условному оператору, то есть ко второму (**if** temp < 0). Поэтому сообщение «Прохладно!» выводится при 0 ≤ temp < 10. Если нужно, чтобы часть **else** относилась к первому оператору (**if** temp < 10), внутренний условный оператор нужно заключить в блок — ограничить фигурными скобками:

```
if( temp < 10 ) {
  if( temp < 0 )
    cout << "Холодно!";
  }
else
  cout << "Тепло!";
```

В этом случае сообщение "Тепло!" выводится для значений temp, больших или равных 10.

Логические переменные

Так же, как и в Python, в C++ есть логические (булевы) переменные, которые могут принимают значения `true` («истина») или `false` («ложь»). Они относятся к типу `bool`:

```
bool b = true;
b = false;
```

Интересно, что команда

```
cout << sizeof(bool);
```

выводит 1, т. е. переменная `b` занимает целый байт в памяти, хотя для неё достаточно одного бита. Дело в том, что отдельные биты в памяти хранить невозможно — только целые байты, состоящие из восьми бит. Это связано с тем, что в памяти персональный адрес имеет только каждый байт, но не отдельный бит.

В логической переменной можно хранить значение какого-то условия и затем использовать его в условном операторе.

```
int a;
cin >> a;
bool isEven = (a % 2 == 0);
```

Здесь переменная `isEven` будет равна `true`, если введённое значение переменной `a` — чётное число, и `false` — если нечётное. Заметьте, что переменная `isEven` будет хранить информацию о чётности значения `a` на момент проверки, даже если значение `a` после этого изменится.

Пусть нужно выяснить, есть ли среди значений переменных `a`, `b` и `c` хотя бы два равных. Нам нужно проверить равенство в трёх парах: `a` и `b`, `a` и `c`, `b` и `c`. Используем логическую переменную вместо вложенных условных операторов:

```
bool found = false; // пока не нашли пару равных
if( a == b ) found = true; // проверяем пару a, b
if( a == c ) found = true; // проверяем пару a, c
if( b == c ) found = true; // проверяем пару b, c
```

Первые две строки можно заменить одной:

```
bool found = (a == b);
```

которая записывает в переменную `found` значение `true`, если значения `a` и `b` равны.

Теперь логическую переменную можно использовать вместо условия в условном операторе:

```
if( found )
  cout << "Нашли равные!";
else
  cout << "Равных нет.";
```

Чтобы правильно выводить на экран русские буквы, можно подключить уже известную вам библиотеку TX Library. При запуске программы она автоматически настроит консоль для работы с русскими буквами.

Сложные условия

Сложные условия в языке C++ строятся из простых условий (отношений) с помощью логических операций И, ИЛИ и НЕ. Этих операций достаточно, чтобы построить любое условие, поэтому они называются базовыми или *базисными*.

Операция И в C++ записывается как `&&`, операция ИЛИ — как `||`, а операция «НЕ» — как `!`. Однако стандарт языка разрешает использовать для этих операций и другие, более удобные обозначения: `and`, `or` и `not` (такие же, как и в Python).

Проверим, является ли значение *a* двузначным числом. Для этого нужно, чтобы одновременно были выполнены два условия: *a* больше, чем 9, и меньше, чем 100. Используем операцию И:

```
if (9 < a  and  a < 100)
  cout << "О, двузначное число!";
else
  cout << "Ну вот. Не двузначное число.";
```

Можно было поступить иначе: число *не* двузначное, если оно меньше 10 или больше 99. Тут уже нужна операция ИЛИ — выполнение одного из двух условий:

```
if (a < 10  or  a > 99)
  cout << "Снова не двузначное число... :(";
else
  cout << "Наконец-то двузначное число!";
```

Какой же из этих двух вариантов выбрать? Всегда выбирайте тот, который понятнее для вас, который легче прочесть, в котором меньше шансов сделать ошибку.

Операция НЕ применяется тогда, когда легко построить обратное условие. Допустим, нужно вывести сообщение, если точка *не* попала внутрь заштрихованного треугольника (рис. 2.1).

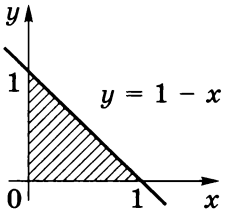


Рис. 2.1

Сначала запишем сложное условие, которое определяет, что точка с координатами (x, y) находится в треугольнике. Для этого должны выполняться три условия:

- точка не ниже оси OX, т. е. $y \geq 0$;
- точка не левее оси OY, т. е. $x \geq 0$;
- точка не выше линии $y = 1 - x$, т. е. $y \leq 1 - x$.

Все условия должны выполняться одновременно, т. е. объединить их нужно с помощью операции И. А если к этому условию применить операцию НЕ, то как раз и получится нужное нам условие — «точка не в треугольнике»:

```
if( not(y >= 0 and x >= 0 and y <= 1-x) )
  {
  /* точка не в треугольнике,
     что-то делаем... */
  }
```

Конечно, можно избавиться от операции НЕ, но делать это нужно очень аккуратно, помня про законы логики (в частности, про закон де Моргана). В нашем случае требуется не только заменить каждое из трёх условий на обратное, но и использовать операции ИЛИ вместо И:

```
if( y < 0 or x < 0 or y > 1-x )
  {
  /* точка не в треугольнике,
     что-то делаем... */
  }
```

Операции в сложном условии в C++ выполняются в том же порядке, что и в языке Python:

1) операции в скобках;
2) операции НЕ;
3) арифметические операции (+, -, *, /, %);
4) операции сравнения (<, <=, >, >=, ==, !=);
5) операции И;
6) операции ИЛИ.

Изменить порядок действий можно с помощью круглых скобок.

При анализе сложных условий в C++ используются «ленивые вычисления»: как только стал известен результат, работа прекращается. Рассмотрим условный оператор:

```
if( a != 0 and b/a > 10 ) {
  ...
  }
```

Если `a = 0`, то первое из двух условий ложно, и поэтому всё условие тоже ложно, независимо от истинности второго условия. В этом случае второе условие не будет проверяться. Заметим, что здесь такую проверку делать *нельзя*: произойдёт деление на ноль и программа аварийно завершится.

При использовании операции ИЛИ достаточно, чтобы одно из условий было истинно:

```
if( a != 0 or b != 0 ) {
  ...
  }
```

Поэтому при ненулевом значении a проверка условия $b \neq 0$ не выполняется.

Пример

Абоненты телефонной станции по сниженному тарифу оплачивают все звонки в выходные дни, а также в рабочие дни в вечернее и ночное время (с 18^{00} до 7^{00}).

В переменной day записан номер дня (1 — понедельник, 7 — воскресенье), а в переменной hour — время (в часах), когда был сделан звонок. На телефонной станции нужно определить, какой тариф использовать — обычный или сниженный.

В этой задаче достаточно сложное условие, и для того чтобы сделать программу более понятной, имеет смысл разбить его на части и ввести логические переменные.

Пусть логическая переменная isHoliday принимает значение true, если это выходной день (т. е. значение переменной day равно 6 или 7):

```
bool isHoliday = (day > 5);
```

Введём вторую логическую переменную: nightHour, которая принимает значение true, если время звонка — вечернее или ночное, то есть значение переменной hour больше или равно 18 (вечер) или меньше или равно 6 (раннее утро):

```
bool nightHour = (hour >= 18 or hour <= 6);
```

Теперь можно записать нужное условие:

```
if ( isHoliday or (not isHoliday and nightHour) )
  cout << "Сниженный тариф!";
else
  cout << "Поздравляем! У вас полный тариф.";
```

Это условие можно ещё упростить. Ведь вечером и ночью в выходные дни тоже действует сниженный тариф, поэтому условие **not** isHoliday можно не писать:

```
if( isHoliday or nightHour )
  cout << "Извините, сниженный тариф!";
else
  cout << "Наконец-то! Полный тариф.";
```

В таком виде программа читается значительно легче, чем если бы мы написали все три условия после **if** в виде одного сложного условия.

Выводы

- Условие в условном операторе **if-else** всегда записывается в круглых скобках.
- Если при истинности (или ложности) условия нужно выполнить несколько операторов, они объединяются в составной оператор (блок) с помощью фигурных скобок.
- В обе ветви условного оператора можно вставлять любые операторы, в том числе и другие (вложенные) условные операторы.
- В логической переменной можно хранить логические значения: `true` (истина) или `false` (ложь).
- Простые условия (отношения) в сложном условии не обязательно брать в скобки.
- Операции И, ИЛИ и НЕ в сложных условиях обозначаются **and**, **or** и **not** соответственно.

Вопросы и задания

1. Напишите программу, которая вводит три целых числа, записывает их в переменные `a`, `b` и `c` и переставляет значения переменных в памяти так, чтобы оператор
```
cout << a << " " << b << " " << c;
```
всегда выводил их в порядке возрастания (неубывания).

2. Программист написал программу для выбора наименьшего из двух чисел так:
```
if (a < b) min = a;
if (b < a) min = b;
```
В каких случаях эта программа будет работать неправильно? Запишите программу правильно, используя один условный оператор в полной форме.

3. Определите, результат какой математической операции выводит на экран эта программа:
```
if (x < 0)
  cout << -x;
else
  cout << x;
```

4. Определите, чему будет равно значение переменной `x` после выполнения этого фрагмента программы:
```
int x = 5, y = 2;
if (x < y)
  y = x + 2;
  x += y;
```
Проверьте свой ответ с помощью программы, объясните результат.

5. Программисту нужно было записать в переменную found значение true, если среди значений a, b и c есть хотя бы два равных. Что неверно в его программе?
```
bool found;
found = (a == b);
found = (a == c);
found = (b == c);
```
6. Напишите программу, которая определяет знак введённого целого числа, т. е. выводит –1 для отрицательных чисел, 0 для нуля и 1 для положительных чисел.
7. Напишите программу, которая вводит целое число и выводит ответ «да», если оно нечётное. Проверьте, чтобы она работала и для отрицательных чисел.
8. Напишите программу, которая вводит четырёхзначное число и выводит ответ «да», если оно является палиндромом. Палиндром — это число, которое не меняется, если его цифры записать в обратном порядке, например: 1221.
9. Покажите, что приведённая программа не всегда верно определяет минимальное из трёх чисел, записанных в переменные a, b и c:
```
if( a < b ) min = a;
else        min = b;
if( c < b ) min = c;
```
Приведите контрпример, т. е. значения переменных, при которых в переменной min будет получен неверный ответ. Как нужно доработать программу, чтобы она всегда работала правильно?
10. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение true (истина), если среди введённых чисел есть хотя бы одно отрицательное.
11. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение true (истина), если все эти числа двузначные.
12. С клавиатуры вводятся три числа, которые обозначают длины верёвок. Напишите программу, которая записывает в логическую переменную значение true (истина), если можно соединить две из трёх верёвок так, чтобы длина полученной верёвки была не менее 25 м (расход верёвки на связывание не учитывать).
13. *Проект.* Постройте свою небольшую экспертную систему по интересующей вас тематике.
14. Запишите на языке C++ условия, которые истинны, если:
    а) значение x принадлежит отрезку $[a; b]$;
    б) значение x *не* принадлежит отрезку $[a; b]$;
    в) оба значения x и y положительные;
    г) значения x и y одного знака (оба положительные либо оба отрицательные);

д) значения x и y имеют противоположные знаки (одно из них положительное, другое — отрицательное);
е) из значений x и y хотя бы одно равно нулю;
ж) из значений x и y одно равно нулю, а второе не равно нулю.

15. Что будет выведено на экран после выполнения программы при a = 2?

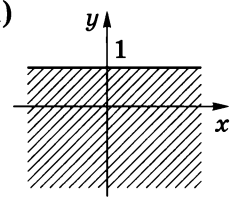
```
if ( a == 1 and a == 2 and a == 3 )
  cout << "Да!";
else
  cout << "Хорошая попытка! Но нет.";
```

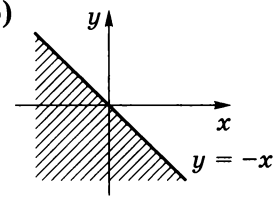
16. Напишите программу, которая вводит с клавиатуры три целых числа и записывает в логическую переменную значение true (истина), если:
а) среди введённых чисел есть хотя бы одно с цифрой 1 на конце;
б) все введённые числа заканчиваются на 1.

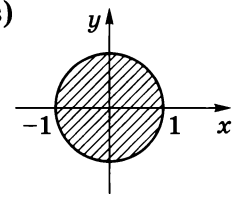
*17. Напишите программу, которая вводит возраст человека (целое число, не превышающее 120) и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «25 лет».

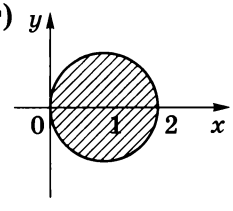
**18. Напишите программу, которая вводит целое число, не превышающее 100, и выводит его прописью, например: 21 → «двадцать один».

19. Напишите программу, которая вводит координаты точки на плоскости и определяет, попала ли эта точка в заштрихованную область.

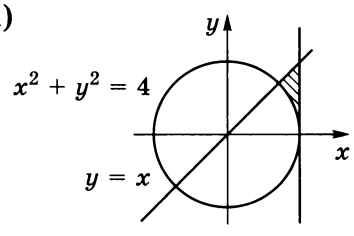
а) 

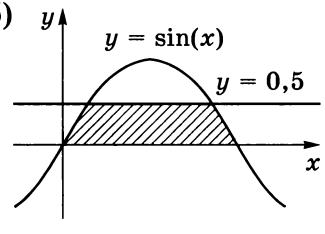
б) 

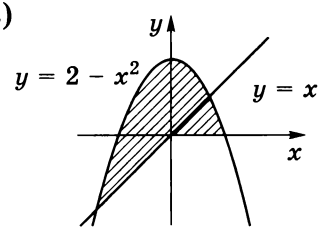
в) 

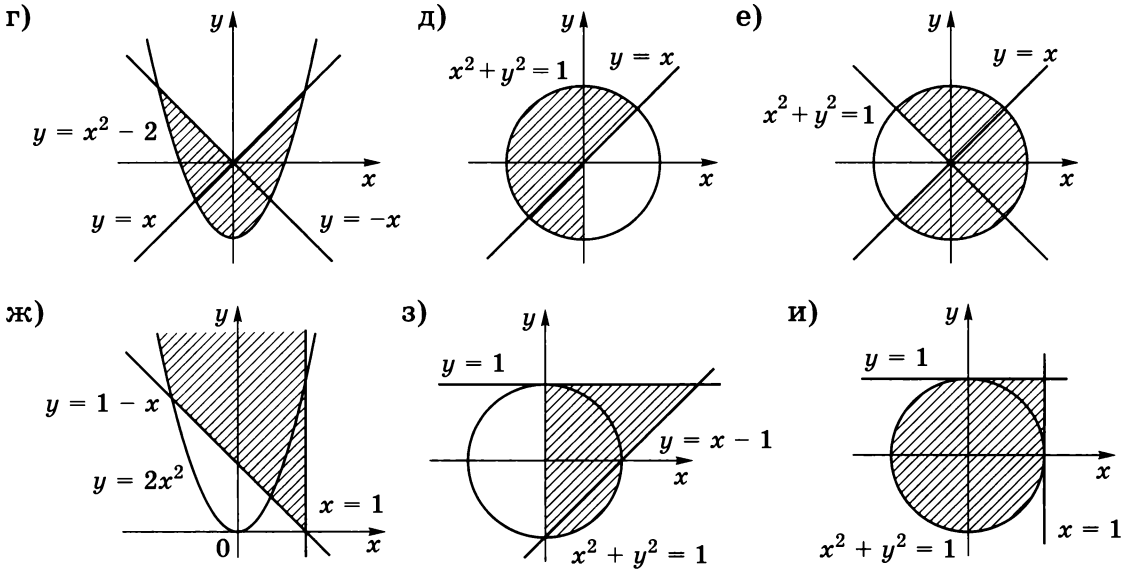
г) 

*20. Напишите программу, которая вводит координаты точки на плоскости и определяет, попала ли эта точка в заштрихованную область.

а) 

б) 

в) 



§ 21
Циклы

Ключевые слова:

- цикл с предусловием
- цикл с постусловием
- цикл по переменной

Цикл с предусловием

Изучая язык Python, мы познакомились с двумя видами циклов: циклами с условием и циклами по переменной. Причём в циклах с условием в Python условие проверяется при входе в цикл, такой цикл называется *циклом с предусловием* (т. е. с предварительной проверкой условия).

Цикл с предусловием есть и в языке C++. Например, вывести 10 раз слово «crocodile» можно с помощью такого цикла:

```
int count = 0;
while( count < 10 ) {   // заголовок цикла
  cout << "crocodile" << endl;
  count ++;
  }
```

Здесь переменная `count` — это счётчик повторений цикла. Условие после слова **`while`** обязательно заключается в скобки.

Если в теле цикла нужно выполнить несколько операторов, они объединяются в составной оператор (блок) — заключаются в фигурные скобки. Отступы никак не влияют на работу программы на C++.

Если убрать фигурные скобки, то в теле цикла останется только оператор вывода, а команда `count++` (увеличение переменной `count` на единицу) будет выполняться только один раз после окончания цикла. Но такой цикл никогда не закончится (подумайте почему). К сожалению, эту ошибку сложно обнаружить: с точки зрения синтаксиса (правил записи команд) языка C++ в этой программе всё верно.

Циклы с постусловием

Цикл, в котором проверка условия выполняется в конце очередной итерации (перед выполнением следующей), называется *циклом с постусловием*. Общая форма такого цикла:

```
do {
  ...
  }
while(условие);
```

Тело цикла, обозначенное многоточием, будет выполняться многократно, пока условие в скобках после слова **while** остаётся истинным.

У такого цикла есть важная особенность: поскольку условие проверяется *после* очередной итерации, цикл всегда выполняется хотя бы один раз. Это может быть нужно, например, при вводе данных с проверкой правильности ввода.

Пусть требуется ввести год рождения человека, причём допустимыми будем считать значения от 1900 до 2018 включительно. Фрагмент программы, в котором вводится год рождения с проверкой, выглядит так:

```
int yearBirth;
bool isValid;
do {
  cout << "Введите год рождения: ";
  cin >> yearBirth;
  isValid = (1900 <= yearBirth and
                     yearBirth <= 2018);
  if ( not isValid )
    cout << "Неверный год рождения! Ай-ай-ай!"
         << endl;
  }
while( not isValid );
```

Переменная `yearBirth` используется для хранения года рождения, а логическая переменная `isValid` принимает значение `true` (истина), если введено правильное значение. Если введённое значение неверно, эта переменная становится равной `false`, и выводится сообщение об

ошибке. Кроме того, условие **not** isValid в последней строке цикла (после слова **while**) истинно, поэтому цикл повторяется.

Как только будет введено допустимое значение, переменная isValid станет истинной, условие **not** isValid — ложным, и цикл закончится.

Можно ли обойтись без цикла с постусловием? Да, можно, но придётся использовать «обходной манёвр». В нашем примере можно, например, написать цикл с предусловием так:

```
int yearBirth;
bool isValid = false;
while( not isValid ) {
  ...
  }
```

Тело цикла, обозначенное многоточием, — такое же, как и в предыдущем варианте программы.

Для того чтобы цикл выполнился в первый раз, нам обязательно надо сделать так, чтобы при входе в цикл условие **not** isValid было истинно. Для этого значение переменной isValid должно быть равно false, мы присвоили его при объявлении переменной. Программист может легко забыть о том, что надо записать такое начальное значение, и тогда цикл сработает неверно.

Ещё один вариант решения этой задачи — бесконечный цикл с досрочным выходом:

```
while( true ) {
  cout << "Введите год рождения: ";
  cin >> yearBirth;
  isValid = (1900 <= yearBirth and
                    yearBirth <= 2018);
  if ( isValid ) break;
  cout << "Неверный год рождения! Ай-ай-ай!"
       << endl;
  }
```

Условие цикла true — это всегда истинное условие. Поэтому выйти из такого цикла можно только с помощью специального оператора, который, как и в языке Python, называется **break**. Как только вводится правильное значение, переменная isValid принимает значение true, и происходит выход из цикла (этот оператор выделен в программе фоном).

В языке C++ любое значение, отличное от false, 0 и специальной величины NULL, воспринимается как истинное условие. Поэтому часто используют такую запись бесконечного цикла:

```
while( 1 ) {
  ...
  }
```

Вычисление квадратного корня

Один из самых известных алгоритмов вычисления квадратного корня, придуманный ещё в Древней Греции, — метод Герона Александрийского, который сводится к многократному применению формулы

$$x_i = \frac{1}{2}\left(x_{i-1} + \frac{a}{x_{i-1}}\right).$$

Здесь a — это число, из которого извлекается корень, а x_{i-1} и x_i — предыдущее и следующее *приближения* (неточные значения корня). Фактически здесь вычисляется среднее арифметическое чисел x_{i-1} и (a/x_{i-1}). Пусть одно из этих значений меньше, чем $\sqrt{a}$, тогда второе обязательно больше, чем $\sqrt{a}$. Поэтому их среднее арифметическое с каждым шагом приближается к значению корня.

Метод Герона «сходится» (т. е. приводит к правильному решению) при любом начальном приближении x_1 (не равном нулю). Например, можно выбрать начальное приближение $x_1 = a$.

Метод Герона — это метод, в котором используются циклические вычисления, или *итерации*. Такой метод называется *итерационным*.

Когда же нужно остановить итерации? Численные (приближённые) методы решения задач не позволяют найти точный ответ. Поэтому обычно в таких случаях задаётся допустимая ошибка — малая величина ε, и задача ставится так: найти значение x, которое отличается от неизвестного точного решения x^* не более чем на ε.

В начале программы вводим с клавиатуры значение a, для которого нужно вычислить квадратный корень:

```
float a;
cin >> a;
```

Затем объявляем необходимые переменные:

```
float x = a, x0 = 0;
float eps = 0.00001;
```

В переменных `x` и `x0` будем хранить текущее и предыдущее приближения, для них задаются начальные значения: $x_{i-1} = 0$ и $x_i = a$. В переменную `eps` запишем заданную допустимую ошибку ε.

Мы остановим цикл, когда разница между последним и предпоследним приближениями станет по модулю не больше, чем ε:

```
while( abs(x-x0) > eps ) {
  x0 = x;                        // (1)
  x = (x + a/x) / 2;             // (2)
  }
```

Стандартная функция `abs` вычисляет модуль переданного ей числа — разности `x – x0`. Цикл выполняется, пока эта разность по

модулю остаётся больше, чем ε. Для использования функции abs нужно подключить математическую библиотеку cmath:

```
include <cmath>
```

В теле цикла в строке (1) запоминаем текущее приближение в переменной x0 (для следующего шага оно будет предыдущим), и в строке (2) вычисляем новое приближение по формуле Герона.

После окончания цикла выводим результат на экран:

```
cout << x;
```

Для проверки правильности расчётов можно сравнить полученный результат с результатом вызова стандартной функции sqrt(a). Если вычисления выполнены верно, два значения должны отличаться не более чем на ε.

Отметим ещё одну особенность нашей программы. Можно задать даже нулевое значение допустимой ошибки ε (eps=0), и программа всё равно будет работать правильно. В этом случае цикл остановится тогда, когда разница между x_i и x_{i-1} станет по модулю меньше, чем машинная точность хранения вещественных чисел (см. § 6), т. е. значения x_i и x_{i-1} станут одинаковыми с точки зрения компьютера.

Циклы по переменной

Циклы по переменной в языке C++ так же, как в Python, начинаются со служебного слова for. Например, чтобы 10 раз вывести слово «hippo», можно использовать такой цикл:

```
for( int k=0; k<10; k++ )
  cout << "hippo" << endl;
```

Здесь переменная цикла — это целая переменная k, она объявляется прямо в заголовке цикла **for**.

Круглые скобки после слова **for** содержат три части, разделённые точками с запятой (а не запятыми!). Первая часть (**int** k=0) выполняется один раз до входа в цикл: в переменную k записывается начальное значение 0. Вторая часть (k<10) — это условие выполнения цикла; цикл заканчивает работу, как только это условие станет ложным. Третья часть (k++) выполняется каждый раз в конце очередного повторения тела цикла. Таким образом, в нашем цикле переменная k с каждым повторением тела цикла увеличивается на единицу и последовательно «проходит» все целые значения от 0 до 9.

После завершения работы цикла переменная k, объявленная в заголовке цикла, не определена, обращение к ней приведёт к ошибке. Её область видимости — только команда **for**.

Циклы по переменной удобно использовать для перебора вариантов. Найдём все двузначные *автоморфные* числа. Автоморфным называется число, запись которого — это последние цифры его квадрата (например, $25^2 = 6\underline{25}$).

В цикле переберём все двузначные числа от 10 до 99 включительно. Для каждого проверяем условие автоморфности: последние две цифры квадрата числа (т. е. остаток от деления этого квадрата на 100) совпадают с самим числом:

```
for( int k=10; k<=99; k++ ) {
  int squareK = k*k;
  if( squareK % 100 == k )
    cout << k << endl;
  }
```

Здесь в тело цикла входит несколько команд, поэтому они объединяются в составной оператор фигурными скобками. Переменная `squareK` — это *локальная* переменная цикла. Область видимости этой переменной — тело цикла, за пределами цикла её использовать нельзя (это вызовет ошибку трансляции).

Закон изменения переменной цикла задаётся третьей частью заголовка цикла и может быть любым. Например, можно перебрать все нечётные значения от 99 до 1 и вывести их квадраты:

```
for( int k=99; k>0; k-=2 )
  cout << k*k << endl;
```

В отличие от многих языков программирования переменная цикла в C++ может быть не только целочисленной, но и, например, вещественной. Следующий цикл выводит значения квадратов вещественных чисел:

```
for( float x=1.5; x<=2; x +=0.1 )
  cout << x << " " << x*x << endl;
```

Выводы

- В языке C++ есть два вида циклов с условием — циклы с предусловием и циклы с постусловием.
- В цикле с предусловием проверка условия выполняется в начале очередной итерации цикла, в цикле с постусловием — в конце очередной итерации.
- Цикл с постусловием всегда выполняется хотя бы один раз, цикл с предусловием может не выполниться ни разу.
- Для досрочного выхода из цикла используют оператор **break**.
- В языке C++ есть цикл по переменной. В заголовке такого цикла (цикла **for**) задаются: начальное значение переменной цикла, условие продолжения цикла и правило изменения переменной цикла.
- Переменная цикла в C++ может быть как целой, так и вещественной.

Вопросы и задания

1. Найдите и исправьте ошибку в программе.

```
k = 0;
while( k < 10 )
  cout << "crocodile";
  k++;
```

2. Верны ли следующие утверждения для языка C++:
   а) любой цикл **for** можно переписать как цикл с условием;
   б) любой цикл с предусловием можно переписать как цикл **for**?
   Если вы отвечаете на какой-то вопрос «да», предложите алгоритм такого преобразования.
3. Используя дополнительные источники, выясните, что означают специальные символы `'\t'` и `'\n'`.
4. Напишите программу, которая вводит с клавиатуры вещественное число и вычисляет его квадратный корень с помощью формулы Герона. Сравните результат ваших расчётов и результат стандартной функции `sqrt` из библиотеки `cmath`.
5. Напишите программу, которая выводит на экран все трёхзначные автоморфные числа.
6. Натуральное число называется *числом Армстронга*, если сумма цифр числа, возведённых в N-ю степень, где N — количество цифр в числе, равна самому числу. Например, $153 = 1^3 + 5^3 + 3^3$. Найдите все трёхзначные числа Армстронга.
7. Ряд чисел Фибоначчи задается следующим образом: первые два числа равны 1 ($F_1 = F_2 = 1$), а каждое следующее равно сумме двух предыдущих: $F_n = F_{n-1} + F_{n-2}$. Напишите программу, которая вводит натуральное число N и выводит на экран первые N чисел Фибоначчи.

**8. Напишите программу, которая определяет, сколько чисел из входной последовательности натуральных чисел являются числами Фибоначчи. Программа получает на вход количество чисел в последовательности, а затем — сами числа.

*9. Напишите программу, которая в последовательности натуральных чисел определяет количество простых чисел. Программа получает на вход количество чисел в последовательности, а затем — сами числа.

10. Используя библиотеку `TX Library`, выполните задания 5–9 к § 13 на языке C++.
11. Напишите программу, которая в квадрате со стороной 400 пикселей перекрашивает в случайные цвета 500 пикселей со случайными координатами. Подсчитайте, сколько точек попадёт в каждый из четырёх квадратов со стороной 200 пикселей, на которые можно разбить исходный квадрат.

Интересный сайт

cyberforum.ru — форум программистов и специалистов по компьютерной технике

§ 22
Анимация

Ключевые слова:

- анимация
- процедура
- пауза
- нажатие клавиши

Принципы анимации

Вы уже знаете, что анимация основана на быстром изменении картинки на экране: если изображение меняется чаще, чем 16 раз в секунду, человеческий глаз не успевает реагировать на каждое изменение и видит плавное движение.

Используя библиотеку TX Library, мы будем строить анимацию самостоятельно, программируя все её этапы.

Рассмотрим уже известную задачу: движение шарика (или любой другой фигуры) на фоне. Пусть шарик расположен в точке холста с координатами (x, y), и нужно переместить его в точку с координатами $(x + 5, y)$. Для этого нужно:

1) «стереть» шарик, т. е. восстановить фон в том месте, где сейчас нарисован шарик;
2) изменить положение шарика (его координаты на холсте);
3) запомнить участок фона, который будет испорчен при выводе шарика в новом месте;
4) вывести изображение шарика поверх фона.

Мы будем использовать одноцветный фон, поэтому запоминать изображение, перекрытое шариком, нам не нужно. Чтобы стереть шарик, достаточно залить это место цветом фона.

В начале программы нужно подключить библиотеку TX Library:

```
#include "TXLib.h"
```

Рисуем шарик

Шарик будем изображать в виде круга жёлтого цвета, контур сделаем прозрачным. Базовой точкой, определяющей *текущие координаты* шарика (т. е. координаты в данный момент времени), удобно считать центр окружности. Обозначим его координаты через xCenter и yCenter, а радиус шарика — через R.

Рисование шарика состоит из трёх команд:

```
txSetColor( TX_TRANSPARENT );
txSetFillColor( TX_YELLOW );
txCircle( xCenter, yCenter, R );
```

В первой команде выбираем прозрачный цвет контура, во второй — жёлтый цвет заливки, с помощью третьей рисуем круг. Лучше офор-

мить эти действия в виде процедуры, которая принимает координаты центра шарика и его радиус:

```
void drawBall(int xCenter, int yCenter, int R)
  {
  txSetColor( TX_TRANSPARENT );
  txSetFillColor( TX_YELLOW );
  txCircle( xCenter, yCenter, R );
  }
```

Как мы уже выяснили, чтобы стереть шарик на синем фоне, достаточно нарисовать его синим цветом в том же месте. Получается такая процедура стирания:

```
void clearBall( int xCenter, int yCenter, int R )
  {
  txSetColor( TX_TRANSPARENT );
  txSetFillColor( TX_BLUE );
  txCircle( xCenter, yCenter, R );
  }
```

Начальное положение

Сначала создаём графическое окно:

```
txCreateWindow( 400, 400 );
```

Рисуем фон — синий квадрат:

```
txSetFillColor( TX_BLUE );
txRectangle( 0, 0, 400, 400 );
```

Шарик будем двигать слева направо по середине холста на уровне $y = 200$ (рис. 2.2).

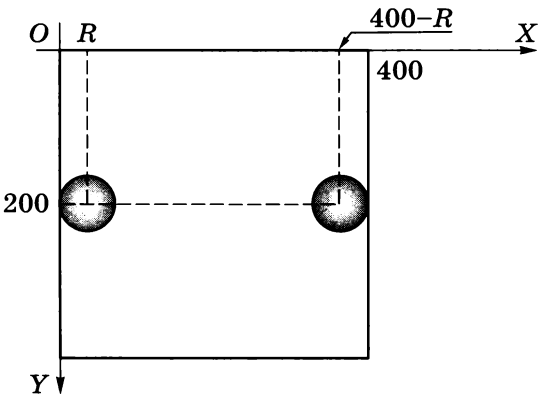


Рис. 2.2

Когда шарик находится вплотную к левой границе, его x-координата равна радиусу R, а у правой границы xCenter = 400 – R.

Задаём размер и начальное положение шарика в программе:

```
int R = 10; // радиус шарика
int xCenter = R, yCenter = 200;
```

Анимация движения

Анимация может быть построена в виде такого цикла:

```
while( /*шарик не вышел за границу холста*/ ) {
  // рисуем шарик
  // пауза
  // стираем шарик
  // двигаем шарик (меняем координаты)
  }
```

Пока такая программа работать не будет: все действия и условие в заголовке цикла записаны в виде комментариев. Такая запись называется *псевдокодом*, он помогает понять и построить логику программы. Но нам нужно переписать псевдокод на языке C++.

Обратите внимание, что между рисованием и стиранием нужно сделать небольшую паузу (например, длиной в 20 миллисекунд). Если паузы не будет, мы не сможем увидеть движение — шарик проскочит всю ширину холста раньше, чем мы успеем что-то заметить.

Движение шарика сводится к тому, что мы меняем его координаты — переменные `xCenter` и `yCenter`. Шарик во время изменения координат не должен быть виден, иначе мы будем стирать его не в том месте, гда он был нарисован, и за ним на холсте останется след.

Для рисования шарика будем вызывать процедуру `drawBall`, а для стирания — процедуру `clearBall`. Паузу нужно сделать с помощью функции `txSleep` (от английского *sleep* — спать) из библиотеки `TX Library`, она принимает один параметр – время паузы в миллисекундах.

Условие «шарик не вышел за границу холста» выразим через его x-координату: пока она не больше, чем $400 - R$, шарик находится в границах синего квадрата. В результате получаем такой цикл:

```
while( xCenter <= 400-R ) {
   drawBall( xCenter, yCenter, R );
   txSleep( 20 );
   clearBall( xCenter, yCenter, R );
   xCenter += 5;
   }
```

Последняя строка в теле цикла говорит о том, что за одну итерацию цикла шарик смещается на 5 пикселей вправо.

Собрать всю программу вы можете самостоятельно.

Обработка нажатия клавиши

Теперь попробуем остановить цикл анимации (и завершить работу программы) при нажатии на клавишу *Escape* (*Esc*) на клавиатуре. Для этого нужно добавить в тело цикла такой условный оператор (пишем пока условно, на псевдокоде):

```
if( /*нажата клавиша Escape*/ )
  break;
```

Как вы помните, оператор **break** выполняет досрочный выход из цикла.

Определить, нажата ли клавиша, можно с помощью функции `GetAsyncKeyState` из стандартной библиотеки операционной системы Windows. Эта функция проверяет состояние клавиши, код которой ей передали, и возвращает целое число: ноль, если заданная клавиша не нажата, и ненулевое значение, если нажата.

Код клавиши *Escape* имеет символьное обозначение `VK_ESCAPE`. Получается такой условный оператор, обеспечивающий выход при нажатии клавиши *Escape*:

```
if( GetAsyncKeyState(VK_ESCAPE) )
  break;
```

Эту команду нужно поместить в тело цикла, например сразу после заголовка цикла.

Выводы

- Анимация в программе — это многократное повторение следующих действий:
  1) рисование объекта;
  2) пауза (чтобы увидеть объект);
  3) стирание объекта;
  4) изменение координат (перемещение объекта).
- Проверить нажатие клавиши можно с помощью функции `GetAsyncKeyState`: она возвращает ненулевое значение, если клавиша с указанным кодом нажата.

Вопросы и задания

1. Можно ли переставить команду `txSleep` в программе анимации в другое место в теле цикла? Ответ обоснуйте.
2. Как можно изменить программу анимации, чтобы ускорить движение шарика? Предложите два варианта решения этой задачи и обсудите ограничения каждого из них.
3. Как завершится выполнение программы анимации, если оператор выхода по нажатию клавиши Escape поместить между вызовами процедур `drawBall` и `clearBall`?

4. Измените программу анимации так, чтобы контур шарика был чёрным (цвет TX_BLACK). Какие проблемы у вас возникли? Чем они вызваны? Как их решить? Выберите свои цвета для фона, контура и заливки шарика.
5. Выясните экспериментально, какое значение возвращает функция GetAsyncKeyState, если клавиша с указанным кодом нажата.
6. Доработайте программу анимации так, чтобы шарик после выхода за границы синего квадрата вновь появлялся с другой стороны холста.
7. Выполните рефакторинг (переработку кода) программы анимации, избавившись от «магических чисел» (5, 400): запишите их в переменные в самом начале основной программы и далее везде используйте эти переменные.
8. Напишите программу, в которой через поле (синий прямоугольник) движутся два квадрата навстречу друг другу.

*9. Доработайте предыдущую программу так, чтобы квадраты отталкивались от границ поля и начинали движение в противоположном направлении.

*10. Напишите программу, в которой шарик управляется с помощью клавиш-стрелок (их коды обозначаются как VK_LEFT, VK_RIGHT, VK_UP и VK_DOWN). Шарик не должен выходить за границы поля.

*11. Напишите программу, в которой шарик движется непрерывно и меняет направление при нажатии на клавиши-стрелки.

*12. Доработайте программу из предыдущего задания так, чтобы шарик отталкивался от границ поля.

ОГЛАВЛЕНИЕ