Глава I.

Введение в программирование

1. Простейшие программы

Зачем нужно программирование?

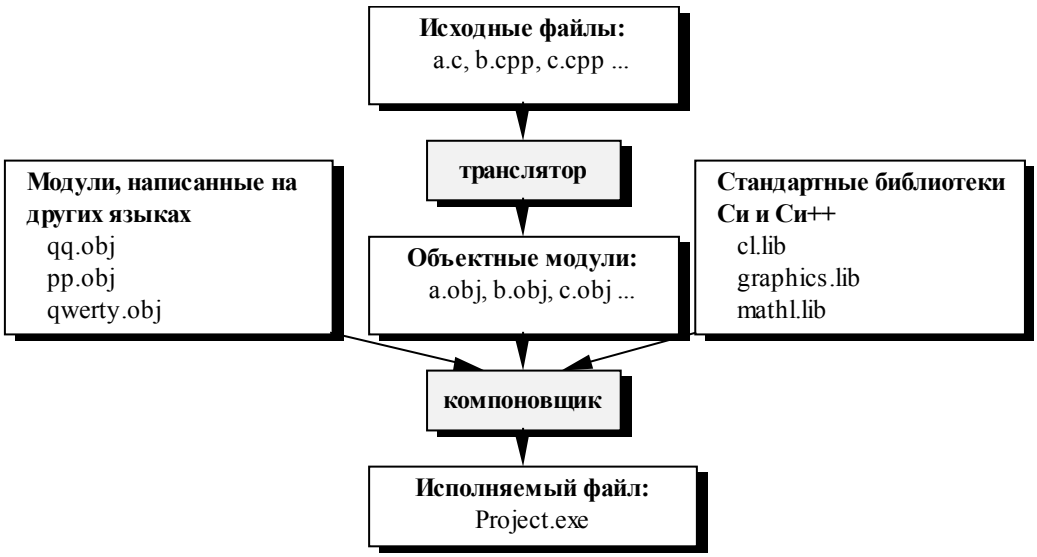
Иногда создается впечатление, что все существующие задачи могут быть решены с помощью готовых программ для компьютеров. Во многом это действительно так, но опыт показывает, что всегда находятся задачи, которые не решаются (или плохо решаются) стандартными средствами. В этих случаях приходится писать собственную программу, которая делает все так, как вы этого хотите (или нанимать за большие деньги умного дядю, который способен это сделать).

Два этапа создания программ

Программа на языке Си, также как и на большинстве современных языков, создается в два этапа

- ***трансляция*** программы - перевод текста вашей программы в машинные коды
- ***компоновка*** - сборка различных частей программы и подключение стандартных библиотек.

Схема создания программы на Си может быть изображена так



Почему же не сделать все за один шаг? Для простейших программ это действительно было бы проще, но для сложных проектов двухступенчатый процесс имеет явные преимущества:

- обычно сложная программа разбивается на несколько отдельных частей (***модулей***), которые отлаживаются отдельно и зачастую разными людьми; поэтому в завершении остается лишь собрать готовые модули в единый проект;
- при исправлении в одном модуле не надо снова транслировать все остальные (это могут быть десятки тысяч строк);
- на компоновке можно подключать модули, написанные на других языках, например, на Ассемблере (в машинных кодах).

Трансляторы языка Си являются ***компиляторами***, то есть они переводят (транслируют) сразу всю программу в машинный код, а не транслируют строчка за строчкой во время выполнения, как это делают ***интерпретаторы***, например, Бейсик. Это позволяет значительно

ускорить выполнение программы и не ставить интерпретатор на каждый компьютер, где программа будет выполняться.

Исходные файла программы на языке Си имеют расширение ***.c** или ***.cpp** (если в них использованы специальные возможности Си++ — расширения языка Си). Это обычные текстовые файлы, которые содержат текст программы. Транслятор переводит их в файлы с теми же именами и расширением ***.obj** — это так называемые ***объектные файлы.*** Они уже содержат машинный код для каждого модуля, но еще не могут выполняться - их надо связать вместе и добавить библиотеки стандартных функций (они имеют расширение ***.lib**). Это делает компоновщик, который умеет также включать в программу файлы ***.obj,** исходный код которых был написан на других языках. В результате получается один файл ***.exe,** который и представляет собой готовую программу.

Простейшая программа на Си

Такая программа состоит всего из 12 символов, но заслуживает внимательного рассмотрения. Вот она:

```
void main()
{
}
```

Основная программа в Си всегда называется именем `main` (будьте внимательны -Си различает большие и маленькие буквы, а все стандартные операторы Си записываются маленькими буквами). Пустые скобки означают, что `main` не имеет аргументов, а слово `void` (пустой) говорит о том, что она также и не возвращает никакого значения, то есть, является ***процедурой***. Фигурные скобки обозначают начало и конец процедуры `main` - поскольку внутри них ничего нет, наша программа ничего не делает, она просто соответствует правилам языка Си, ее можно скомпилировать и получить **exe-**файл.

Вывод текста на экран

Составим теперь программу, которая делает что-нибудь полезное, например, выводит на экран слово «Привет».

```
#include <stdio.h>
void main()
{
printf("Привет");
}
```

подключение функций стандартного ввода и вывода, описание которых находится в файле **stdio.h**

вызов функции вывода на экран

Что новенького?

Перечислим новые элементы, использованные в этой программе:

- Чтобы использовать стандартные функции, необходимо сказать транслятору, что есть функция с таким именем и перечислить тип ее аргументов - тогда он сможет определить, верно ли мы ее используем. Это значит, что надо подключить ***описание*** этой функции. Описания стандартных функций Си находятся в так называемых ***заголовочных файлах*** с расширением ***.h** (в каталоге ***C:\BORLANDC\INCLUDE***)

- Для подключения заголовочных файлов используется директива (команда) ***препроцессора*** "***#include***", после которой в угловых скобках ставится имя файла заголовка (***Препроцессор***

- это специальная программа, которая обрабатывает текст вашей программы раньше транслятора. Все команды препроцессора начиняются знаком "#"). Внутри угловых скобок не должно быть пробелов. Для подключения еще каждого нового заголовочного файла надо использовать новую команду "***#include***".

- Для вывода информации на экран используется функция `printf`. В простейшем случае она принимает единственный аргумент - строку в кавычках, которую надо вывести на экран.
- Каждый оператор языка Си заканчивается точкой с запятой.

Как запустить программу?

Чтобы проверить эту программу, надо сначала "напустить" на нее транслятор, который переведет ее в машинные коды, а затем компоновщик, который подключит стандартные функции и создаст исполняемый файл. раньше все это делали, вводя команды в командной строке или с помощью так называемых пакетных файлов. На современном уровне все этапы создания, трансляции, компоновки, отладки и проверки программы объединены и выполняются внутри специальной программы-оболочки, которую называют ***интегрированная среда разработки*** (***IDE*** - *integrated development environment*). В нее входят

- редактор текста
- транслятор
- компоновщик
- отладчик

В этой среде вам достаточно набрать текст программы и нажать на одну клавишу, чтобы она выполнилась (если нет ошибок).

Для запуска оболочки надо набрать в командной строке[1]

bc

На экране появляется окно оболочки и, скорее всего, последняя программа, с которой в ней работали. Чтобы закрыть все лишние окна, надо нажимать на клавиши ***Alt-F3*** до тех пор, пока весь экран не станет серого цвета - это значит, что все окна убраны с рабочего стола.

Теперь нажмите клавишу ***F3*** и введите имя новой программы (не более 8 символов) и нажмите клавишу ***Enter.*** Будет открыто новое пустое окно, в котором надо ввести текст программы.

Когда вы набрали всю программу, нажмите клавишу ***F2***, чтобы сохранить ее на диске. Рекомендуется делать это чаще на случай отключения питания или сбоя компьютера.

После этого можно запустить программу на выполнение, нажав клавиши ***Ctrl-F9***. Если в ней есть ошибки, вы увидите в нижней части экрана окно ***Message*** (Сообщение), в котором перечислены ошибки (***Error***) и предупреждения (***Warning***), причем та строка, в которой транслятору что-то не понравилось, выделяется голубым цветом в окне программы. Активным сейчас является окно сообщений, если вы сдвигаете в нем курсор (зеленую строку) стрелками вверх и вниз, то в окне программы также сдвигается голубая полоса — вы перешли к другой ошибке и увидели строку, в которой машина ее обнаружила. При поиске ошибок надо помнить, что

- часто реальная ошибка заключена не в выделенной строке, а в предыдущей - проверяйте и ее тоже
- часто одна ошибка вызывает еще несколько, и появляются так называемые наведенные ошибки
- самые распространенные ошибки:

[1] Для специалистов: каталог, в котором расположен файл ***bc.exe*** должен быть включен в системный путь ***PATH***, определяемый в файле ***autoexec.bat***.

Unable to open include file 'xxx.h'	не найден заголовочный файл 'xxx.h' (неверно указано его имя, он удален или т.п.)
Function 'xxx' should have a prototype	функция 'xxx' не объявлена (не подключен заголовочный файл или не объявлена своя функция, или неверное имя функции)
Unterminated string or character constant	не закрыты кавычки
Statement missing ;	нет точки с запятой в конце оператора
Compound statement missing }	не закрыта фигурная скобка
Undefined symbol 'xxx'	не объявлена переменная 'xxx'

Основные клавиши оболочки Borland C

F1	помощь (справочная система)
F2	запись файла в активном окне на диск
F3	чтение файла с диска или создание нового файла
Alt-F3	закрыть активное окно
F5	раскрыть окно на весь экран
Ctrl-F9	запуск программы на выполнение
Alt-F5	показать рабочий экран
Alt-X	выход из оболочки

Работа с блоками текста

Часто надо скопировать, удалить или переместить часть текста. Для этого есть специальные клавиши работы с блоками. Выделенный блок помечается в редакторе серым цветом.

Ctrl-Y	удалить строку, в которой стоит курсор
Shift-стрелки	расширить выделенный блок
Ctrl-Del, Shift-Del	удалить выделенный блок
Ctrl-Insert	запомнить выделенный блок в буфере
Shift-Insert	вставить блок из буфера
Ctrl-K + B	установить начало блока
Ctrl-K + K	установить конец блока
Ctrl-K + L	выделить текущую строку (в которой стоит курсор)
Ctrl-K + C	скопировать выделенный блок выше курсора
Ctrl-K + V	переместить выделенный блок выше курсора
Ctrl-K + H	отменить выделение блока

Остановим мгновение

Если запускать эту программу из оболочки *Borland C,* то обнаружится, что программа сразу заканчивает работу и возвращается обратно в оболочку, не дав посмотреть результат ее

работы на экране. Бороться с этим можно так - давайте скажем компьютеру, что в конце работы надо дождаться нажатия любой клавиши.

```
#include <stdio.h>
#include <conio.h>
void main()
{
printf("Привет");      // вывод на экран
getch();              /*ждать нажатия клавиши*/
}
```

подключение функций консольного ввода и вывода, описание которых находится в файле ***conio.h***

вызов функции, которая ждет нажатия на любую клавишу

Что новенького?

- Задержка до нажатия любой клавиши выполняется функцией `getch()`.
- Описание этой функции находится в заголовочном файле ***conio.h.***
- Знаки // обозначают начало ***комментария*** — все правее них до конца строки не обрабатывается компьютером и служит нам для пояснения программы.
- Комментарий также можно ограничивать парами символов **/*** (начало комментария) и ***/** (конец комментария). В этом случае комментарий может быть многострочный, то есть состоять из нескольких строк.

2. Переменные

Типы данных и переменные

Для обработки данных их необходимо хранить в памяти. При этом к этим данным надо как-то обращаться. Обычно люди обращаются друг к другу по имени, такой же способ используется в программировании: каждой ячейке памяти (или группе ячеек) дается свое собственное имя. Используя это имя можно прочитать информацию из ячейки и записать туда новую информацию.

Переменная - это ячейка в памяти компьютера, которая имеет имя и хранит некоторое значение. Значение переменной может меняться во время выполнения программы. При записи в ячейку нового значения старое стирается.

С точки зрения компьютера все данные в памяти - это числа (более точно - наборы нулей и единиц). Тем не менее, и вы (и компьютер) знаете, что с целыми и дробными числами работают по-разному. Поэтому в каждом языке программирования есть разные типы данных (переменных), для обработки которых используются разные методы. Основными данными в языке Си являются

- целые переменные (тип `int` - от английского *integer* - целый) занимают 2 байта в памяти
- вещественные переменные, которые могут иметь дробную часть (тип `float` - от английского *floating point* - плавающая точка), занимают 4 байта в памяти
- символы (тип `char` - от английского *character* - символ) занимают 1 байт в памяти

Для использования все переменные необходимо объявлять - то есть сказать компьютеру, чтобы он выделил на них ячейку памяти нужного размера и присвоил ей нужное имя. Переменные обычно объявляются в начале программы. Для объявления надо написать название типа переменных (`int`, `float` или `char`), а затем через запятую имена всех объявляемых переменных. При желании можно сразу записать в новую ячейку нужное число, как показано в примерах ниже. Если переменной не присваивается никакого значения, то в ней находится "мусор", то есть то, что было там раньше.

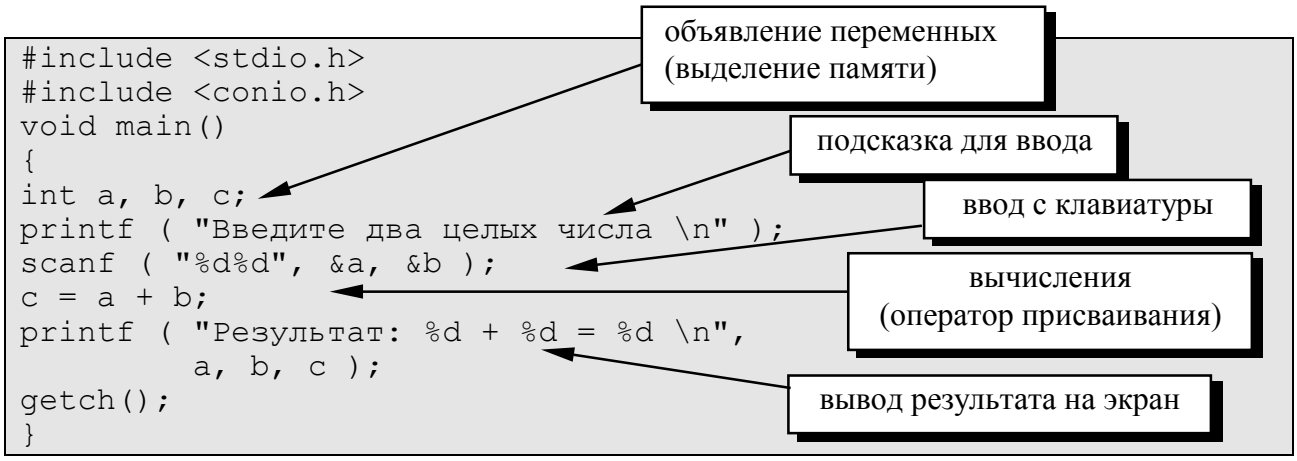
Примеры.

```
int a;                          // выделить память под целую
                                // переменную a

float b, c;                     // две вещественных переменных b и c

int Tu104, Il86=23, Yak42;      // три целых переменных, причем в Il86
                                //  сразу записывается число 23

float x=4.56, y, z;             // три вещественных переменных,
                                // причем в x сразу записывается число 4.56

char c, c2='A', m;              // три символьных переменных, причем в c2
                                // сразу записывается символ 'A'
```

Вычисление суммы двух чисел (ввод и вывод)

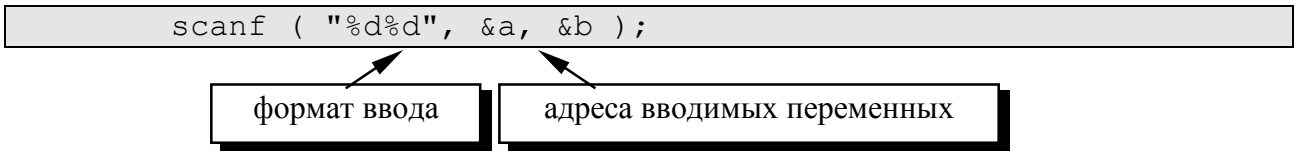
Задача. Ввести с клавиатуры два целых числа и вывести на экран их сумму.

Сразу запишем решение задачи на языке Си.



Что новенького?

- Программа чаще всего содержит 4 части:
  ⇒ объявление переменных
  ⇒ ввод исходных данных
  ⇒ обработка данных (вычисления)
  ⇒ вывод результата на экран или на печать
- Перед вводом данных необходимо вывести на экран подсказку (иначе компьютер будет ждать ввода данных, а пользователь не будет знать, что от него хочет машина).
- Символы `"\n"` в функции `printf` обозначают переход в начало новой строки.
- Для ввода данных используют функцию `scanf`.



⇒ Формат ввода - это строка в кавычках, в которой перечислены один или несколько форматов ввода данных:

%d	ввод целого числа (переменная типа `int`)
%f	ввод вещественного числа (переменная типа `float`)
%c	ввод одного символа (переменная типа `char`)

⇒ После формата ввода через запятую перечисляются **адреса** ячеек памяти, в которые надо записать введенные значения. Почувствуйте разницу:

a	значение переменной ***a***
&a	адрес переменной ***a***

⇒ Количество форматов в строке должно быть равно количеству адресов в списке. Кроме того, тип переменных должен совпадать с указанным: например, если ***a*** и ***b*** - целые переменные, то следующие вызовы функций ошибочны

`scanf ( "%d%d", &a );`	неясно, куда записывать второе введенное число
`scanf ( "%d%d", &a, &b, &c );`	переменная ***c*** не будет введена, так как на нее не задан формат
`scanf ( "%f%f", &a, &b );`	нельзя вводить целые переменные по вещественному формату

- Для вычислений используют ***оператор присваивания***, в котором

  ⇒ справа от знака равенства стоит арифметическое выражение, которое надо вычислить

  ⇒ слева от знака равенства ставится имя переменной, в которую надо записать результат

  ```
  c = a + b;        // вычислить сумму a и b и записать результат в c
  ```

- Для вывода чисел и значений переменных на экран используют функцию `printf`

```
printf ( "Результат: %d + %d = %d \n", a, b, c );
```

содержание скобок при вызове функции `printf` очень похоже на функцию `scanf`

⇒ Сначала идет символьная строка — формат вывода — в которой можно использовать специальные символы

%d	вывод целого числа
%f	вывод вещественного числа
%c	вывод одного символа
%s	вывод символьной строки
\n	переход в начало новой строки

все остальные символы (кроме некоторых других специальных команд) просто выводятся на экран.

⇒ В символьной строке мы сказали компьютеру, *какие* данные (целые, вещественные или символьные) надо вывести на экран, но не сказали *откуда* (из каких ячеек памяти) их брать. Поэтому через запятую после формата вывода надо поставить список чисел или переменных, значения которых надо вывести, при этом можно сразу проводить вычисления.

```
printf ( "Результат: %d + %d = %d \n", a, 5, a+5 );
```

⇒ Так же, как и для функции `scanf`, надо следить за совпадением типов и количества переменных и форматов вывода.

Арифметические выражения

Из чего состоят арифметические выражения?

Арифметические выражения, стоящие в правой части оператора присваивания, могут содержать

- целые и вещественные числа (в вещественных числах целая и дробная часть разделяются точкой, а не запятой, как это принято в математике)
- знаки арифметических действий

+ —	сложение, вычитание
** /*	умножение, деление
%	остаток от деления

- вызовы стандартных функций

`abs(i)`	модуль целого числа ***i***

`fabs(x)`	модель вещественного числа ***x***
`sqrt(x)`	квадратный корень из вещественного числа ***x***
`pow(x,y)`	вычисляет ***x*** в степени ***y***

- круглые скобки

Особенности арифметических операций

При использовании деления надо помнить, что

*При делении целого числа на целое остаток от деления отбрасывается, таким образом, **7/4** будет равно 1. Если же надо получить вещественное число и не отбрасывать остаток, делимое или делитель надо преобразовать к вещественной форме. Например:*

```
int i, n;    float x;
i = 7;
x = i / 4;          // x=1, делится целое на целое
x = i / 4.;         // x=1.75, делится целое на дробное
x =(float) i / 4;   // x=1.75, делится дробное на целое
n = 7. / 4.;        // n=1, результат записывается в
                    // целую переменную
```

Наибольшие сложности из всех действий вызывает взятие остатка. Если надо вычислить остаток от деления переменной ***a*** на переменную ***b*** и результат записать в переменную ***ostatok***, то оператор присваивания выглядит так:

```
ostatok = a % b;
```

Приоритет арифметических операций

В языках программирования арифметические выражения записываются в одну строчку, поэтому необходимо знать ***приоритет (старшинство) операций***, то есть последовательность их выполнения. Сначала выполняются

- операции в скобках, затем
- вызовы функций, затем
- умножение, деление и остаток от деления, слева направо, затем
- сложение и вычитание, слева направо.

Например:

```
x  =  ( a + 5 * b ) * fabs ( c + d ) – ( 3 * b – c );
      2   1     5   4      3      8     6   7
```

Для изменения порядка выполнения операций используются круглые скобки. Например, выражение

$$y = \frac{4x+5}{(2x-15z)(3z-3)} - \frac{5x}{x+z+3}$$

в компьютерном виде запишется в виде

```
y = (4*x + 5) / ((2*x – 15*z)*(3*z – 3)) – 5 * x /
    (x + z + 3);
```

Странные операторы присваивания

В программировании часто используются несколько странные операторы присваивания, например:

```
i = i + 1;
```

Если считать это уравнением, то оно бессмысленно с точки зрения математики. Однако с точки зрения информатики этот оператор служит для увеличения значения переменной ***i*** на единицу. Буквально это означает: взять старое значение переменной ***i***, прибавить к нему единицу и записать результат в ту же переменную ***i***.

Инкремент и декремент

В языке Си определены специальные операторы быстрого увеличения на единицу (***инкремента***)

```
i ++;          //  или
++ i;
```

что равносильно оператору присваивания

```
i = i + 1;
```

и быстрого уменьшения на единицу (***декремента***)

```
i --;     //  или
-- i;
```

что равносильно оператору присваивания

```
i = i - 1;
```

Между первой и второй формами этих операторов есть некоторая разница, но только тогда, когда они входят в состав более сложных операторов или условий.

Сокращенная запись арифметических выражений

Если мы хотим изменить значение какой-то переменной (взять ее старое значение, что-то с ним сделать и записать результат в эту же переменную), то удобно использовать сокращенную запись арифметических выражений:

Сокращенная запись	Полная запись
`x += a;`	`x = x + a;`
`x -= a;`	`x = x - a;`
`x *= a;`	`x = x * a;`
`x /= a;`	`x = x / a;`
`x %= a;`	`x = x % a;`

Форматы для вывода данных

Целые числа

Первым параметром при вызове функций `scanf` и `printf` должна стоять символьная строка, определяющая формат ввода или данных. Для функции `scanf`, которая выполняет ввод данных, достаточно просто указать один из форматов "`%d`", "`%f`" или "`%c`" для ввода целого числа, вещественого числа или символа, соответственно. В то же время форматная

строка в функции printf позволяет управлять выводом на экран, а именно, задать размер поля, которое отводится для данного числа.

Ниже показаны примеры форматирования при выводе целого числа 1234. Чтобы увидеть поле, которое отводится для числа, оно ограничено слева и справа скобками.

Пример вывода	Результат	Комментарий
`printf ( "[%d]", 1234);`	`[1234]`	На число отводится минимально возможное число позиций.
`printf ( "[%6d]", 1234);`	`[  1234]`	На число отводится 6 позиций, выравнивание вправо.
`printf ( "[%-6d]", 1234);`	`[1234  ]`	На число отводится 6 позиций, выравнивание влево.
`printf ( "[%2d]", 1234);`	`[1234]`	Число не помещается в заданные 2 позиции, поэтому область вывода расширяется.

Для вывода символов используются такие же приемы форматирования, но формат "`%d`" заменяется на "`%c`".

Вещественные числа

Для вывода (и для ввода) вещественных чисел могут использоваться три формата: "`%f`", "`%e`" и "`%g`". В таблице показаны примеры использования формата "`%f`".

Пример вывода	Результат	Комментарий
`printf ( "[%f]", 123.45);`	`[123.450000]`	На число отводится минимально возможное число позиций, выводится 6 знаков в дробной части.
`printf("[%9.3f]", 123.45);`		На число отводится всего 9 позиций, из них 3 – для дробной части, выравнивание вправо.
`printf("[%-9.3f]", 123.45);`	`[123.450  ]`	На число отводится всего 9 позиций, из них 3 – для дробной части, выравнивание влево.
`printf("[%6.4f]", 123.45);`	`[123.4500]`	Число не помещается в заданные 6 позиций (4 цифры в дробной части), поэтому область вывода расширяется.

Формат "`%e`" применяется в научных расчетах для вывода очень больших или очень маленьких чисел, например, размера атома или расстояния до Солнца. С представлением числа в так называемом *стандартном виде* (с выделенной *мантиссой* и *порядком*). Например, число `123.45` может быть записано в стандартном виде как $123.45 = 1.2345 \times 10^2$. Здесь `1.2345` – мантисса (она всегда находится в интервале от 1 до 10), а 2 – порядок (мантисса умножается на 10 в этой степени). При выводе по формату "%e" также можно задать число позиций, которые отводятся для вывода числа, и число цифр в дробной части мантиссы. Порядок всегда указывается в виде двух цифр, перед которыми стоит буква "`e`" и знак порядка (плюс или минус).

Пример вывода	Результат	Комментарий
`printf ( "[%e]", 123.45);`	`[1.234500e+02]`	На число отводится минимально возможное число позиций, выводится 6 знаков в дробной части мантиссы.
`printf("[%12.3e]", 123.45);`	`[ 1.234e+02]`	На число отводится всего 12 позиций, из них 3 – для дробной части мантиссы, выравнивание вправо.
`printf("[%-12.3e]", 123.45);`	`[1.234e+02 ]`	На число отводится всего 12 позиций, из них 3 – для дробной части мантиссы, выравнивание влево.
`printf("[%6.2e]", 123.45);`	`[1.23e+02]`	Число не помещается в заданные 6 позиций (2 цифры в дробной части мантиссы), поэтому область вывода расширяется.

Формат "`%g`" применяется для того, чтобы удалить лишние нули в конце дробной части числа и автоматически выбрать формат (в стандартном виде или с фиксированной точкой). Для очень больших или очень маленьких чисел выбирается формат с плавающей точкой (в стандартном виде). В этом формате можно задать общее число позиций на число и количество значащих цифр.

Пример вывода	Результат	Комментарий

`printf ("[%g]", 12345);`
`printf ("[%g]", 123.45);`
`printf ("[%g]", 0.000012345);`
	`[12345]`
	`[123.45]`
	`[1.2345e-05]`
		На число отводится минимально возможное число позиций, выводится не более 6 значащих цифр.

`printf ("[%10.3g]", 12345);`
`printf ("[%10.3g]", 123.45);`
`printf("[%10.3g]",0.000012345);`
	`[ 1.23e+04]`
	`[ 123]`
	`[ 1.23e-05]`
		На число отводится всего 10 позиций, из них 3 значащие цифры, выравнивание вправо. Чтобы сделать выравнивание влево, используют формат "`%-10.3g`".

3. Циклы

Зачем нужны циклы?

Теперь посмотрим, как вывести на экран это самое приветствие 10 раз. Конечно, можно написать 10 раз оператор `printf`, но если надо вывести строку 200 раз, то программа значительно увеличится. Поэтому надо использовать ***циклы.***

Цикл - это последовательность команд, которая выполняется несколько раз.

В языке Си существует несколько видов циклов.

Цикл с известным числом шагов (`for`)

Часто мы заранее знаем заранее (или можем рассчитать), сколько раз нам надо выполнить какую-то операцию. В некоторых языках программирования для этого используется цикл `repeat` - повтори заданное количество раз. Подумаем, как выполнять такой цикл. В самом деле, в памяти выделяется ячейка и в нее записывается число повторений. Когда программа выполняет тело цикла один раз, содержимое этой ячейки (***счетчик***) уменьшается на единицу. Выполнение цикла заканчивается, когда в этой ячейке будет нуль.

В языке Си цикла `repeat` нет, а есть цикл `for`. Он не скрывает ячейку-счетчик, а требует явно объявить ее (выделить под нее память), и даже позволяет использовать ее значение в теле цикла. Ниже показан пример программы, которая печатает приветствие 10 раз.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i;
for ( i=1; i <= 10; i ++)
   {
   printf("Привет");
   }
getch();
}
```

объявление переменной ***i*** (выделение памяти)

заголовок цикла

тело цикла

Что новенького?

- Цикл `for` используется тогда, когда количество повторений цикла заранее известно или может быть вычислено.
- Цикл `for` состоит из заголовка и тела цикла.
- В заголовке после слова `for` в круглых скобках записываются через точку с запятой три выражения:

  ⇒ начальные условия: операторы присваивания, которые выполняются один раз перед выполнением цикла

  ⇒ условие, при котором выполняется следующий шаг цикла; если условие неверно, работа цикла заканчивается; если оно неверно в самом начале, цикл не выполняется ни

одного раза (говорят, что это - *цикл с предусловием*, то есть условие проверяется перед выполнением цикла);

⇒ действия в конце каждого шага цикла (в большинстве случаев это операторы присваивания)

- В каждой части заголовка может быть несколько операторов, разделенных запятыми. Примеры заголовков:

```
for ( i = 0; i < 10; i ++ ) {  ...  }

for ( i = 0, x = 1.; i < 10; i += 2, x *= 0.1 ){ ... }
```

- Тело цикла заключается в фигурные скобки; если в теле цикла стоит всего один оператор, скобки можно не ставить.

- В тело цикла могут входить любые другие операторы, в том числе и другие циклы (такой прием называется *"вложенные циклы"*).

- Для того, чтобы легче разобраться в программе, все тело цикла и ограничивающие его скобки сдвигаются вправо на 2-3 символа.

Чему равен квадрат числа?

Напишем программу, которая вводит с клавиатуры натуральное число ***N*** и выводит на экран квадраты всех целых чисел от 1 до ***N*** таком виде

Квадрат числа 1 равен 1
Квадрат числа 2 равен 4
...

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i, N;                          // i - переменная цикла
printf ( "Введите число N: " ); // подсказка для ввода
scanf ( "%d", &N );              // ввод N с клавиатуры
for ( i = 1; i <= N; i ++)       // цикл: для всех i от 1 до N
  {
  printf ( "Квадрат числа %d равен %d\n", i, i*i);
  }
getch();
}
```

Мы объявили две переменные: ***N*** — максимальное число, ***i*** — вспомогательная переменная, которая в цикле принимает последовательно все значения от ***1*** до ***N***. Для ввода значения ***N*** мы напечатали на экране подсказку и использовали функцию `scanf` с форматом ***%d*** (ввод целого числа).

При входе в цикл выполняется оператор ***i = 1,*** и затем переменная ***i*** с каждым шагом увеличивается на 1 (***i ++***). Цикл выполняется пока истинно условие ***i <= N***. В теле цикла единственный оператор вывода печатает на экране само число и его квадрат по заданному формату. Для возведения в квадрат или другую невысокую степень лучше использовать умножение.

Цикл с условием (`while`)

Очень часто заранее невозможно сказать, сколько раз надо выполнить какую-то операцию, но можно определить условие, при котором она должна заканчиваться. Такое задание на русском языке может выглядеть так: делай эту работу до тех пор, ***пока*** она не будет закончена (пили бревно, пока оно не будет распилено; иди вперед, пока не дойдешь до двери). Слово ***пока*** на английском языке записывается как `while`, и так же называется еще один вид цикла.

Задача. Ввести целое число (меньше 2 000 000 000) и определить, сколько в нем цифр.

Для решения этой задачи обычно применяется такой алгоритм. Число делится на 10 и отбрасывается остаток, и так до тех пор, пока результат деления не равен нулю. С помощью специальной переменной (она называется *счетчиком*) считаем, сколько раз выполнялось деление - столько цифр и было в числе. Понятно, что нельзя заранее определить, сколько раз надо разделить число, поэтому надо использовать цикл с условием.

В этой задаче еще важно, что число ***N*** может быть очень большое. Дело в том, что в переменную типа `int` «помещается» целое число в интервале от ***-32768*** до ***32767***. Для больших чисел надо использовать тип `long int` (длинное целое). Переменные этого типа занимают в памяти 4 байта и могут хранить значения в пределах 2 млрд. (примерно). Для ввода и вывода таких чисел используют формат ***%ld***.

```
#include <stdio.h>
#include <conio.h>

void main()
{
int count=0; // count - переменная-счетчик
long N;
printf ( "\nВведите число N: " );  // подсказка
scanf ( "%ld", &N );                // ввод N с клавиатуры

while ( N > 0 )
  {
  N /= 10;
  count ++;
  }

printf ( "В этом числе %d цифр\n", count );
getch();
}
```

заголовок цикла (пока N > 0)

тело цикла

Что новенького?

- Цикл `while` используется тогда, когда количество повторений цикла заранее неизвестно и не может быть вычислено.
- Цикл `while` состоит из заголовка и тела цикла.
- В заголовке после слова `while` в круглых скобках записывается условие, при котором цикл продолжает выполняться; когда условие становится неверно, цикл заканчивается.
- В условии можно использовать знаки логических отношений и операций
  `>` `<` больше, меньше

`>=`	`<=`	больше или равно, меньше или равно
`==`		равно
`!=`		не равно

- Если условие неверно в самом начале, то цикл не выполняется ни разу (это цикл с *предусловием*).

- Если условие никогда не становится *ложным* (неверным), то цикл никогда не заканчивается; в таком случае говорят, что программа ***"зациклилась"*** — это серьезная логическая ошибка.

- В языке Си любое число, не равное нулю, обозначает истинное условие, а ноль — ложное условие:

```
while ( 1 ){ ... }   // бесконечный цикл

while ( 0 ){ ... }   // цикл не выполнится ни разу
```

- Тело цикла заключается в фигурные скобки; если в теле цикла стоит всего один оператор, скобки можно не ставить.

- В тело цикла могут входить любые другие операторы, в том числе и другие циклы (такой прием называется *"вложенные циклы"*).

- Для того, чтобы легче разобраться в программе, все тело цикла и ограничивающие его скобки сдвигаются вправо на 3-4 символа.

- Если надо работать с большими целыми числами, используется тип данных `long int`. Для ввода и вывода переменных этого типа используют формат ***`%ld`***.

Цикл с постусловием (`do – while`)

Существуют также случаи, когда надо выполнить цикл хотя бы дин раз, а затем на каждом шагу делать проверку некоторого условия и закончить цикл, когда это условие станет ложным. Для этого используется ***цикл с постусловием*** (то есть условие проверяется не в начале, а в конце цикла). Не рекомендуется применять его слишком часто, поскольку он напоминает такую ситуацию: прыгнул в бассейн, и только потом посмотрел, есть ли в нем вода. Рассмотрим случай, когда его использование оправдано.

Задача. Ввести натуральное число и найти сумму его цифр. Организовать ввод числа так, чтобы нельзя было ввести отрицательное число или нуль.

Любая программа должна обеспечивать защиту от неверного ввода данных (иногда такую защиту называют "защитой от дурака" — *fool proof*). Поскольку пользователь может вводить данные неверно сколько угодно раз, то надо использовать цикл с условием. С другой стороны, по крайней мере один раз надо ввести число, поэтому нужен цикл с постусловием.

В отличие от предыдущей программы, теперь надо при каждом делении определять остаток (последняя цифра числа равна остатку от деления его на 10) и суммировать все остатки в специальной переменной.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int sum=0, N;     // sum - сумма цифр числа

do {
  printf ( "\nВведите натуральное число:" );
  scanf ( "%d", &N );
  }
while ( N <= 0 );

while ( N > 0 ) {
  sum += N % 10;
  N /= 10;
  count ++;
  }
printf ( "Сумма цифр этого числа равна %d\n", sum );
getch();
}
```

заголовок цикла

тело цикла

условие цикла (пока N <= 0)

Что новенького?

- Цикл `do—while` используется тогда, когда количество повторений цикла заранее неизвестно и не может быть вычислено.
- Цикл состоит из заголовка `do`**,** тела цикла и завершающего условия.
- Условие записывается в круглых скобках после слова `while`, цикл продолжает выполняться, пока условие верно; когда условие становится неверно, цикл заканчивается.
- Условие проверяется только в конце очередного шага цикла (это цикл *с постусловием*), таким образом, цикл всегда выполняется хотя бы один раз.
- Если условие никогда не становится ложным (неверным), то цикл никогда не заканчивается; в таком случае говорят, что программа ***"зациклилась"*** — это серьезная логическая ошибка.
- Тело цикла заключается в фигурные скобки; если в теле цикла стоит всего один оператор, скобки можно не ставить.
- В тело цикла могут входить любые другие операторы, в том числе и другие циклы (такой прием называется *"вложенные циклы"*).
- Для того, чтобы легче разобраться в программе, все тело цикла и ограничивающие его скобки сдвигаются вправо на 2-3 символа.

Вычисление сумм последовательностей

Суммы с заданным числом элементов

Задача. Найти сумму первых 20 элементов последовательности

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{6} - \frac{4}{12} + \cdots$$

Чтобы решить эту задачу, надо определить закономерность в изменении элементов. В данном случае можно заметить:

- Каждый элемент представляет собой дробь.
- Числитель дроби при переходе к следующему элементу возрастает на единицу.
- Знаменатель дроби с каждым шагов увеличивается в 2 раза.
- Знаки перед дробями чередуются (плюс, минус и т.д.).

Любой элемент последовательности можно представить в виде

$$a_i = \frac{zc}{d},$$

где изменение переменных **z**, **c** и **d** описываются следующей таблицей (для первых пяти элементов)

i	**1**	**2**	**3**	**4**	**5**
z	1	-1	1	-1	1
c	1	2	3	4	5
d	2	3	6	12	24

У переменной **z** меняется знак (эту операцию можно записать в виде **z=-z**), значение переменной с увеличивается на единицу (**c++**), а переменная **d** умножается на 2 (**d=d*2**). Алгоритм решения задачи можно записать в виде следующих шагов

- Записать в переменную **S** значение 0. В этой ячейке будет накапливаться сумма.
- Записать в переменные **z**, **c** и **d** начальные значения (для первого элемента): **z=1, c=1, d=2**.
- Сделать 20 раз:
  ⇒ добавить к сумме значение очередного элемента
  ⇒ изменить значения переменных **z**, **c** и **d** для следующего элемента.

```
#include <stdio.h>
void main()
{
float S, z, c, d;
int i;
S = 0; z = 1; c = 1; d = 2;
for ( i = 1; i <= 20; i ++ )
   {
   S = S + z*c/d;
   z = - z;
   c ++;
   d = d * 2;
   }
printf("Сумма S = %f", S);
}
```

начальные значения

добавить элемент к сумме

изменить переменные

Суммы с ограничивающим условием

Рассмотрим более сложную задачу, когда количество элементов заранее неизвестно.

Задача. Найти сумму всех элементов последовательности

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{6} - \frac{4}{12} + \cdots,$$

которые по модулю меньше, чем 0.001.

Эта задача имеет решение только тогда, когда элементы последовательности убывают по модулю и стремятся к нулю. Поскольку мы не знаем, сколько элементов войдет в сумму, надо использовать цикл `while` (или `do - while`). Один из вариантов решения показан ниже.

```
#include <stdio.h>
void main()
{
float S, z, c, d, a;
S = 0; z = 1; c = 1; d = 2;
a = 1;
while ( a >= 0.001 )
   {
   a = c / d;
   S = S + z*a;
   z = - z;
   c ++;
   d = d * 2;
   }
printf("Сумма S = %f", S);
}
```


Цикл закончится тогда, когда переменная ***a*** (она обозначает модуль очередного элемента последовательности) станет меньше 0.001. Чтобы программа вошла в цикл на первом шаге, в эту переменную надо записать любое значение, большее, чем 0.001.

Очевидно, что если переменная ***a*** не будет уменьшаться, то условие в заголовке цикла всегда будет истинно и программа «зациклится».

4. Выбор вариантов

Зачем нужны условные операторы?

В простейших программах все команды выполняются одна за другой последовательно. Однако часто надо выбрать тот или иной вариант действий в зависимости от некоторых условий, то есть, если условие верно, поступать одним способом, а если неверно — другим. Для этого применяют условные операторы. В языке Си существует два вида условных операторов:

- оператор `if—else` для выбора из двух вариантов
- оператор множественного выбора `switch` для выбора из нескольких вариантов

Условный оператор `if — else`

Задача. Ввести с клавиатуры два вещественных числа и определить наибольшее из них.

По условию задачи нам надо вывести один из двух вариантов ответа: если первое число больше второго, то вывести на экран его, если нет — то второе число. ниже показаны два варианта решения этой задачи: в первом результат сразу выводится на экран, а во втором сначала наибольшее из двух чисел записывается в третью переменную.

```
#include <stdio.h>
#include <conio.h>
void main()
{
float A, B;
printf ("Введите A и B :");
scanf ( "%f%f", &A, &B );
if ( A > B )
 {
 printf ( "Наибольшее %f",
           A );
 }
else
 {
 printf ( "Наибольшее %f",
           B );
 }
getch();
}
```

```
#include <stdio.h>
#include <conio.h>
void main()
{
float A, B, Max;
printf("Введите A и B : ");
scanf ( "%f%f", &A, &B );
if ( A > B )
    {
    Max = A;
    }
else
    {
    Max = B;
    }
printf ( "Наибольшее %f",
          Max );
getch();
}
```

заголовок с условием

блок "если"

блок "иначе"

Что новенького?

- Условный оператор имеет следующий вид:

```
if ( условие )
   {
   ...   // блок "если" — операторы, которые выполняются,
       // если условие в скобках истинно
   }
else
  {
   ...   // блок "иначе " — операторы, которые выполняются,
         // если условие в скобках ложно
  }
```

- Эта запись представляет собой единый оператор, поэтому между скобкой, завершающей блок "***если"*** и словом `else` не могут находиться никакие операторы.
- После слова `else` никогда НЕ ставится условие — блок "иначе" выполняется тогда, когда основное условие, указанное в скобках после `if`, ложно.
- Если в блоке "если" или в блоке "иначе" только один оператор, то фигурные скобки можно не ставить.
- В условии можно использовать любые отношения и логические операции.
- Если в блоке "иначе" не надо ничего делать (например: "если в продаже есть мороженое, купи мороженое", а если нет ...), то весь блок "иначе" можно опустить и использовать сокращенную форму условного оператора:

```
if ( условие )
   {
   ...   // блок "если" – операторы, которые выполняются,
         // если условие в скобках истинно
   }
```

Например, решение предыдущей задачи могло бы выглядеть так:

```
#include <stdio.h>
#include <conio.h>
void main()
{
float A, B, Max;
printf("Введите A и B : ");
scanf ( "%f%f", &A, &B );
Max = A;
if ( B > A )
   Max = B;
printf ( "Наибольшее %f", Max );
getch();
}
```

- В блоки "если" и "иначе" могут входить любые другие операторы, в том числе и другие вложенные условные операторы; при этом оператор `else` относится к ближайшему предыдущему `if`:

```
if ( A > 10 )
   if ( A > 100 )
      printf ( "У вас очень много денег." );
   else
      printf ( "У вас достаточно денег." );
else
   printf ( "У вас маловато денег." );
```

- Чтобы легче разобраться в программе, все блоки "если" и "иначе" (вместе с ограничивающими их скобками) сдвигаются вправо на 2-3 символа.

Сложные условия

Простейшие условия состоят из одного отношения (больше, меньше и т.д.). Иногда надо написать условие, в котором объединяются два или более простейших отношений. Например,

фирма отбирает сотрудников возраста 25—40 лет. Тогда простейшая программа могла бы выглядеть так:

```
#include <stdio.h>
#include <conio.h>

void main()
{
int age;
printf ( "\nВведите ваш возраст: " );
scanf ( "%d", &age );
if ( 25 <= age  &&  age <= 40 )
     printf ( "Вы нам подходите." );
else
     printf ( "Извините, Вы нам не подходите." );
getch();
}
```

сложное условие

Что новенького?

- Сложное условие состоит из двух или нескольких простых отношений, которые объединяются с помощью знаков *логических операций*:

  ⇒ операция "***И"*** — требуется одновременное выполнение двух условий

  ```
  условие_1   &&   условие_2
  ```

  Эту операцию можно описать следующей таблицей (она называется *таблицей истинности*)

условие_1	условие_2	условие_1 && условие_1
ложно (0)	ложно (0)	ложно(0)
ложно (0)	истинно (1)	ложно(0)
истинно (1)	ложно (0)	ложно(0)
истинно (1)	истинно (1)	истинно (1)

  ⇒ операция ***"ИЛИ"*** — требуется выполнение хотя бы одного из двух условий (или обоих сразу)

  ```
  условие_1   ||  условие_2
  ```

  Таблица истинности запишется в виде

условие_1	условие_2	условие_1 || условие_1
ложно (0)	ложно (0)	ложно(0)
ложно (0)	истинно (1)	истинно (1)
истинно (1)	ложно (0)	истинно (1)
истинно (1)	истинно (1)	истинно (1)

  ⇒ в очень сложных условиях иногда используется операция ***"НЕ"*** — отрицание условия (или обратное условие)

  ```
  ! условие
  ```

  Например, следующие два условия равносильны

  ```
  A > B                  ! ( A <= B )
  ```

- Порядок выполнения (*приоритет*) логических операций и отношений:

⇒ операции в скобках, затем
⇒ операция "НЕ", затем
⇒ логические отношения `>, <, >=, <=, ==, !=`, затем
⇒ операция "И", затем
⇒ операций "ИЛИ"

- Для изменения порядка действий используются круглые скобки.

Досрочный выход из цикла

Иногда надо выйти из цикла и перейти к следующему оператору, не дожидаясь окончания очередного шага цикла. Для этого используют специальный оператор `break`. Можно также сказать компьютеру, что надо завершить текущий шаг цикла и сразу перейти к новому шагу (не выходя из цикла) — для этого применяют оператор `continue`.

Задача. Написать программу, которая вычисляет частное и остаток от деления двух введенных целых чисел. Программа должна работать в цикле, то есть запрашивать значения делимого и делителя, выводить результат, снова запрашивать данные и т.д. Если оба числа равны нулю, надо выйти из цикла и завершить работу программы. Предусмотреть сообщение об ошибке в том случае, если второе число равно нулю, а первое — нет.

Особенность этой задачи состоит в том, что при входе в цикл мы не можем определить, надо ли будет выполнить до конца очередной шаг. Необходимая информация поступает лишь при вводе данных с клавиатуры. Поэтому здесь используется бесконечный цикл `while(1){...}` (единица считается истинным условием). Выйти из такого цикла можно только с помощью специального оператора `break`.

В то же время, если второе число равно нулю, то оставшуюся часть цикла не надо выполнять. Для этого служит оператор `continue`.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int A, B;
while ( 1 )
   {
   printf ( "\nВведите делимое и делитель:" );
   scanf ( "%d%d", &A, &B );
   if ( A = = 0  &&  B = = 0 ) break;
   if ( B = = 0 ) {
      printf ( "Деление на ноль" );
      continue;
      }
   printf ( "Частное %d остаток %d", A/B, A%B )
   }
getch();
}
```

бесконечный цикл

выход из цикла

досрочный переход в начало цикла

Что новенького?

- Если только внутри цикла можно определить, надо ли делать вычисления в теле цикла и надо ли продолжать цикл (например, при вводе исходных данных), часто используют бесконечный цикл, внутри которого стоит оператор выхода `break`:

```
while ( 1 ) {
...
if ( надо выйти )  break;
...
}
```

- С помощью оператора `break` можно досрочно выходить из циклов `for`, `while`, `do – while`.
- Чтобы досрочно завершить текущий шаг цикла и сразу перейти к следующему шагу, используют оператор `continue`.

Переключатель `switch` (множественный выбор)

Если надо выбрать один из нескольких вариантов в зависимости от значения некоторой целой или символьной переменной, можно использовать несколько вложенных операторов ***if***, но значительно удобнее использовать специальный оператор ***switch.***

Задача. Составить программу, которая вводит с клавиатуры русскую букву и выводит на экран название животного на эту букву.

```
#include <stdio.h>
#include <conio.h>
void main()
{
char c;
printf("\nВведите первую букву:");
c = getch();
switch ( c )
  {
  case 'а': printf("\nАнтилопа"); break;
  case 'б': printf("\nБарсук"); break;
  case 'в': printf("\nВолк"); break;
  default:  printf("\nНе знаю я таких!");
  }
getch();
}
```

получить код нажатой клавиши

заголовок

варианты выбора

вариант по умолчанию

Что новенького?

- Функция `getch` возвращает в качестве результата код нажатой клавиши в таблице кодов. Код клавиши - это целое число от 0 до 255.
- Оператор множественного выбора `switch` состоит из заголовка и тела оператора, заключенного в фигурные скобки.
- В заголовке после ключевого слова `switch` в круглых скобках записано имя переменной (целой или символьной), в зависимости от значения которой делается выбор между несколькими вариантами.
- Каждому варианту соответствует метка `case`, после которой стоит одно из возможных значений этой переменной и двоеточие; если значение переменной совпадает с одной из меток, то программа переходит на эту метку и выполняет все последующие операторы.
- Оператор `break` служит для выхода из тела оператора `switch`.

- Если убрать все операторы `break`, то, например, при нажатии на букву ***'a'*** будет напечатано

```
Антилопа
Барсук
Волк
```

- Если значение переменной не совпадает ни с одной из меток, программа переходит на метку `default` (по умолчанию, то есть если ничего другого не указано).

- Можно ставить две метки на один оператор, например, чтобы программа реагировала как на большие, так и на маленькие буквы, надо в теле оператора `switch` написать так:

```
case 'а':
case 'А':
    printf("\nАнтилопа"); break;
case 'б':
case 'Б':
    printf("\nБарсук"); break;
```

и так далее.

5. Методы отладки программ

Отладочные средства *Borland C 3.1*

Пошаговое выполнение

Обычно программа запускается на выполнение клавишами ***Ctrl-F9*** и выполняется безостановочно. Если вы заметили, что она работает не так, как вы хотели, ее надо ***отлаживать***, то есть искать и исправлять в ней ошибки.

Лучший способ отладки — это выполнить программы по строчкам, останавливаясь после выполнения каждой строки и проверяя значения переменных в памяти. Для этой цели служат специальные программы — ***отладчики***.

Отладчики бывают автономные и встроенные. С системой программирования ***Borlanc C*** поставляются автономные отладчики ***Turbo Debugger*** (файлы ***td.exe, td286.exe*** и ***td386.exe***). Однако для отладки простых программ удобнее использовать встроенный отладчик оболочки ***Borland C***, в которой вы набираете и запускаете программу. Для того, чтобы с помощью встроенного отладчика выполнить программу по строчкам, служит комбинация клавиш ***F8***.

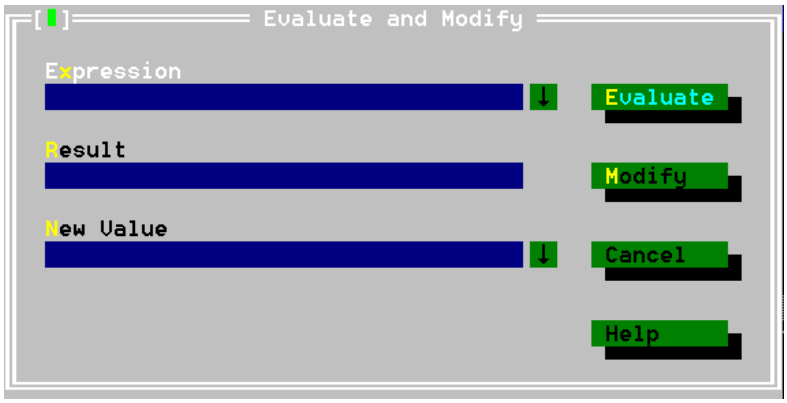
Полоска голубого цвета обозначает строку, которая будет выполняться следующей (но еще не выполнилась). Если нажать на ***F8***, выполняются все операторы в этой строке и выделение переходит на следующую строку. Для запуска программы без остановки надо нажать ***Ctrl-F9***, для остановки программы — ***Ctrl-F2***.

Однако в этом режиме встроенный отладчик не позволяет входить внутрь вызываемых процедур (по шагам выполняется только основная программа). Для входа в процедуру или функцию используют комбинацию ***F7*** (если в этой строчке нет вызова процедур и функций, она равносильна ***F8***).

Просмотр и изменение значений переменных

Пошаговое выполнение программы позволяет лишь посмотреть, какие операторы программы выполняются, сколько раз и в какой последовательности. Однако чаще всего этого оказывается недостаточно для обнаружения ошибки.

Очень мощным средством отладки является просмотр значений переменных во время выполнения программы. Для этого надо нажать ***Ctrl-F4***, после чего появится такое окно.



Если в этом окне в поле ***Expression*** ввести имя переменной или массива и нажать клавишу ***Enter***, то соответствующее значение появится в поле ***Result.*** Более того, если вы захотите изменить значение этой переменной (и посмотреть, как будет выполняться программа дальше, если бы значение этой переменной было бы другое), надо перейти в поле ***New Value***, ввести там новое значение и нажать ***Enter***.

Окно просмотра переменных

В интегрированной среде ***Borland C*** существует специальное окно, в котором вы можете постоянно наблюдать значения переменных во время выполнения программы. Для добавления

новой переменной в это окно просмотра надо нажать ***Ctrl-F7,*** ввести имя переменной и нажать ***Enter***. Чтобы перейти в окно просмотра, надо выбрать в меню пункты ***Window-Watch*** (или нажать клавиши ***Alt-W + W***).

Точки останова (Breakpoints)

Встроенный отладчик позволяет также остановить программу в любом месте. Для этого надо поставить курсор в нужную строчку и нажать ***Ctrl-F8***. При этом строка выделяется красным цветом (для снятия точки останова надо еще раз нажать ***Ctrl-F8***). Программа, запущенная по ***Ctrl-F9***, выполняется без остановки, пока не встретится точка останова. В этом случае вы снова оказываетесь в окне редактора и можете просмотреть значения переменных и продолжить программу в пошаговом режиме.

Существуют еще дополнительные возможности — задать останов в некоторой строке при определенном условии или после прохождения через эту точку определенное количество раз. Это выполняется с помощью меню ***Debug—Breakpoints-Edit.***

Отладочные комбинации клавиш

Alt-F9	компиляция текущего модуля (проверка ошибок)
F9	компиляция всех модулей проекта и компоновка выполняемой программы (***exe-***файла)
Ctrl-F9	запуск программы (если надо, то она компилируется и компонуется)
F8	выполнить один шаг и остановиться
F7	войти в процедуру
Alt-F5	просмотреть экран программы
Ctrl-F2	остановить пошаговое выполнение
F4	выполнить до строки, в которой стоит курсор
Ctrl-F4	просмотр и изменение значений переменных
Ctrl-F7	добавить переменную в окно просмотра

Практические приемы

Отключение частей кода

Довольно часто при внесении каких-то изменений программа перестает работать. Основная задача в этом случае — определить, в каком именно месте возникает ошибка (иначе говоря, с какого места программа перестает работать верно). При этом хорошие результаты дает *метод отключения блоков программы:* начиная с конца программы, новые блоки заключаются в знаки комментария (**/*** и *******/*), то есть компьютер их не выполняет. Таким образом, добиваются, чтобы программа работала правильно, и определяют строку, в которой происходит ошибка.

Ручная прокрутка программы

Если другие способы не помогают, приходится делать *ручную прокрутку программы*, то есть выполнять программу вручную вместо компьютера, записывая результаты на лист бумаги.

Обычно составляют таблицу, в которую записывают изменения всех переменных (неизвестное значение переменной обозначают знаком вопроса). Рассмотрим (ошибочную) программу, которая вводит натуральное число и определяет, простое оно или нет. Мы выяснили, что она дает неверный результат при ***N=5*** (печатает, что 5 – якобы составное число). Построим таблицу изменения значений переменных для этого случая.

```
#include <stdio.h>
void main()
{
   int   N, i, count = 0;
   printf("Введите число ");
   scanf("%d", &N);
   for ( i = 2; i <= N; i ++ )
       if ( N %i == 0 )
            count ++;
   if ( count == 0 )
        printf("Число простое");
   else printf("Число составное");
}
```

N	i	count
2	?	0
	2	
	3	
	4	
	5	1

Выполняя вручную все действия, выясняем, что программа проверяет делимость числа ***N*** на само себя, то есть, счетчик делителей `count` всегда будет не равен нулю. Теперь, определив причину ошибки, легко ее исправить. Для этого достаточно заменить условие в цикле на `i < N`.

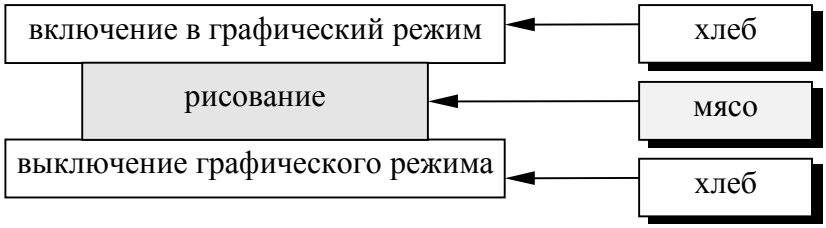
Проверка крайних значений

Очень важно проверить работу программы (и функций) на крайних значениях возможного диапазона исходных данных. Например, для программы, определяющей простоту натуральных чисел, надо проверить, будет ли она работать для чисел ***1*** и ***2***. Аналогично, если мы отлаживаем программу, которая ищет заданное слово в строке, надо проверить ее для тех случаев, когда искомое слово стоит первым или последним в строке.

6. Работа в графическом режиме

Общая структура программы

Особенность современных компьютеров заключается в том, что они могут работать в двух режимах — текстовом и графическом, поэтому для рисования на экране линий, прямоугольников и т.п. надо включать монитор в графический режим. При этом текст также можно выводить на экран, но с помощью других функций. Таким образом, графическая программа на Си имеет структуру сэндвича:



При переключении режима (из текстового в графический или наоборот) все содержимое экрана стирается. Поэтому перед выходом из графического режима надо делать паузу до нажатия клавиши — иначе не увидеть результат работы программы.

Простейшая графическая программа

Сейчас мы напишем простейшую графическую программу. Она не делает ничего полезного, просто включает монитор в графический режим, делает паузу до нажатия клавиши и выключает графический режим. Поэтому программа эта так же неполноценна, как сэндвич без мяса.

```
#include <graphics.h>
#include <conio.h>
void main()
{
int  d = VGA, m = VGAHI;
initgraph ( &d, &m, "c:\\borlandc\\bgi" );
getch();
closegraph();
}
```

подключить графические функции

определить тип драйвера экрана
режим VGAHI: 640x480, 16 цветов

включить графический режим

выключить графический режим

Что новенького?

- Для использования графических функции надо подключить файл заголовков ***graphics.h*** в начале программы.
- Для включения монитора в графический режим надо объявить две целых переменные:

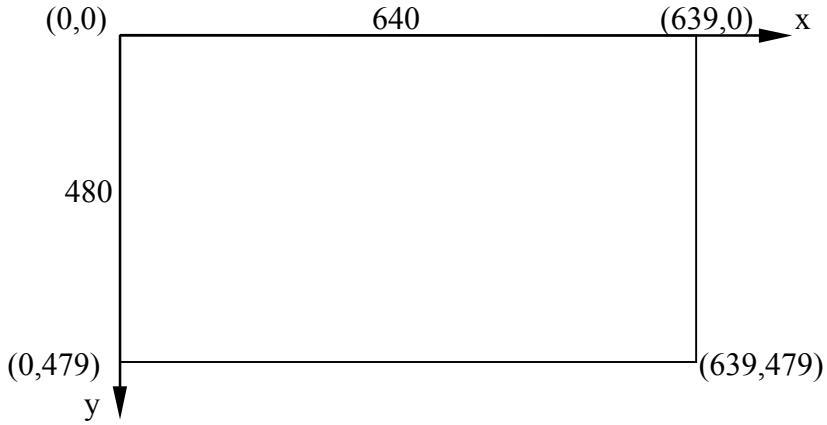
  ⇒ Одна из них (в примере — ***d***) обозначает тип драйвера монитора (***драйвер*** — это специальный набор процедур, который выполняет графические операции). Эта переменная может принимать значения ***CGA, EGA, VGA*** и ***DETECT*** (определить тип монитора автоматически). Все современные мониторы поддерживают стандарт ***VGA***, поэтому мы будем использовать его.

⇒ Вторая переменная (в примере — ***m***) обозначает тип графического режима, например, ***VGAHI*** обозначает режим 640 на 480 точек, 16 цветов.

- Можно попросить, чтобы графический режим был выбран автоматически. Для этого в переменную ***d*** надо записать значение ***DETECT*** (все большие буквы), а значение переменной ***m*** не играет роли.
- Функция `initgraph()` включает графический режим. Она имеет 3 параметра: адреса описанных двух выше переменных и строка, в которой содержится путь к файлу-драйверу заданного графического режима (или пустая строка, если этот файл находится в текущем каталоге).
- В строках языка Си обратный слэш `'\'` — это специальный символ (вспомните `'\n'` — переход на новую строку), поэтому он дублируется, если надо включить в строку сам этот символ.
- Драйверы графического режима имеют расширение ****.bgi*** и находятся в подкаталоге ***BGI*** того каталога, в который установлен компилятор Си. Для мониторов ***EGA*** и ***VGA*** используется драйвер ***egavga.bgi.***
- Графический режим выключается с помощью функции `closegraph.` При этом экран очищается и монитор переключается в текстовый режим.

Графические функции

Пришло время нарисовать что-то на экране. Для этого надо представлять, как определять координаты на графическом экране.



Что новенького?

- В режиме ***VGAHI*** экран имеет размер 640 на 480 точек (*пикселов*).
- Начало координат, точка (0,0), находится в левом верхнем углу экрана.
- Ось ***x*** направлена влево, ось ***y*** — вниз (в отличие от общепринятой математической системы координат).
- В режиме ***VGAHI*** можно использовать 16 цветов, каждый из них имеет цифровое и символьное обозначения (лучше в программах использовать символьное обозначение, которое более понятно):

0	BLACK	черный	8	DARKGRAY	темно-серый
1	BLUE	синий	9	LIGHTBLUE	светло-синий
2	GREEN	зеленый	10	LIGHTGREEN	светло-зеленый
3	CYAN	морской волны	11	LIGHTCYAN	светлый морской волны
4	RED	красный	12	LIGHTRED	светло-красный
5	MAGENTA	фиолетовый	13	LIGHTMAGENTA	светло-фиолетовый
6	BROWN	коричневый	14	YELLOW	желтый
7	LIGHTGRAY	светло-серый	15	WHITE	белый

Для рисования используются стандартные функции

`putpixel (x, y, color);`	установить точке ***(x,y)*** цвет
`n = getpixel ( x, y );`	получить цвет точки ***(x,y)*** и записать его в целую переменную ***n***
`setcolor(color);`	установить цвет линий ***color***
`line ( x1, y1, x2, y2 );`	отрезок (x_1,y_1)- (x_2,y_2)
`rectangle ( x1, y1, x2, y2 );`	прямоугольник: левый верхний угол (x_1,y_1), правый нижний — (x_2,y_2)
`circle ( x, y, R );`	окружность с центром радиуса ***R***
`setfillstyle ( SOLID_FILL, color );`	установить режим сплошной заливки цветом ***color*** (влияет на работу функций `bar`, `floodfill` и некоторых других)
`bar ( x1, y1, x2, y2 );`	залитый прямоугольник
`floodfill ( x, y, color );`	залить область, которая содержит точку ***(x,y)*** и ограничена замкнутой линией цвета ***color***
`outtextxy ( x, y, "Привет!" );`	вывести текст, левый верхний угол которого будет находиться в точке ***(x,y)***

Пример программы

Напишем программу, которая использует стандартные функции для рисования на белом фоне красного прямоугольника с синей границей и синими диагоналями, и желтого круга с фиолетовой границей в центре экрана. Внутри круга вывести текст "Привет" зеленого цвета.

```
#include <graphics.h>
#include <conio.h>

void main()

{
int d = DETECT, m;

initgraph ( &d, &m, "C:\\BORLANDC\\BGI" );

setfillstyle (SOLID_FILL, WHITE);
bar ( 0, 0, 639, 479 );

setfillstyle ( SOLID_FILL, RED);
bar ( 220, 160, 420, 320 );

setcolor ( BLUE );
rectangle ( 220, 160, 420, 320 );

line ( 220, 160, 420, 320 );
line ( 220, 320, 420, 160);

setcolor ( MAGENTA );
circle ( 320, 240, 50);

setfillstyle (SOLID_FILL, YELLOW);
floodfill ( 320, 240, MAGENTA );

setcolor ( GREEN );
outtextxy ( 295, 235, "Привет!" );

getch();

closegraph();

}
```

автоматически выбрать режим

закрашенный прямоугольник

контур прямоугольника

диагонали

контур окружности

заливка окружности

надпись зеленого цвета

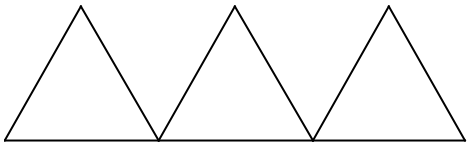
Что новенького?

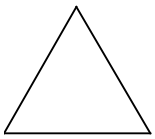
- Чтобы сделать нужный фон, надо залить весь экран выбранным цветом с помощью функции `bar`.
- Цвет линий надо устанавливать с помощью `setcolor` до рисования линий, а цвет заливки — с помощью `setfillstyle` до вызова функции `bar` или `floodfill`.
- Установленные цвета линий и заливки остаются рабочими до тех пор, пока не будут переустановлены с помощью `setcolor` или `setfillstyle`.

7. Процедуры

Пример задачи с процедурой

Часто в программах бывает легко выделить одинаковые элементы (например, одинаковые фигуры в рисунке). Составим программу, которая рисует на экране три одинаковых треугольника.





Единственное, чем отличаются эти треугольники — это место на экране. Его удобно задать координатами ***(x,y)*** одного из его углов. Пусть ***(x,y)*** — координаты левого нижнего угла треугольника, его высота 50 пикселов и основание 40 пикселов. Координаты вершин такого треугольника показаны на рисунке (с учетом того, что ось ***y*** направлена вниз). Таким образом, надо нарисовать три линии заданного цвета.

Конечно, можно три раза подряд написать почти одинаковые строчки и получить нужный результат. Однако лучше всего научить компьютер рисовать такие треугольники — ввести новую команду (к сожалению, ее название надо писать латинскими буквами) и расшифровать ее, а затем вызывать эту команду, указывая координаты левого нижнего угла для каждого нового треугольника. Такие новые команды, введенные программистом, называют *процедурами*.

Процедура - это вспомогательная программа (*подпрограмма)*, предназначенная для выполнения каких-либо действий, которые встречаются в нескольких местах программы.

Решение нашей задачи выглядит так:

```
#include <graphics.h>
#include <conio.h>
void triangle ( int x, int y );
void main()
{
int     grDriver = VGA, grMode = VGAHI;
initgraph ( &grDriver, &grMode, "C:\\BORLANDC\\BGI" );
triangle ( 100, 100 );
triangle ( 140, 100 );
triangle ( 140, 100 );
getch();
closegraph();
}
void triangle ( int x, int y )
{
line ( x, y, x + 20, y - 50 );
line ( x, y, x + 40, y );
line ( x + 40, y, x + 20, y - 50 );
}
```

объявление процедуры

вызовы процедуры

тело процедуры

Что новенького?

- Процедура оформляется так же, как основная программа: заголовок и тело процедуры в фигурных скобках.
- Перед именем процедуры ставится слово `void`. Это означает, что она не возвращает результат, а только выполняет какие-то действия.
- После имени в скобках через запятую перечисляются ***параметры*** процедуры — те величины, от которых зависит ее работа.
- Для каждого параметра отдельно указывается его тип (`int, float, char`).
- Имена параметров можно выбирать любые, допустимые в языке Си.
- Параметры, перечисленные в заголовке процедуры, называются ***формальными*** — это значит, что они доступны только внутри процедуры при ее вызове.
- Желательно выбирать осмысленные имена параметров процедуры — это позволяет легче разобраться в программе потом, когда уже все забыто.
- При вызове процедуры надо указать ее имя и в скобках ***фактические*** параметры, которые подставляются в процедуре вместо формальных параметров.
- Фактические параметры — это числа или любые арифметические выражения (в этом случае сначала рассчитывается их значение).
- Первый фактический параметр подставляется в процедуре вместо первого формального параметра, и т.д.
- Все процедуры необходимо объявить *до* основной программы; это делается для того, чтобы к моменту вызова процедуры транслятор знал, что есть такая процедура, а также сколько она имеет параметров и каких. Это позволяет находить ошибки на трансляции, например такие:

`triangle ( 100 );`	**Too few parameters** (слишком мало параметров)
`triangle ( 100, 100, 5 );`	**Extra parameter** (лишний параметр)

- При объявлении процедуры после заголовка ставится точка с запятой, а в том месте, где записано тело процедуры — не ставится.
- Часто процедуры вызываются только один раз — в этом случае их задача — разбить большую основную программу (или процедуру) на несколько самостоятельных частей, поскольку рекомендуется, чтобы каждая процедура была длиной не более 50 строк (2 экрана по 25 строк), иначе очень сложно в ней разобраться.
- Для досрочного выхода из процедуры используется оператор `return`**,** при его выполнении работа процедуры заканчивается.
- Если оператор `return` стоит в основной программе, программа заканчивает работу.
- В процедуре можно использовать несколько операторов `return`: при выполнении любого из них работа процедуры заканчивается.

Улучшение процедуры

Можно попробовать сделать процедуру более универсальной: менять цвет границы, залить треугольник выбранным цветом, менять его высоту и ширину основания. Это означает, что надо ввести в процедуру дополнительные параметры. Их количество не должно быть очень

большим (более 8-10), потому что в этом случае можно легко перепутать их порядок. В нашей улучшенной процедуре будет 6 параметров:

x, y	координаты левого нижнего угла
a, h	длина основания и высота треугольника
color	цвет линий
fillColor	цвет заливки внутренней части

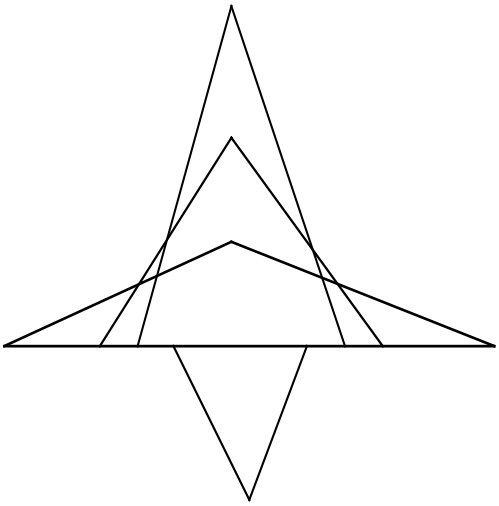
```
void triangle ( int x, int y, int a, int h, int color,
                int fillColor )
{
setcolor ( color );
line ( x, y, x + a / 2, y - h );
line ( x, y, x + a, y );
line ( x + 40, y, x + a / 2, y - h );
setfillstyle ( SOLID_FILL, fillColor );
floodfill ( x + a / 2, y - h / 2, color);
}
```

изменить цвет линии

изменить цвет заливки

заливка

Это позволит нам рисовать значительно более сложные фигуры и раскрашивать их, например, так:



```
triangle ( 100, 100, 40, 60,
           BLUE, GREEN);
triangle ( 90, 100, 60, 40,
           LIGHTBLUE,CYAN);
triangle ( 70, 100, 100, 20,
           LIGHTCYAN,RED);
triangle ( 110, 100, 20, -30,
           YELLOW,WHITE);
```

Из этого примера видно, как сделать, чтобы развернуть треугольник не вверх, а вниз: достаточно просто поставить отрицательную высоту при вызове процедуры.

8. Функции

Отличие функций от процедур

Функции, также как и процедуры, предназначены для выполнения одинаковых операций в разных частях программы. Они имеют одно существенное отличие: задача процедуры вычислить и вернуть в вызывающую программу *значение-результат* (в простейшем случае это целое, вещественное или символьное значение).

Функция - это вспомогательная программа, предназначенная для вычисления некоторой величины. Она также может выполнять какие-то полезные действия.

Покажем использование функции на примере. Решим задачу, которую мы уже решали раньше.

Задача. Написать программу, которая вводит целое число и определяет сумму его цифр. Использовать функцию, вычисляющую сумму цифр числа.

Вспомним, что для того чтобы найти последнюю цифру числа, надо взять остаток от его деления на 10. Затем делим число на 10, отбрасывая его последнюю цифру, и т.д. Сложив все эти остатки-цифры, мы получим сумму цифр числа.

```
#include <stdio.h>
#include <conio.h>
int SumDigits ( int N )
{
int d, sum = 0;
while ( N != 0 )
  {
  d = N % 10;
  sum = sum + d;
  N = N / 10;
  }
return sum;
}
void main()
{
int     N, s;
printf ( "\nВведите целое число ");
scanf ( "%d", &N );
s = SumDigits (N);
printf ( "Сумма цифр числа %d равна %d\n", N, s );
getch();
}
```

тело функции

вызов функции

Что новенького?

- Функция оформляется так же, как процедура: заголовок и тело функции в фигурных скобках.
- Перед именем процедуры ставится ***тип результата*** (`int, float, char,` и т.д.) — это означает, что она возвращает значение указанного типа.
- После имени в скобках через запятую перечисляются ***параметры*** функции — те величины, от которых зависит ее работа.
- Для каждого параметра отдельно указывается его тип (`int, float, char`).

- Имена параметров можно выбирать любые, допустимые в языке Си.
- Параметры, перечисленные в заголовке функции, называются ***формальными*** — это значит, что они доступны только внутри функции при ее вызове.
- Желательно выбирать осмысленные имена параметров — это позволяет легче разобраться в программе потом.
- При вызове функции надо указать ее имя и в скобках ***фактические*** параметры, которые подставляются в функции вместо формальных параметров.
- Фактические параметры — это числа или любые арифметические выражения (в этом случае сначала рассчитывается их значение).
- Первый фактический параметр подставляется в функции вместо первого формального параметра, и т.д.
- Для того, чтобы определить значение функции, используется оператор `return`, после которого через пробел записывается возвращаемое значение – число или арифметическое выражение. Примеры:

```
return  34
return s;
return a + 4*b - 5;
```

  При выполнении оператора `return` работа процедуры заканчивается.
- В функции можно использовать несколько операторов `return`.
- Если функции находятся ниже основной программы, их необходимо объявить *до* основной программы. Для объявления функции надо написать ее заголовок с точкой с запятой в конце.
- При объявлении функции после заголовка ставится точка с запятой, а в том месте, где записано тело функции — не ставится.

Логические функции

Очень часто надо составить функцию, которая просто решает какой-то вопрос и отвечает на вопрос "Да" или "Нет". Такие функции называются *логическими*. Вспомним, что в Си ноль означает ложное условие, а единица – истинное.

Логическая функция - это функция, возвращающая **1** (если ответ "Да") или **0** (если ответ "Нет").

Логические функции используются в основном в двух случаях:

- Если надо проанализировать ситуацию и ответить на вопрос, от которого зависят дальнейшие действия программы.
- Если надо выполнить какие-то сложные операции и определить, была ли при этом какая-то ошибка.

Простое число или нет?

Задача. Ввести число ***N*** и определить, простое оно или нет. Использовать функцию, которая отвечает на этот вопрос.

Теперь расположим тело функции ниже основной программы. Чтобы транслятор знал об это функции во время обработки основной программы, надо объявить ее заранее.

```
#include <stdio.h>
#include <conio.h>
int Simple ( int N );
//--- основная программа ---
void main()
{
int     N;
printf ( "\nВведите целое число ");
scanf ( "%d", &N );
if  ( Simple (N) )
     printf ( "Число %d - простое\n", N );
else printf ( "Число %d - составное\n", N );
getch();
}
//--- функция ---
int Simple ( int N )
{
for ( int i = 2; i*i <= N; i ++ )
   if ( N % i = =  0 ) return 0;
return 1;
}
```

объявление функции

вызов функции

нашли делитель — не простое

нет ни одного делителя

Была ли ошибка?

Задача. Включить монитор в графический режим и в случае ошибки выдать сообщение и завершить работу.

Идея основана на том, что существует функция `graphresult`, которая возвращает *код ошибки* — если она возвращает 0, то ошибки не было, а если не нуль, то включить графический режим не удалось (с помощью функции `grapherrormsg` можно получить текстовое описание ошибки на английском языке).

```
#include <graphics.h>
#include <conio.h>
void main()
{
int     grDriver = VGA, grMode = VGAHI, errorCode;
initgraph ( &grDriver, &grMode, "C:\\BORLANDC\\BGI" );
errorCode = graphresult();
if ( errorCode ) {
    printf ( "Ошибка: %s",  grapherrormsg (errorCode) );
    return;
    }
getch();
closegraph();
}
```

получить код ошибки

вывести сообщение об ошибке

выйти из программы

Функции, возвращающие два значения

По определению функция может вернуть только одно значение-результат. Если надо вернуть два и больше результатов, приходится использовать специальный прием — ***передачу параметров по ссылке***.

Задача. Написать функцию, которая определяет максимальное и минимальное из двух целых чисел.

В следующей программе используется достаточно хитрый прием: мы сделаем так, чтобы функция изменяла значение переменной, которая принадлежит основной программе. Один результат (минимальное из двух чисел) функция вернет как обычно, а второй – за счет изменения переменной, которая передана из основной программы.

```
#include <stdio.h>
#include <conio.h>
int MinMax ( int a, int b, int &Max )
{
if ( a > b )
     { Max = a; return b; }
else { Max = b; return a; }
}
void main()
{
int     N, M, min, max;
printf ( "\nВведите 2 целых числа ");
scanf ( "%d%d", &N, &M );
min = MinMax ( N, M, max );
printf ( "Наименьшее из них %d, наибольшее – %d\n",
          min, max );
getch();
}
```

параметр — результат

вызов функции

Обычно при передаче параметра в процедуру или функцию в памяти создается копия переменной и процедура работает с этой копией. Это значит, что все изменения, сделанные в процедуре с переменной-параметром, не отражаются на значении этой переменной в вызывающей программе.

Если перед именем параметра в заголовке функции поставить знак ***&*** (вспомним, что он также используется для определения адреса переменной), то процедура работает прямо с переменной из вызывающей программы, а не с ее копией. Поэтому в нашем примере процедура изменит значение переменой ***max*** из основной программы и запишет туда максимальное из двух чисел.

Что новенького?

- Если надо, чтобы функция вернула два и более результатов, поступают следующим образом:
  ⇒ один результат передается как обычно с помощью оператора `return`
  ⇒ остальные возвращаемые значения передаются через изменяемые параметры
- Обычные параметры не могут изменяться процедурой, потому что она в самом деле работает с *копиями* параметров (например, если менять значения ***a*** и ***b*** в процедуре ***MinMax***, соответствующие им переменные ***N*** и ***M*** в основной программе не изменятся).
- Любая процедура и функция может возвращать значения через изменяемые параметры.
- Изменяемые параметры (или параметры, передаваемые по ссылке) объявляются в заголовке процедуры специальным образом: перед их именем ставится знак ***&*** — в данном случае он означает ссылку, то есть процедура может менять значение параметра (в данном случае функция меняет значение переменной ***max*** в основной программе).
- При вызове таких функций и процедур вместо фактических изменяемых параметров надо подставлять только имя переменной (не число и не арифметическое выражение — в этих случаях транслятор выдает предупреждение и формирует в памяти временную переменную).

9. Структура программ

Составные части программы

В составе программы можно выделить несколько частей:

- Подключение *заголовочных файлов* — это строки, которые начинаются с `#include`
- Объявление *констант* (строки `#define` или объявления `const`)

  `#define N 20` или так: `const N = 20;`
- *Глобальные переменные* — это переменные, объявленные вне основной программы и процедур. К таким переменным могут обращаться все процедуры и функции данной программы (их не надо еще раз объявлять в этих процедурах).
- *Объявление функций и процедур* — обычно ставятся выше основной программы. По требованиям языка Си в тот момент, когда транслятор находит вызов процедуры, она должна быть объявлена и известны типы всех ее параметров.
- *Основная программа* — она может располагаться как до всех процедур, так и после них. Не рекомендуется вставлять ее в середину, между процедурами, так как при этом ее сложнее найти.

Глобальные и локальные переменные

Глобальные переменные доступны из любой процедуры или функции. Поэтому их надо объявлять вне всех процедур. Остальные переменные, объявленные в процедурах и функциях, называются *локальными* (местными), поскольку они известны только той процедуре, где они объявлены. Следующий пример показывает различие между локальными и глобальными переменными.

```
#include <stdio.h>
int var = 0;
void ProcNoChange ()
{
int var;
var = 3;
}
void ProcChange1 ()
{
var = 5;
}
void ProcChange2 ()
{
int var;
var = 4;
::var = ::var * 2 + var;
}
void main()
{
ProcChange1();          // var = 5;
ProcChange2();          // var = 5*2 + 4 = 14;
ProcNoChange();         // var не меняется
printf ( "%d", var );  // печать глобальной переменной
}
```

объявить глобальную переменную

меняется только локальная переменная

меняется только глобальная переменная

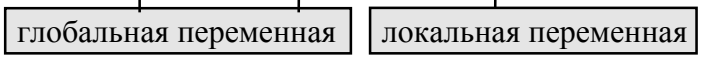
меняется локальная переменная

меняется глобальная переменная

Что новенького?

- Глобальные переменные не надо заново объявлять в процедурах.
- Если в процедуре объявлена локальная переменная с таким же именем, как и глобальная переменная, то используется ***локальная*** переменная.
- Если имена глобальной и локальной переменных совпадают, то для обращения к глобальной переменной в процедуре перед ее именем ставится два двоеточия:

```
::var = ::var * 2 + var;
```



Однако специалисты рекомендуют использовать очень мало глобальных переменных или не использовать их вообще, потому что глобальные переменные

- затрудняют анализ и отладку программы
- повышают вероятность серьезных ошибок — можно не заметить, что какая-то процедура изменила глобальную переменную
- увеличивают размер программы, так как заносятся в блок данных, а не создаются в процессе выполнения программы

Поэтому глобальные переменные используют в крайних случаях:

- для хранения глобальных системных настроек (цвета экрана и т.п.)
- если переменную используют три и более процедур и по каким-то причинам неудобно передавать эти данные в процедуру как параметр

Везде, где можно, надо передавать данные в процедуры и функции через их параметры. Если же надо, чтобы процедура меняла значения переменных, надо передавать параметр по ссылке.

Оформление текста программы

Зачем оформлять программы?

Зачем же красиво и правильно оформлять тексты программ? На этот вопрос вы сможете ответить сами, сравнив две абсолютно одинаковые (с точки зрения транслятора) программы:

```
#include <stdio.h>
void main()
{
float      x, y;
printf ("\nВведите 2 числа ");
scanf  ("%d%d", &x, &y);
printf ("Их сумма %d ", x+y);
}
```

```
#include <stdio.h> void
main() { float  x, y;
printf ( "\nВведите 2
числа " ); scanf (
"%d%d", &x, &y ); printf
( "Их сумма %d ", x+y );}
```

То, что в них отличается и называется грамотным оформлением (очевидно, что оно присутствует в первой программе).

Оформление текста программы необходимо для того, чтобы
- отлаживать программу (искать и исправлять ошибки в ней)
- разбираться в алгоритме работы программы

Оформление процедур и функций

При оформлении функций и процедур рекомендуется придерживаться следующих правил:

- Имена функций и процедур должны быть информативными, то есть нести информацию о том, что делает эта функция. К сожалению, транслятор не понимает русские имена, поэтому приходится писать по-английски. Если вам сложно писать имена на английском языке, можно писать русские слова английскими буквами. Например, процедуру, рисующую квадрат, можно объявить так:

```
void Square ( int x, int y, int a );
```

или так

```
void Kvadrat ( int x, int y, int a );
```

- Перед заголовком процедуры надо вставлять несколько строк с комментарием (здесь можно писать по-русски). В комментарий записывается самая важная информация: что означают параметры функции, что она делает и какое значение она возвращает, ее особенности.
- Одинаковые операции в разных частях программы оформляются в виде процедур и функций
- Не рекомендуется делать функции длиной более 25-30 строк, так как при этом они не умещаются на один экран монитора (25 строк), становятся сложными и запутанными. Если функция получается длинной, ее надо разбить на более мелкие процедуры и функции.
- Чтобы отделить одну часть функции от другой используют пустые строки. Крупный смысловой блок функции можно выделять строкой знаков минус в комментариях.

Приведем пример оформления на примере функции, которая рисует на экране закрашенный ромб с заданными параметрами, если весь ромб умещается на экран (при этом результат функции равен 1) или возвращает признак ошибки (число 0).

```
//*********************************************
// ROMB — рисование ромба в заданной позиции
//      (x,y)    — координаты центра ромба
//      a, b — ширина и высота ромба
//      color, colorFill — цвета границы и заливки
//      Возвращает 1, если операция выполнена, и 0 если
//      ромб выходит за пределы экрана
//*********************************************
int Romb ( int x, int y, int a, int b, int color,
           int colorFill )
{
   if ( (x < a) || (x > 640-a) || (y < a) || (y > 480-b) )
             return 0;
//-----------------------------------
   setcolor ( color );
   line ( x-a, y, x, y-b );      line ( x-a, y, x, y+b );
   line ( x+a, y, x, y-b ); line ( x+a, y, x, y+b );

   setfillstyle ( SOLID_FILL, colorFill );
   floodfill ( x, y, color );

   return 1;
}
```

заголовок

обработка ошибки

крупный блок

выделение пустой строкой

Отступы

Отступы используются для выделения структурных блоков программы (процедур, циклов, условных операторов). Отступы позволяют легко искать лишние и недостающие скобки, понимать логику работы программы и находить в ней ошибки. При расстановке отступов рекомендуется соблюдать следующие правила:

- Величина отступа равна 2 — 3 символа.
- Все тело любой процедуры и функции имеет один отступ.
- Дополнительным отступом выделяются
  ⇒ тело циклов `for, while, do-while`
  ⇒ тело условного оператора `if` и блока `else`
  ⇒ тело оператора множественного выбора `switch`
- Парные скобки должны находиться на одной вертикали (чтобы найти скобку, парную данной, надо поставить на эту скобку курсор и нажать на клавиши ***Ctrl-Q*** и затем ***{*** или ***}.***

10. Анимация

Что такое анимация?

Анимация – это оживление изображения на экране (от английского слова *animate* - оживлять). При этом объекты движутся, вращаются, сталкиваются, меняют форму и цвет и т.д.

Чтобы сделать программу с эффектами анимации, надо решить две задачи:

- двигать объект так, чтобы он мигал как можно меньше
- обеспечить управление клавишами или мышкой во время движения

В этом разделе рассматриваются самые простые задачи этого типа. Во всех программах предусмотрен выход по клавише ***Escape***.

Движение объекта

Составим программу, которая передвигает по экрану какой-то объект (в нашем случае – равнобедренный треугольник) от левой границы экрана к правой. Программа заканчивает работу, когда объект вышел за границы экрана или была нажата клавиша ***Escape***.

Предварительный анализ

Рассмотрим объект, который движется по экрану. Пусть это будет равнобедренный треугольник с основанием 20 пикселов и высотой также 20 пикселов. Он будет двигаться по экрану, поэтому координаты всех вершин будут меняться. Чтобы строить треугольник в любом месте экрана, мы выберем одну из точек в треугольнике в качестве базовой (главной) и обозначим ее координаты за ***(x,y).*** Теперь выразим через них координаты всех вершин (см. рисунок).

Теперь надо придумать способ изобразить движение так, чтобы рисунок не мигал и программа работала одинаково на всех компьютерах независимо от и быстродействия. Для этого применяют такой алгоритм:

1. рисуем фигуру на экране
2. делаем небольшую задержку (обычно 10-20 мс)
3. стираем фигуру
4. меняем ее координаты
5. переходим к началу алгоритма

Эти действия повторяются до тех пор, пока не будет получена команда «закончить движение» (в нашем случае - нажата клавиша ***Excape*** или объект вышел за правую границу экрана).

Пусть движение треугольника происходит на черном фоне. Тогда самый быстрый и простой способ стереть его – это нарисовать его же, но черным цветом. Поэтому удобно написать процедуру, параметрами которой являются координаты ***x*** и ***y***, а также цвет линий ***color***. Когда мы используем черный цвет, фигура стирается с экрана.

Все действия, которые входят в алгоритм, надо выполнить много раз, поэтому используем цикл. Кроме того, мы заранее не знаем, сколько раз должен выполниться этот цикл, поэтому применяем цикл `while` (цикл с условием).

Условие окончания цикла – выход фигуры за границы экрана или нажатие на клавишу ***Escape***. Мы будем работать в графическом режиме с разрешением 640 на 480 пикселов, где

координата ***x*** может меняться от 0 до 639, поэтому условие выхода треугольника за границы экрана имеет вид ***x + 20 >= 640***, а нужное нам условие продолжения цикла:

```
x + 20 < 640
```

Обработка событий клавиатуры

Надо также обеспечить выход по клавише ***Escape***. При этом объект должен двигаться и нельзя просто ждать нажатия на клавишу с помощью функции `getch`. В этом случае используют следующий алгоритм:

1. Проверяем, нажата ли какая-нибудь клавиша; это делает функция `kbhit`, которая возвращает результат 1 (ответ «да»), если клавиша нажата, и результат 0 (ответ «нет»), если никакая клавиша не нажата. В программе проверка выполнена с помощью условного оператора

   ```
   if ( kbhit() ) { ... }
   ```

2. Если клавиша нажата, то
   - Определяем код этой клавиши, вызывая функцию `getch`. Код клавиши – это ее номер в таблице символов. Если на символ отводится 1 байт памяти, то всего можно использовать 256 разных символов и их коды изменяются в интервале от 0 до 255.
   - Если полученный код равен коду клавиши ***Escape*** (27), то выходим из цикла

Для того, чтобы управлять программой с помощью клавиш, надо использовать их коды. Вот некоторые из них

Escape	27
Enter	13
пробел	32

Программа

Вообще говоря, в процедуре можно рисовать любую фигуру. В нашем случае рисуем равнобедренный треугольник. Если процедура расположена ниже основной программу, ее надо объявить заранее.

Первые две строчки основной программы – включение графического режима. Затем объявляются переменные ***x, y*** – координаты фигуры, и ***dx*** – величина смещения фигуры за 1 шаг. После этого устанавливаем начальные значения всех переменных.

В цикле `while` условием продолжения будет ***x + 20 < 640***, то есть выполнять пока фигура находится в пределах экрана. Нажатие на клавишу ***Escape*** обрабатывается внутри цикла. Сначала мы определяем, нажата ли какая-нибудь клавиша (с помощью функции `kbhit`), затем определяем ее код (функция `getch`) и, если он равен коду ***Escape***, выходим из цикла с помощью оператора `break`.

В основной части цикла рисуем фигуру с помощью процедуры, затем делаем задержку на 20 мс, вызывая функцию `delay` с параметром 20, и стираем фигуру. Для использования функции `delay` надо подключить дополнительную библиотеку функций, добавив в начало программы строчку

```
#include <dos.h>
```

После этого изменяем координаты и возвращаемся к началу цикла. Переменная ***dx*** обозначает величину смещения за 1 шаг. Если увеличить ***dx***, фигура будет двигаться быстрее, но скачками. Заметим, что если ***dx*** станет отрицательным, то фигура будет двигаться влево, потому что ее координата `x` уменьшается.

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>
void Figure ( int x, int y, int color )
{
   setcolor ( color );
   line ( x, y, x+20, y );
   line ( x, y, x+10, y-20 );
   line ( x+10, y-20, x+20, y );
}
void main()
{
   int d = VGA, m = VGAHI;
   int x, y, dx;
   initgraph ( &d, &m, "c:\\borlandc\\bgi" );
   x = 0; y = 240;
   dx = 1;
   while ( x + 20 < 640 )
      {
      if ( kbhit() )
        if ( getch() == 27 ) break;
      Figure ( x, y, YELLOW );
      delay ( 20 );
      Figure ( x, y, BLACK );
      x += dx;
      }
   closegraph();
}
```

функция рисует

начальные координаты

шаг

если нажата клавиша…

если нажали ***Esc***, выход

рисуем фигуру

задержка 20 мс

стираем фигуру

двигаем фигуру

Что новенького?

- Чтобы определить нажата ли какая-нибудь клавиша, используется функция `kbhit`. Она возвращает 1, если клавиша нажата, или 0, если нет.
- Если клавиша уже была нажата, ее код можно получить с помощью функции `getch`.
- Чтобы сделать задержку на заданное время, используется процедура `delay`, которая описана в файле `dos.h`. Параметром этой процедуры является величина задержки в миллисекундах. Если уменьшать задержку, фигура будет двигаться быстрее.

Отскок от поверхности

Теперь сделаем так, чтобы треугольник не уходил за границы экрана, а «отскакивал» от них и начинал движение в другую сторону, и так до тех пор, пока не нажата клавиша ***Escape***.

В предыдущей программа была использована переменная ***dx***, которая задает смещение фигуры за 1 шаг. Если ***dx>0,*** то фигура движется вправо (координата `x` увеличивается), а если ***dx<0,*** то влево. Таким образом, чтобы сделать отскок от правой стенки, надо в момент «столкновения» (то есть, при ***x+20>=639***) поменять ***dx*** на некоторое отрицательное значение. Аналогично, во время отскока от левой стенки (то есть, при ***x<=0***) значение ***dx*** должно стать положительным.

```
while ( 1 )                          бесконечный цикл
   {
   if ( kbhit() )
      if ( getch() == 27 ) break;
   Figure ( x, y, YELLOW );
   delay ( 10 );                                  меняем направление
   Figure ( x, y, BLACK );
   if ( x + 20 >= 639 ) dx = - 1;
   if ( x <= 0 )        dx = 1;
   x += dx;
   }
```

Здесь показан только основной цикл, поскольку оставшаяся часть программы не изменяется.

Заметим, что изменился заголовок цикла `while`, теперь вместо условия стоит единица. В языке Си это означает истинное условие, то есть всегда верное. Таким образом, мы получили бесконечный цикл. Чтобы выйти из него, надо нажать клавишу ***Esc***. При этом программа выходит на оператор `break` и цикл заканчивается.

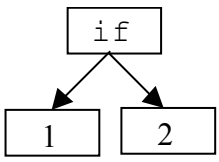
Управление клавишами-стрелками

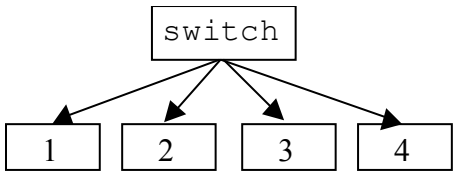
Предварительный анализ

Принцип работы программы очень простой: получив код клавиши, надо сдвинуть объект в соответствующую сторону. Мы введем две переменные ***dx*** и ***dy***, которые будут обозначать величину изменения координат фигуры `x` и `y` за 1 шаг цикла. У нас обрабатываются 4 варианта:

движение влево	***dx > 0, dy = 0***
движение вправо	***dx < 0, dy = 0***
движение вверх	***dx = 0, dy < 0***
движение вниз	***dx = 0, dy > 0***

Это значит, что надо сделать выбор одного из четырех вариантов в зависимости от кода нажатой клавиши. Для этого можно использовать несколько условных операторов `if`, но существует специальный оператор `switch`, который позволяет легко организовать выбор из нескольких вариантов.





Еще одна проблема связана с тем, что клавиши управления курсором (стрелки) – не совсем обычные клавиши. Они относятся к группе функциональных клавиш, у которых нет кодов в таблице символов ***ASCII***. Когда нажата одна из специальных клавиш, система реагирует на нее как на 2 нажатия, причем для первого код символа всегда равен нулю, а для второго мы получим специальный код (так называемый *скан-код*, номер клавиши на клавиатуре). Мы будем использовать упрощенный подход, когда анализируется только этот второй код:

влево	**75**	**вверх**	**72**
вправо	**77**	**вниз**	**80**

У такого приема есть единственный недостаток: объект будет также реагировать на нажатие клавиш с кодами 75, 77, 72 и 80 в таблице символов, то есть на заглавные латинские буквы K, M, H и P.

Простейший случай

Составим программу, при выполнении которой фигура будет двигаться только тогда, когда мы нажмем на клавишу-стрелку. В цикле мы сначала рисуем фигуру, ждем нажатия на клавишу и принимаем ее код с помощью функции `getch`. После этого стираем фигуру в том же месте (пока не изменились координаты) и, в зависимости от этого кода, меняем координаты фигуры нужным образом.

```
#include <graphics.h>
#include <conio.h>
void Figure ( int x, int y, int color )
{
... // здесь записываем ту же самую функцию, что и раньше
}
void main()
{
   int d = VGA, m = VGAHI;
   int x, y, key;
   initgraph ( &d, &m, "c:\\borlandc\\bgi" );
   x = 320; y = 240;
   while ( 1 )
      {
      Figure ( x, y, YELLOW );
      key = getch();
      if ( key == 27 ) break;
      Figure ( x, y, BLACK );
      switch ( key ) {
        case 75: x --; break;
        case 77: x ++; break;
        case 72: y --; break;
        case 80: y ++;
        }
      }
   closegraph();
}
```

начать с центра экрана

рисуем фигуру

ждем нажатия на клавишу

стираем фигуру

двигаем фигуру

В операторе `switch` значения координат меняются на единицу, хотя можно использовать любой шаг. В конце обработки каждого варианта надо ставить оператор `break`, чтобы не выполнялись строки, стоящие ниже.

Непрерывное движение

Теперь рассмотрим более сложный случай, когда объект продолжает движение в выбранном направлении даже тогда, когда ни одни клавиша не нажата, а при нажатии клавиши-стрелки меняет направление. Здесь надо использовать переменные ***dx*** и ***dy***, которые задают направление движения. Сначала мы определяем, нажата ли клавиша, а затем определяем ее код, записываем его в переменную ***key***, и обрабатываем это нажатие с помощью оператора `switch`.

```
#include <graphics.h>
#include <conio.h>
#include <dos.h>
void Figure ( int x, int y, int color )
{
... // здесь записываем ту же самую функцию, что и раньше
}
void main()
{
   int d = DETECT, m;
   int x, y, dx, dy, key;
   initgraph ( &d, &m, "c:\\borlandc\\bgi" );
   x = 320; y = 240;
   dx = 1;  dy = 0;
   while ( 1 )
      {
      if ( kbhit() ) {
        key = getch();
        if ( key == 27 ) break;
        switch ( key ) {
           case 75: dx = - 1; dy = 0; break;
           case 77: dx = 1;   dy = 0; break;
           case 72: dx = 0;   dy = - 1; break;
           case 80: dx = 0;   dy = 1;
           }
        }
      Figure ( x, y, YELLOW );
      delay ( 10 );
      Figure ( x, y, BLACK );
      x += dx;
      y += dy;
      }
   closegraph();
}
```

начать с центра экрана, двигаться влево

этот блок выполняется только тогда, когда нажата какая-нибудь клавиша

11. Случайные и псевдослучайные числа

Что такое случайные числа?

Представьте себе снег, падающий на землю. Допустим, что мы сфотографировали природу в какой-то момент. Сможем ли мы ответить на такой вопрос: куда точно упадет следующая снежинка? Навряд ли, потому что это зависит от многих причин — от того, какая снежинка ближе к земле, как подует ветер и т.п. Можно сказать, что снежинка упадет в *случайное место*, то есть в такое место, которое нельзя предсказать заранее.

Для моделирования случайных процессов (типа падения снежинок, броуновского движения молекул вещества и т.п.) на компьютерах применяют ***случайные числа***.

Случайные числа - это такая последовательность чисел, в которой невозможно назвать следующее число, зная сколько угодно предыдущих.

Получить случайные числа на компьютере достаточно сложно. Иногда для этого применяют различные источники радиошумов. Однако математики придумали более универсальный и удобный способ — *псевдослучайные числа.*

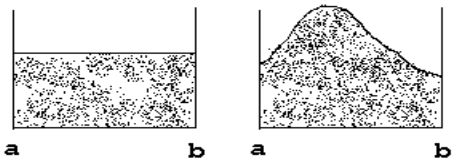
Псевдослучайные числа - это последовательность чисел, обладающая свойствами, близкими к свойствам случайных чисел, в которой каждое следующее число вычисляется на основе предыдущих по специальной математической формуле.

Таким образом, эта последовательность ведет себя так же, как и последовательность случайных чисел, хотя, зная формулу, мы можем получить следующее число в последовательности.

Распределение случайных чисел

Обычно используют псевдослучайные числа (далее для краткости мы называем их просто случайными), находящиеся в некотором интервале. Например, представим себе, что снежинки падают не на всю поверхность земли, а на отрезок оси **`OX`** от **`a`** до **`b`**. При этом очень важно знать некоторые ***общие свойства*** этой последовательности. Если понаблюдать за снежинками в безветренную погоду, то слой снега во всех местах будет примерно одинаковый, а при ветре — разный. Про случайные числа в первом случае говорят, что они имеют ***равномерное распределение***, а во втором случае — ***неравномерное***.

равномерное распределение



неравномерное распределение

Большинство стандартных датчиков псевдослучайных чисел (то есть формул, по которым они вычисляются) дают равномерное распределение в некотором интервале.

Поскольку случайные числа в компьютере вычисляются по формуле, то для того, чтобы повторить в точности какую-нибудь случайную последовательность достаточно просто взять то же самое начальное значение.

Функции для работы со случайными числами

В языке Си существуют следующие функции для работы со случайными числами (их описание находится в заголовочном файле ***stdlib.h*** — это значит, что его необходимо подключать в начале программы):

`n = rand();`	получить случайное целое число в интервале от 0 до ***RAND_MAX*** (это очень большое целое число — 32767)
`n = random ( max );`	получить случайное целое число в интервале от 0 до ***max-1***
`srand ( m );`	установить новое начальное значение случайной последовательности
`randomize();`	установить случайное начальное значение случайной последовательности

Случайные числа в заданном интервале

Для практических задач надо получать случайные числа в заданном интервале `[a,b]`. Важно уметь получать последовательности как целых, так и вещественных случайных чисел. С помощью случайных чисел решаются очень многие задачи, которые очень трудно (или невозможно) решить другими способами. Например, так моделируется движение молекул в физике и биологии.

Рассмотрим сначала целые числа. Если `random(n)` генерирует случайное число в интервале `[0,n-1]`, то очевидно, что

```
k = random(n) + a;
```

дает последовательность в интервале `[a,a+n-1]`. Поскольку нам нужно получить интервал `[a, b]`, мы имеем `b=a+n-1`, откуда `n=b-a+1`. Поэтому

> Для получения случайных целых чисел с равномерным распределением в интервале `[a,b]` надо использовать формулу
>
> ```
> k = random(b-a+1) + a;
> ```

Более сложным оказывается вопрос о случайных вещественных числах. Самое лучшее, что мы можем сделать, это разделить результат функции `rand()` на `RAND_MAX`:

```
x = (float) rand() / RAND_MAX;
```

и получить таким образом, случайное вещественное число в интервале `[0,1]` **(**при этом надо не забыть привести одно из этих чисел к вещественному типу, иначе деление одного целого числа на большее целое число будет всегда давать ноль**)**.

Длина интервала такой последовательности равна ***1***, а нам надо получить в конечном счете интервал длиной ***b-a***. Поэтому если теперь это число умножить на ***b-a*** и добавить к результату ***a***, мы получаем как раз нужный интервал.

> Для получения случайных вещественных чисел с равномерным распределением в интервале `[a,b]` надо использовать формулу
>
> ```
> x = rand()*(b-a)/RAND_MAX + a;
> ```

До этого момента мы говорили только о получении случайных чисел с равномерным распределением. Как же получить неравномерное? На этот вопрос математика отвечает так: из равномерного распределения можно получить неравномерное, применив к этим данным

некоторую математическую операцию. Например, чтобы основная часть чисел находилась в середине интервала, можно брать среднее арифметическое нескольких последовательных случайных чисел с равномерным распределением.

Снег на экране

Приведенная ниже программа генерирует случайное значение ***x*** в интервале `[160,480]`, случайное значение ***y*** в интервале `[120,360]` и проверяет цвет точки с координатами `(x,y)`. Если эта точка черная, то ее цвет устанавливается случайный, а если нет — черный. Таким образом, разноцветный снег падает в центральный прямоугольник.

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>

void main()
{

int  d = VGA, m = VGAHI;
int  x, y, color;

initgraph ( &d, &m, "C:\\BORLANDC\\BGI" );

while ( ! kbhit () ) {

  x = random(320)+160;
  y = random(240)+120;

  color = random(16);

  if ( getpixel ( x, y ) != BLACK )
    putpixel ( x, y, BLACK );
  else
    putpixel ( x, y, color );
  }
getch();
closegraph();
}
```

цикл: пока не нажата клавиша

получить случайные координаты и цвет точки

проверить цвет точки

Что новенького?

- для определения того, была ли нажата какая-нибудь клавиша, используется функция `kbhit()`, которая возвращает 0, если клавиша не была нажата, и 1, если нажата клавиша. Для того, чтобы определить код этой клавиши, надо вызвать функцию `getch()`. Таким образом, цикл "пока не нажата клавиша" может выглядеть так:

```
while ( ! kbhit() ) { ... }
```

- для того, чтобы получить текущий цвет точки, используется функция `getpixel(x,y)`