Глава I.

Введение в программирование

1. *Простейшие программы*...... *3*
- **Зачем нужно программирование?......3**
- **Два этапа создания программ......3**
- **Простейшая программа на Си......4**
- **Вывод текста на экран......4**
- **Как запустить программу?......5**
- **Остановим мгновение......5**

2. *Переменные*...... *7*
- **Типы данных и переменные......7**
- **Вычисление суммы двух чисел (ввод и вывод)......7**
- **Арифметические выражения......9**
- **Форматы для вывода данных......11**

3. *Выбор вариантов*...... *14*
- **Зачем нужны ветвления?......14**
- **Условный оператор `if – else`......14**
- **Сложные условия......16**
- **Переключатель `switch` (множественный выбор)......17**

4. *Циклы*...... *19*
- **Зачем нужны циклы?......19**
- **Цикл с известным числом шагов (`for`)......19**
- **Цикл с условием (`while`)......20**
- **Цикл с постусловием (`do – while`)......22**
- **Досрочный выход из цикла......23**
- **Вычисление сумм последовательностей......24**

5. *Методы отладки программ*...... *27*
- **Отладочные средства *Dev-C++*......27**

6. *Работа в графическом режиме*...... *31*
- **Простейшая графическая программа......31**
- **Как начать рисовать?......31**
- **Пример программы......34**

7. *Процедуры*...... *35*
- **Пример задачи с процедурой......35**

8. *Функции*...... *38*
- **Отличие функций от процедур......38**
- **Логические функции......39**
- **Функции, возвращающие два значения......40**

9. *Структура программ*...... *42*
- **Составные части программы......42**
- **Глобальные и локальные переменные......42**
- **Оформление текста программы......43**

10. *Анимация*...... *46*
- **Что такое анимация?......46**
- **Движение объекта......46**

🗎 **Управление клавишами-стрелками 48**

11. Случайные и псевдослучайные числа 51

🗎 **Что такое случайные числа? 51**
🗎 **Распределение случайных чисел 51**
🗎 **Функции для работы со случайными числами 52**
🗎 **Случайные числа в заданном интервале 52**
🗎 **Снег на экране 53**

1. Простейшие программы

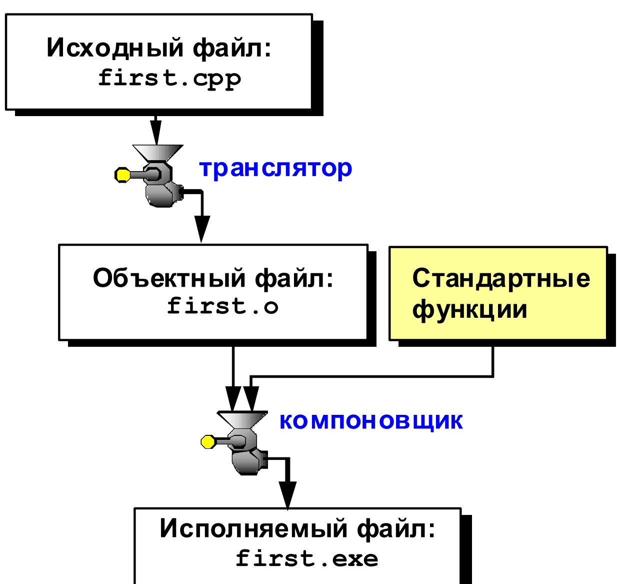
Зачем нужно программирование?

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, так же как и на большинстве современных языков программирования, создается в два этапа

1) **трансляция** – перевод текста программы в машинные коды;
2) **компоновка** – сборка частей программы и подключение стандартных функций.



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 (переводить в машинные коды) все остальные (это могут быть десятки тысяч строк);
- при компоновке во многих системах можно подключать модули, написанные на других языках, например, на Ассемблере (в машинных кодах).

Трансляторы языка Си называются **компиляторами**: они переводят (транслируют) сразу всю программу в машинный код, а не транслируют строчка за строчкой во время выполнения, как это делают **интерпретаторы**. Это позволяет значительно ускорить выполнение программы и не ставить интерпретатор на каждый компьютер, где программа будет выполняться.

Исходный файл программы на языке Си имеет расширение **`*.c`** или **`*.cpp`** (расширение **`*.cpp`** говорит о том, что в программе могут быть использованы возможности языка Си++). Это обычный текстовый файл, в который записывают текст программы в любом текстовом редакторе, например, в Блокноте.

Транслятор переводит исходный файл (вернее, записанную в нём программу) в машинные коды и строит так называемый *объектный файл* с тем же именем и расширением **`*.o`**. Хотя в нем уже записан машинный код, объектный файл еще нельзя запускать на компьютере, потому что в нем не хватает стандартных функций (например, для ввода и вывода данных).

Компоновщик подключает стандартные функции, хранящиеся в библиотеках (они имеют расширение **`*.a`**). В результате получается один файл с расширением **`*.exe`**, который и представляет собой готовую программу.

Простейшая программа на Си

Такая программа состоит всего из 8 символов. Вот она:

```
main()
{
}
```

Основная программа 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. Фигурные скобки обозначают начало и конец основной программы − 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>
main()
{
printf("Привет");
}
```

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

вызов функции вывода на экран

Что новенького?

- 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 включить в программу *описание* этой функции. Описания стандартных функций Си находятся в так называемых *заголовочных файлах* с расширением **`*.h`** (в каталоге **`C:\Dev-Cpp\include`**).

- Для подключения заголовочных файлов используется директива (команда) препроцессора[1] **`#include`**, после которой в угловых скобках ставится имя файла. Внутри угловых

[1] **Препроцессор** – это специальная программа, которая обрабатывает текст вашей программы раньше транслятора. Все команды препроцессора начиняются знаком **#**.

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).

В оболочке *Dev-C++* для запуска программы надо нажать клавишу **`F9`**. Если в программе есть ошибки, вы увидите в нижней части экрана оболочки сообщения об этих ошибках (к сожалению, на английском языке). Если щелкнуть по одной из этих строчек, в тексте программы выделяется строка, в которой транслятору что-то не понравилось.

При поиске ошибок надо помнить, что

- часто ошибка сделана не в выделенной строке, а в предыдущей – проверяйте и ее тоже;
- часто одна ошибка вызывает еще несколько, и появляются так называемые наведенные ошибки.

Остановим мгновение

Если запускать рассмотренную выше программу, то обнаружится, что программа сразу заканчивает работу и возвращается обратно в оболочку, не дав нам 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>
main()
{
printf("Привет");       // вывод на экран
getch();               /*ждать нажатия клавиши*/
}
```

подключение заголовочного файла **`conio.h`**

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 транслятором 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, вы (и компьютер) знаете, что с целыми и дробными числами работают по-разному. Поэтому в каждом языке программирования есть разные типы данных, для обработки которых используются разные методы. Например,

- **целые** переменные – тип **int** (от английского *integer* – целый), занимают 4 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

Любую переменную, которые вы будете использовать в программе, необходимо объявлять – сказать компьютеру, чтобы он выделил для неё ячейку памяти нужного размера и присвоил ей 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 значение, 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, I186=23, Yak42; // три целых переменных,
                           // в I186 сразу записывается число 23
float x=4.56, y, z;    // три вещественных переменных,
                        // в x сразу записывается число 4.56
char c, c2='A', m;     // три символьных переменных, 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.

```
#include <stdio.h>
#include <conio.h>
main()
{
int a, b, c;  // объявление переменных
printf ( "Введите два целых числа \n" ); // подсказка для ввода
scanf ( "%d%d", &a, &b );                // ввод данных
c = a + b;                 // вычисления (оператор присваивания)
printf ( "Результат: %d + %d = %d \n",
         a, b, c );                      // вывод результата
getch();
}
```

Что новенького?

- Программа чаще всего содержит 4 части:
  - объявление переменных;
  - ввод исходных данных;
  - обработка данных (вычисления);
  - вывод результата.
- 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 — адреса переменных

```
scanf ( "%d%d", &a, &b );
```

  - 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 переменных 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 );`	куда записывать второе введенное число?
`scanf ( "%d%d", &a, &b, &c );`	не задан формат для переменной **c**
`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;        // сумму a и b записать в c
```

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

эти символы вывести без изменений | здесь вывести целые числа | данные для вывода

```
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.

- Одной строки формата недостаточно: в ней сказано, в какое место выводить данные, но не сказано, откуда их взять. 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)	модуль вещественного числа **x**
sqrt(x)	квадратный корень из вещественного числа **x**
pow(x,y)	вычисляет **x** в степени **y**

- круглые скобки для изменения порядка действий

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:

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

Для изменения порядка выполнения операций используются круглые скобки. 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 вывода данных. Для функции **scanf**, которая выполняет ввод данных, достаточно просто указать один из форматов **%d**, **%f** или **%c** для ввода целого числа, вещественного 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]`	Минимально возможное поле.

printf("[%6d]", 1234);	**[1234]**	6 позиций, выравнивание вправо.
printf("[%-6d]", 1234);	**[1234]**	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]**	Минимально возможное поле, 6 знаков в дробной части.
printf("[%9.3f]", 123.45);	**[123.450]**	Всего 9 позиций, из них 3 – для дробной части, выравнивание вправо.
printf("[%-9.3f]", 123.45);	**[123.450]**	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]**	Минимально возможное поле, 6 знаков в дробной части мантиссы.
printf("[%12.3e]", 123.45);	**[1.234e+02]**	Всего 12 позиций, из них 3 – для дробной части мантиссы, выравнивание вправо.
printf("[%-12.3e]", 123.45);	**[1.234e+02]**	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]`
		Минимально возможное поле, 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]`
		Всего 10 позиций, из них 3 значащие цифры, выравнивание вправо. Чтобы сделать выравнивание влево, используют формат **`"%-10.3g"`**.

3. Выбор вариантов

Зачем нужны ветвления?

В простейших программах все команды выполняются одна за другой последовательно. Так реализуются *линейные* алгоритмы. Однако часто надо выбрать тот или иной вариант действий в зависимости от некоторых условий: если условие верно, поступать одним способом, а если неверно — другим. Для этого используют **разветвляющиеся алгоритмы**, которые в языках программирования представлены в виде **условных операторов**. 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 наибольшее из двух чисел сначала записывается в третью переменную **`Max`**.

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

Что новенького?

- Условный оператор имеет следующий вид:

```
if ( условие ) // заголовок с условием
   {
   ... // блок «если» — операторы, которые выполняются,
       // если условие в заголовке 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 знаки логических отношений

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

- В языке Си любое число, не равное нулю, обозначает истинное условие, а ноль — ложное условие.
- 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 )
   {
   ...    // что делать, если условие истинно
   }
```

Например, решение предыдущей задачи могло бы выглядеть так:

```
#include <stdio.h>
#include <conio.h>
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 (запись «лесенкой»).

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 в возрасте от 25 до 40 лет (вклчительно). Тогда простейшая программа могла бы выглядеть так:

```
#include <stdio.h>
#include <conio.h>
main()
{
int age;
printf ( "\nВведите ваш возраст: " );
scanf ( "%d", &age );
if ( 25 <= age  &&  age <= 40 )   // сложное условие
   printf ( "Вы нам подходите." );
else
   printf ( "Извините, Вы нам не подходите." );
getch();
}
```

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_2
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_2
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 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.

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>
main()
{
char c;
printf("\nВведите первую букву:");
scanf("%c", &c); // ввести букву
switch ( c )       // заголовок оператора выбора
  {
  case 'а': printf("\nАнтилопа"); break;
  case 'б': printf("\nБарсук"); break;
  case 'в': printf("\nВолк"); break;
  default:  printf("\nНе знаю я таких!"); // по умолчанию
  }
getch();
}
```

Что новенького?

- 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 этой переменной 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 ***а*** будет напечатано

```
Антилопа
Барсук
Волк
Не знаю таких!
```

- 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.

4. 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 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 ноль.

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>
main()
{
int i;                         // объявление переменной цикла
for ( i = 1; i <= 10; i ++ )   // заголовок цикла
   {                           // начало цикла (открывающая скобка)
   printf("Привет");           // тело цикла
   }                           // конец цикла (закрывающая скобка)
getch();
}
```

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 значения**: 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 (запись «лесенкой»).

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>
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 единицу (**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 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.

```
#include <stdio.h>
#include <conio.h>
main()
{
int N;          // число, с которым работаем
int count=0;  // переменная-счетчик
printf ( "\nВведите число N: " );   // подсказка
scanf ( "%d", &N );                 // ввод N с клавиатуры
while ( N > 0 ) // заголовок цикла «пока N > 0»
  {             // начало цикла (открывающая скобка)
  N /= 10;      // отсекаем последнюю цифру
  count ++;     // увеличиваем счетчик цифр
  }             // конец цикла (закрывающая скобка)
printf ( "В этом числе %d цифр\n", count );
getch();
}
```

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 это условие нарушается (становится ложно), 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 2-3 символа (запись «лесенкой»).

Цикл с постусловием (do — while)

Существуют также случаи, когда надо выполнить цикл хотя бы один раз, 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 ноль.

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, один раз обязательно 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>
main()
{
int N, sum;  // sum - сумма цифр числа
sum = 0;     // сначала сумму обнуляем
do {         // начало цикла
  printf ( "\nВведите натуральное число:" );
  scanf ( "%d", &N );
  }
while ( N <= 0 ); // условие цикла «пока N <= 0»
while ( N > 0 ) {
  sum += N % 10;
  N /= 10;
  }
printf ( "Сумма цифр этого числа равна %d\n", sum );
getch();
}
```

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 (запись «лесенкой»).

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){...}** (напомним, что в языке Си 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>
main()
{
int A, B;
while ( 1 )  // бесконечный цикл, выход только по break!
   {
   printf ( "\nВведите делимое и делитель:" );
   scanf ( "%d%d", &A, &B );
   if ( A == 0  &&  B == 0 ) break; // выход из цикла
   if ( B == 0 ) {
      printf ( "Деление на ноль" );
      continue; // досрочный переход к новому шагу цикла
      }
```

```
   printf ( "Частное %d остаток %d", A/B, A%B )
   }
getch();
}
```

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 любых циклов: **for, while, do—while.**
- Чтобы досрочно завершить текущий шаг цикла и сразу перейти к следующему шагу, используют оператор **continue.**

Вычисление сумм последовательностей

Суммы с заданным числом элементов

Задача. Найти сумму первых 20 элементов последовательности

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{8} - \frac{4}{16} + \cdots$$

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

- каждый элемент представляет собой дробь;
- числитель дроби при переходе к следующему элементу возрастает на единицу;
- знаменатель дроби с каждым шагом 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	4	8	16	32

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>
main()
{
float S, z, c, d;
int i;
S = 0; z = 1; c = 1; d = 2; // начальные значения
for ( i = 1; i <= 20; i ++ )
   {
   S = S + z*c/d;           // добавить элемент к сумме
   z = - z;                 // изменить переменные z, c, d
   c ++;
   d = d * 2;
   }
printf("Сумма S = %f", S);
}
```

Суммы с ограничивающим условием

Рассмотрим более сложную задачу, когда количество элементов заранее неизвестно.

Задача. Найти сумму всех элементов последовательности

$$S = \frac{1}{2} - \frac{2}{4} + \frac{3}{8} - \frac{4}{16} + \cdots,$$

которые по модулю не 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>
main()
{
float S, z, c, d, a;
S = 0; z = 1; c = 1; d = 2; // начальные значения
a = 1;                      // любое число, большее 0.001

while ( a >= 0.001 )
   {
   a = c / d;               // вычислить модуль элемента
   S = S + z*a;             // добавить элемент к сумме
   z = - z;                 // изменить переменные z, c, d
   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».

5. Методы отладки программ

Отладочные средства *Dev-C++*

Что такое отладка?

Слово «отладка» означает «поиск и исправление ошибок в программе». Соответствующий английский термин *debugging* (дословно «удаление жучков») по преданию связан с тем, что в контакты компьютера Mark II в 1940 году попал жучок (моль), из-за которого в вычислительной машине произошел сбой.

Существует три типа ошибок в программах:

- *синтаксические* – ошибки в написании операторов (например, вместо **`printf`** часто неправильно пишут просто **`print`**); эти ошибки легко исправить, потому что их обнаруживает транслятор и чаще всего говорит, в какой именно строчке ошибка;
- *ошибки времени выполнения* – во время работы случилась «аварийная» ситуация, например, деление не ноль;
- *логические* – ошибки в алгоритме (программа работает, но выполняет не то, что нужно); эти ошибки сложнее всего обнаружить, потому что иногда они проявляются совсем не в том месте, где находится неверная команда.

Итак, самое трудное во время отладки – это найти ошибку, то есть определить ошибочный оператор. К сожалению, эту задачу никак не удается автоматизировать, ее может решить только человек. Для облегчения этой работы придумали хитрые приемы и написали специальные программы – **отладчики**.

Трассировка

Трассировка – это вывод сигнальных сообщений в определенных точках программы во время ее работы. Что же это дает?

Во-первых, получив такое сообщение на экран, мы знаем, что программа вышла в эту точку (не завершила работу аварийно раньше и не зациклилась). Во-вторых, в этих сообщениях можно выводить не только текст, но и значения переменных – это позволяет проверять, правильно ли считает программа. Если, например, во 2-ой контрольной точке все значения переменных были верные, а в 3-ой уже нет, то ошибку нужно искать между этими точками.

В этой программе в трех точках (1, 2 и 3) стоят операторы трассировки:

```
main()
{
 int i, X;
 printf("Введите целое число:\n");
 scanf("%d", &X);
 printf("Введено X=%d\n", X);              // точка 1
 for(i=1; i<10; i++)
   {
   printf("В цикле: i=%d, X=%d\n", i, X);  // точка 2
   ...
   }
 printf("После цикла: X=%d\n", X);         // точка 3
 ...
}
```

В точке 1 мы увидим на экране число, которое находится в переменной **X** после ввода значения с клавиатуры – так можно проверить правильность ввода.

В точке 2 мы выводим на экран значения переменных на каждом шаге цикла. Это позволяет сверить их с ручным расчетом и выяснить, верно ли выполняются вычисления в цикле.

В точке 3 значение переменной **X** после завершения цикла выводится на экран.

Заметим, что трассировку можно применять практически всегда, для этого не нужна программа-отладчик.

Отключение части кода

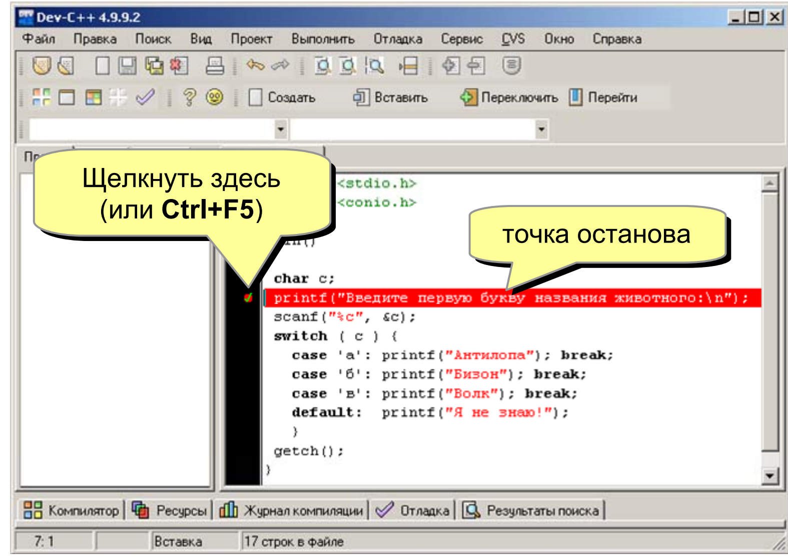
Нередко бывает так: в результате усовершенствований работавшая программа перестает работать. В такой ситуации для поиска ошибки нужно как-то убрать из программы весь новый код и добавлять его «маленькими порциями», чтобы определить, какие именно добавленные операторы портят все дело. Удалять код из программы совсем не хочется, но можно его отключить, сделав комментарием.

Чтобы отключить одну строчку программы, перед ней нужно поставить символы **//**, а для отключения целого блока из нескольких операторов удобнее использовать многострочный комментарий, который начинается символами **/*** и заканчивается символами ***/** (здесь и далее многоточие означает «какие-то команды»):

```
main()
{
 int i, X;
 printf("Введите целое число:\n");
 scanf("%d", &X);
 // X *= X + 2;
 for(i=1; i<10; i++) X *= i;
 /* while ( X > 5 ) {
   i = i * X;
   } */
 ...
}
```

Пошаговое выполнение

Обычно программа выполняется безостановочно от начала до конца. Лучший способ отладки — это выполнить программу по строчкам, останавливаясь после выполнения каждой команды и проверяя значения переменных в памяти. Для этой цели служат специальные программы — **отладчики**.



С оболочкой *Dev-C++* поставляется отладчики *GDB*. Сначала нужно установить *точки останова*, то есть отметить строки, где нужно остановить программу. Для этого достаточно щелкнуть мышкой слева от нужной строчки программы на черном фоне. Повторный щелчок снимает точку останова в этом месте. Кроме того, с помощью комбинации клавиш **Ctrl+F5** можно устанавливать и снимать точку останова в том месте, где стоит курсор.

Когда в программе есть хотя бы одна точка останова, можно запустить ее в отладочном режиме, нажав кнопку **F8.** Отладчик должен остановить программу на первой встретившейся

точке останова. После этого можно выполнять программу в пошаговом режиме (по одной строчке), нажимая клавишу **F7**.

Однако в этом режиме отладчик не позволяет входить внутрь вызываемых процедур (по шагам выполняется только основная программа). Для входа в процедуру или функцию используют комбинацию **Shift+F7**.

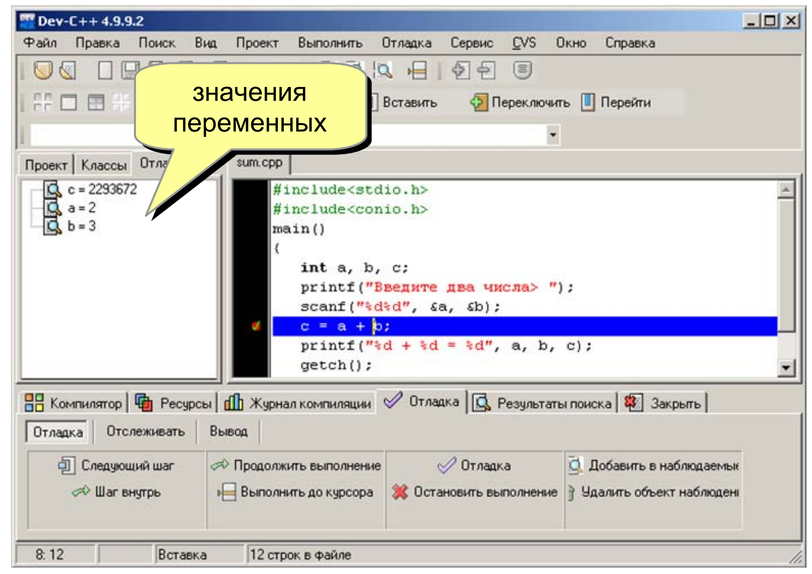
При нажатии клавиш **Ctrl+F7** отладчик запускает программу до следующей точки останова. Завершить отладку можно с помощью комбинации клавиш **Ctrl+Shift+F2**.

Просмотр 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.

Если в режиме отладки навести мышку на имя переменной в тексте программа, ее значение сразу появится в окне слева.



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>
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
5	?	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`**.

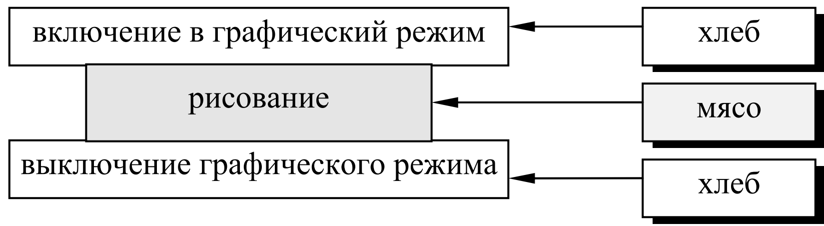
Проверка крайних значений

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 числа **2** (число 1 не считается ни простым, ни составным). 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

Простейшая графическая программа

Графическая программа на Си имеет структуру сэндвича:



Сейчас мы напишем простейшую графическую программу. Она не делает ничего полезного, просто открывает специальное окно для рисования, ждет нажатия клавиши и закрывает это окно. Программа эта так же неполноценна, как сэндвич без мяса.

```
#include <graphics.h>
#include <conio.h>
main()
{
initwindow ( 400, 300 ); // открыть окно для графики 400 на 300
// ... здесь можно рисовать на экране («мясо»)
getch();                 // ждем нажатия клавиши
closegraph();            // закрыть окно
}
```

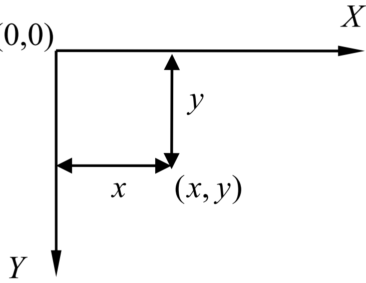
Что новенького?

- Для использования графических функций надо подключить заголовочный файл **graphics.h**.
- Функция **initwindow** открывает дополнительное окно, в котором можно рисовать. В скобках нужно указать ширину и высоту этого окна в пикселях.
- Окно с графикой закрывается с помощью функции **closegraph**.

Как начать рисовать?

Координаты точек

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



- Начало координат, точка (0,0), находится в левом верхнем углу окна.
- Ось **X** направлена вправо, ось **Y** — вниз (в отличие от общепринятой математической системы координат).
- Для любой точки координата **x** – это расстояние до левой границы окна, а **y** – расстояние до верхней границы.

Цвет

Для 16 стандартных цветов заданы числовое и символьное обозначения:

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

Кроме того, можно использовать полную палитру цветов (режим *True Color*, истинный цвет). В этом случае цвет строится из трех составляющих: красной (**R**) , зеленой (**G**) и синей (**B**). Каждая из этих составляющих – целое число от 0 до 255 (256 вариантов), таким образом, всего получается $256^3 = 16\,777\,216$ цветов. Цвета строятся с помощью функции **COLOR**, у нее в скобках перечисляются через запятую значения составляющих **R** , **G** и **B** (именно в таком порядке). Вот, например, некоторые цвета:

COLOR(0,0,0)	черный
COLOR(255, 0, 0)	красный
COLOR(0, 255, 0)	зеленый
COLOR(0, 0, 255)	синий
COLOR(255, 255, 255)	белый
COLOR(100, 100, 100)	серый
COLOR(255, 0, 255)	фиолетовый
COLOR(0, 255, 0)	желтый

Функция **setcolor** устанавливает цвет линий:

```
setcolor ( 10 ); // установить светло-зеленый цвет
```

Все линии (а также прямоугольники, окружности), нарисованные после этой команды, будут светло-зеленого цвета. Эта же функция может установить цвет из полной палитры:

```
setcolor ( COLOR(255,0,255) ); // установить фиолетовый цвет
```

Когда открывается окно для графики, по умолчанию (то есть без дополнительных действий) устанавливается белый цвет.

Работа с отдельными пикселями

Для рисования используются стандартные функции. Для каждого пикселя можно задать свой цвет с помощью функции **putpixel**:

```
putpixel ( x, y, 14 );  // покрасить точку (x,y) в желтый цвет
```

Функция **getpixel** позволяет определить цвет любого пикселя окна:

```
n = getpixel(x, y); // записать цвет точки (x,y) в переменную n
```

Линии

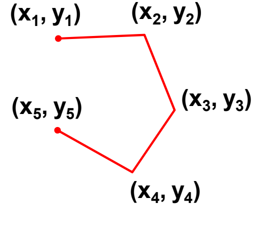
Отрезок можно нарисовать с помощью команды **line**:

```
line ( x1, y1, x2, y2 );  // отрезок (x1,y1)-(x2,y2)
```

Есть и другой способ: сначала перевести курсор (указатель текущего положения) в точку **(x1,y1)** командой **moveto**, а затем нарисовать отрезок в точку **(x2,y2)** командой **lineto**:

```
moveto ( x1, y1 );  // курсор в точку (x1,y1)
lineto ( x2, y2 );  // отрезок в точку (x2,y2)
```

После выполнения команды **lineto** курсор смещается в новую точку **(x2,y2)**. Особенно удобно использовать эти команды при рисовании ломаных линий:




```
setcolor(12);     // красный цвет
moveto (x1, y1); // курсор в первую точку
lineto (x2, y2); // отрезок во вторую точку
lineto (x3, y3); // и т.д.
lineto (x4, y4);
lineto (x5, y5);
```

Прямоугольники

Для рисования прямоугольника нужно задать координаты двух противоположных углов (обычно выбирают левый верхний и правый нижний углы). Цвет контура устанавливается с помощью функции **setcolor**, а сам прямоугольник рисуется командой **rectangle**:

(x_1, y_1) (x_2, y_2)

```
setcolor ( 9 );
rectangle (x1, y1, x2, y2);
```

(x_1, y_1) (x_2, y_2)

Закрашенный прямоугольник рисует команда **bar**. Цвет и стиль заливки нужно заранее установить, вызвав функцию **setfillstyle**:

```
setfillstyle ( 1, 12 ); // стиль 1, цвет 12
bar (x1, y1, x2, y2);
```

Первое число в команде **setfillstyle** задает стиль заливки:

0	– отключить заливку
1	– сплошная заливка
3,4,5,6	– наклонные линии
7,8	– сетка
9,10,11	– точечные узоры

а второе – цвет.

Окружность

Чтобы нарисовать окружность, используют функцию **circle**:

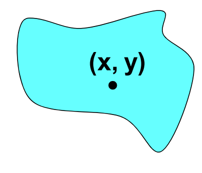
```
setcolor ( COLOR(0,255,0) ); // зеленый цвет
circle ( x, y, R );
```

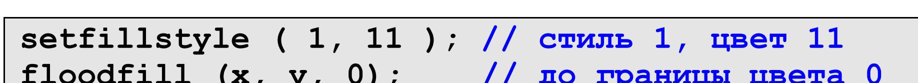
При вызове функции **circle** в скобках указывают координаты центра и радиус окружности в пикселях. Это могут быть числа, имена переменных или арифметические выражения, например:

```
circle ( 200, y0+20, R );
```

Заливка произвольной области

Иногда бывает нужно залить каким-то цветом произвольную область, ограниченную контуром одного цвета. Это можно сделать с помощью функции **floodfill**:





```
setfillstyle ( 1, 11 ); // стиль 1, цвет 11
floodfill (x, y, 0);    // до границы цвета 0
```

Для заливки нужно знать координаты **(x,y)** одной (любой!) точки внутри этой области. Кроме того, нужно, чтобы граница области была одного цвета, без разрывов. Цвет границы указывается последним в списке данных, которые передаются функции **floodfill**.

Надписи

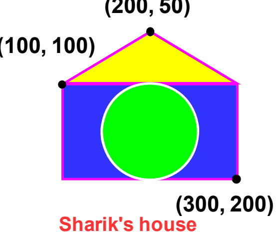
Функция **outtextxy** позволяет выводить текст в любом месте окна. Ей нужно задать координаты **(x,y)** левого верхнего угла текста. Цвет текста устанавливается с помощью функции **setcolor**:

(x, y)
Вася

```
setcolor ( 9 );
outtextxy ( x, y, "Вася" );
```

Пример программы

Напишем программу, которая использует стандартные функции для рисования домика. Попытайтесь разобраться в ней самостоятельно.




```
#include <graphics.h>
#include <conio.h>
main()
{
initwindow (440, 300);
setfillstyle (1, 9);
bar (100,100,300,200);  // синий прямоугольник
setcolor (13);          // с фиолетовой рамкой
rectangle (100,100,300,200);
moveto (100,100);       // крыша-ломаная
lineto (200, 50);
lineto (300,100);
setfillstyle (1, 14);   // зальем крышу желтым
floodfill (200, 75, 13);
setcolor (15);
circle (200, 150,50);   // белая окружность
setfillstyle (1, 10);
floodfill (200,150, 15); // зеленая заливка
setcolor (12);
outtextxy (100, 230, "Sharik's house.");
getch();
closegraph();
}
```

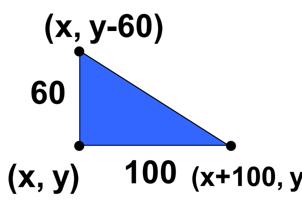
7. Процедуры

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 разного цвета.

Конечно, можно нарисовать отдельно три фигуры «вручную», но хочется как-то облегчить работу, используя их сходство.

Сначала нужно определить, что **общего** имеют эти фигуры (размеры, угол поворота) и в чем их **отличие** (цвет заливки, координаты). Если размеры треугольников и угол поворота известны, то для построения треугольника достаточно знать координаты одной (любой) его точки (удобно выбрать одну из вершин).



Пусть **`(x,y)`** – координаты левого нижнего угла, ширина основания и высота равны соответственно 100 и 60. Тогда легко подсчитать координаты остальных вершин: **`(x,y-60)`** и **`(x+100,y)`**. Здесь мы учли, что ось **`Y`** направлена вниз.

Введем новую команду **`Tr`**, которая будет рисовать треугольник. Вызывать ее будем так:

```
Tr ( x, y, c );
```

Здесь **`c`** – это цвет заливки треугольника.

Вся проблема в том, что компьютер (вернее, программа-транслятор) не знает такой команды и не сможет ее выполнить (выдать ошибку «*Неизвестная функция*»). Значит, нужно *объяснить* эту команду, расшифровать ее через уже известные команды. Расшифровку мы оформим так:

```
void Tr ( int x, int y, int c )
{
   moveto ( x, y );         // курсор в левый нижний угол
   lineto ( x, y-60 );      // рисуем контур
   lineto ( x+100, y );
   lineto ( x, y );
   setfillstyle ( 1, c );   // устанавливаем цвет заливки
   floodfill ( x+20, y-20, 15); // заливка до белой границы
}
```

Новые команды, введенные таким образом, в программировании называются подпрограммами (вспомогательными программами) или **процедурами**.

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

Процедура оформляется почти так же, как и основная программа, только ее имя – не **`main`**, а какое-то другое. Она состоит из заголовка, после которого внутри фигурных скобок записывают **тело процедуры** – те команды, которые выполняются при вызове. Эти команды должны уже быть известны транслятору.

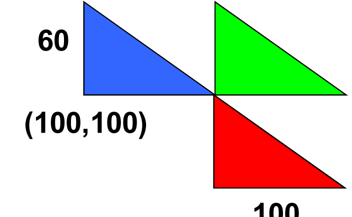
Рассмотрим подробно заголовок процедуры

```
void Tr ( int x, int y, int c )
```

Слово **void** означает, что эта команда выполняет какие-то действия (например, что-то рисует), а не вычисляет какое-то значение[2]. В скобках через запятую перечисляются **параметры** процедуры.

> **Параметры** – это дополнительные данные для работы процедуры.

Можно сделать процедуру без параметров, но она будет всегда выполнять совершенно одинаковые действия. Например, всегда будет рисовать синий треугольник в одном и том же месте. Параметры дают возможность менять результат работы процедуры – перемещать фигуру (изменяя параметры **x** и **y**) и задавать разные цвета заливки (меняя параметр **c**).



Вспомните, что мы включили в параметры все, что изменяется. Эти данные заранее неизвестны, поэтому они обозначаются именами (переменными) и называются **формальными** параметрами. В заголовке процедуры для каждого параметра указывается его тип, как при объявлении переменных.

Как же использовать такую процедуру? Предположим, что левый нижний угол синего треугольника нужно разместить в точке **(100,100)**. Учитывая, что длина основания треугольника и его высота равны соответственно 100 и 60, находим координаты соответствующих углов двух других треугольников: (200,100) для зеленого и (200,160) для красного. Решение нашей задачи выглядит так:

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

void Tr ( int x, int y, int c )
{
   moveto ( x, y );          // курсор в левый нижний угол
   lineto ( x, y-60 );       // рисуем контур
   lineto ( x+100, y );
   lineto ( x, y );
   setfillstyle ( 1, c );  // устанавливаем цвет заливки
   floodfill ( x+20, y-20, 15); // заливка до белой границы
}

main()
{
 initwindow (400, 300);
 Tr (100, 100, COLOR(0,0,255));
 Tr (200, 100, COLOR(0,255,0));
 Tr (200, 160, COLOR(255,0,0));
 getch();
 closegraph();
}
```

Посмотрим на программу внимательно. Процедура (расшифровка, описание новой команды) расположена выше основной программы. Она оформлена в виде отдельного блока, не внутри основной программы, а рядом с ней. Строчка

```
Tr (100, 100, COLOR(0,0,255));
```

это вызов процедуры. В скобках указаны фактические значения всех параметров (**фактические параметры**). При таком вызове первое значение (100) подставляется в процедуру вместо первого параметра – **x**, второе значение – вместо второго параметра и т.д. Важно, что к моменту

[2] Позднее мы изучим подпрограммы, которые возвращают значение-результат – они называются **функциями**.

обработки этой строчки транслятор уже знает процедуру **Tr**, поскольку она была объявлена выше.

Чтобы нарисовать оставшиеся два треугольника, нам не нужно заново два раза писать все команды, вошедшие в процедуру, а достаточно вызвать процедуру еще два раз, с другими параметрами:

```
Tr (200, 100, COLOR(0,255,0));
Tr (200, 160, COLOR(255,0,0));
```

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 предназначена не для вычислений, а для выполнения некоторых действий.
- После имени в скобках через запятую перечисляются **параметры** процедуры — те величины, от которых зависит ее работа. Параметры иногда называют **аргументами** процедуры.
- 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.
- Процедура должна быть объявлена *до* основной программы; чтобы к моменту вызова процедуры транслятор знал, что есть такая процедура, а также сколько она имеет параметров и каких. Это позволяет находить ошибки на этапе трансляции, например такие:

`Tr ( 100 );`	**Too few arguments** (слишком мало параметров).
`Tr (100, 100, 5, 5);`	**Too many arguments** (слишком много…).

- 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.
- В процедуре можно использовать несколько операторов **return**: при выполнении любого из них работа процедуры заканчивается.

8. Функции

Отличие функций от процедур

Теперь мы познакомимся ещё с одним типом подпрограмм – с функциями. 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 (подпрограмма), предназначенная для получения некоторого объекта-результата (например, числа). 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 )   // заголовок функции
{                         // начало функции
int d, sum = 0;
while ( N != 0 )
  {
  d = N % 10;             // тело функции
  sum = sum + d;
  N = N / 10;
  }
return sum;               // функция возвращает значение sum
}                         // конец функции
main()
{
int N, s;
printf ( "\nВведите целое число ");
scanf ( "%d", &N );
s = SumDigits (N);        // вызов функции
printf ( "Сумма цифр числа %d равна %d\n", N, s );
getch();
}
```

Что новенького?

- Функция оформляется так же, как процедура: заголовок и тело функции в фигурных скобках.
- Перед именем функции 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 используются внутри функции вместо формальных параметров.
- Фактические параметры — это числа или любые арифметические выражения (в этом случае сначала рассчитывается их значение).
- Первый фактический параметр используется внутри 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 функции заканчивается.
- В функции можно использовать несколько операторов `return`.
- Если функции расположены 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, главным образом, 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 этой функции во время обработки основной программы, надо объявить её заранее.

```
#include <stdio.h>
#include <conio.h>
int Prime ( int N );   // объявление функции
main()
{
int N;
printf ( "\nВведите целое число ");
scanf ( "%d", &N );
if ( Prime(N) )        // вызов функции
     printf ( "Число %d - простое\n", N );
else printf ( "Число %d - составное\n", N );
getch();
}
int Prime ( int N )   // описание функции
{
for ( int i = 2; i*i <= N; i ++ )
   if ( N % i == 0 ) return 0;  // нашли делитель – составное!
return 1;  // не нашли ни одного делителя – простое!
}
```

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.

параметр-результат

```
#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; }
}
main()
{
int N, M, min, max;
printf ( "\nВведите 2 целых числа ");
scanf ( "%d%d", &N, &M );
```

```
min = MinMax ( N, M, max ); // вызов функции
printf ( "Наименьшее из них %d, наибольшее — %d\n", min, max );
getch();
}
```

Обычно при передаче параметра в процедуру или функцию в памяти создается копия переменной, и функция работает с этой копией. Это значит, что все изменения переменной-параметра, сделанные в функции, 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 функция работает прямо с переменной из вызывающей программы, а не с ее копией. Поэтому в нашем примере функция изменит значение переменой **max** из основной программы и запишет туда максимальное из двух чисел.

Этот приём можно использовать и для процедур: хотя формально они не возвращают никакого значения-результата, можно всё-таки передавать данные в вызывающую программу через изменяемые параметры.

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 подпрограммой, потому что она работает с *копиями* параметров (например, если менять значения **a** и **b** в функции **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 подпрограммы специальным образом: перед их именем ставится знак **&** — в данном случае он означает ссылку, то есть подпрограмма может менять значение параметра (в данном случае функция меняет значение переменной **max** в основной программе).
- При вызове таких функций и процедур вместо каждого фактического изменяемого параметра 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* (постоянных величин):

  ```
  const N = 20;
  ```
- *Глобальные переменные* — это переменные, объявленные вне основной программы и подпрограмм. 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 подпрограммы, она должна быть объявлена и известны типы всех ее параметров.
- *Основная программа* может располагаться как до всех подпрограмм, так и после них. Не рекомендуется вставлять ее между подпрограммами, так как при этом ее сложнее найти.

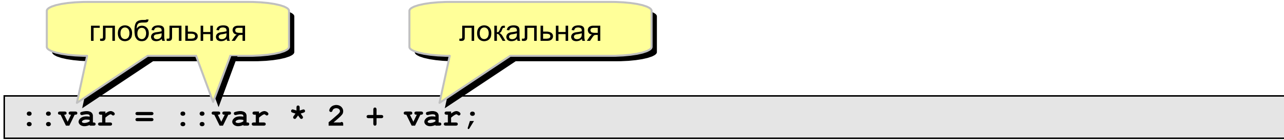
Глобальные и локальные переменные

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

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

Что новенького?

- Глобальные переменные не надо заново объявлять в подпрограммах.
- Если в подпрограмме 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 подпрограмме перед ее именем ставится два двоеточия:



Однако специалисты рекомендуют использовать как можно меньше глобальных переменных, а лучше всего – не использовать их вообще, потому что глобальные переменные

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

Поэтому глобальные переменные применяют в крайних случаях:

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

Везде, где можно, надо передавать данные в процедуры и функции через их параметры. Если же надо, чтобы подпрограмма 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>
main()
{
float  x, y;
printf ("\nВведите 2 числа ");
scanf  ("%d%d", &x, &y);
printf ("Их сумма %d ", x+y);
}
```

```
#include <stdio.h> 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:

- Одинаковые операции в разных частях программы оформляются в виде подпрограмм.
- Имена функций и процедур должны быть информативными, то есть нести информацию о том, что делает эта подпрограмма. 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 подпрограммы надо вставлять несколько строк с комментарием (здесь можно писать по-русски). В комментарий записывается самая важная информация: что означают параметры подпрограммы, что она делает, какое значение она возвращает (если это функция), её особенности.
- Не рекомендуется делать подпрограммы длиной более 25-30 строк, так как при этом они становятся сложными и запутанными. Если подпрограмма получается длинной, ее надо разбить на более мелкие процедуры и функции.
- Чтобы отделить одну часть подпрограммы 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 < b) || (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 ошибок

крупный блок

пустая строка

Отступы

Отступы используются для выделения структурных блоков программы (подпрограмм, 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 2-4 символа.
- Дополнительным отступом выделяются
  - тело циклов **for, while, do-while**
  - тело условного оператора **if** и блока **else**
  - тело оператора множественного выбора **switch**

Вот пример записи программы с отступами (запись «лесенкой»):

```
main()
{
int a = 1, b = 4, i;
for (i=1; i<2; i++)
  {
  if ( a > b )
    {
    b = a + i;
    }
  else
    {
    a = b + i;
    }
  }
printf ( "%d %d\n", a, b );
}
```

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.

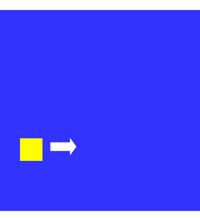
Чтобы сделать программу с эффектами анимации, надо решить две задачи:

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

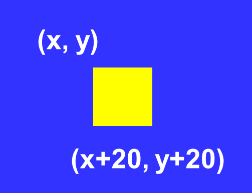
В этом разделе рассматриваются самые простые задачи этого типа. Во всех программах предусмотрен выход по клавише **Esc**.

Движение объекта

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



Предварительный анализ



Рассмотрим объект, который движется по экрану. Пусть это будет квадрат со стороной 20 пикселей. При движении координаты всех его точек будут меняться. Чтобы строить квадрат в любом месте экрана, мы выберем его левый верхний угол в качестве базовой (опорной) точки и обозначим ее координаты за **(x,y)**. Координаты противоположного угла будут равны **(x+20,y+20)**.

Теперь надо придумать способ изобразить движение так, чтобы рисунок не мигал, и программа работала одинаково на всех компьютерах независимо от 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 шагу 1.

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

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

```
void Draw( int x, int y, int color )
{
setfillstyle ( 1, color );  // сплошная заливка, цвет color
bar ( x, y, x+20, y+20 );   // залитый прямоугольник
}
```

Обратите внимание, что эта процедура умеет рисовать и стирать квадрат.

Все действия, которые входят в алгоритм, надо выполнить много раз, поэтому применим 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 **Esc**. Мы будем использовать окно размером 400 на 400 пикселей. При этом координата **x** может меняться от 0 до 399, поэтому нужное нам условие продолжения цикла выглядит так:

```
x + 20 < 400
```

Когда это условие нарушается, квадрат «уехал» за границу окна и нужно закончить выполнение программы.

Обработка событий клавиатуры

Надо также обеспечить выход по клавише **Esc**. 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 0 (ответ «нет»), если никакая клавиша не нажата, и ненулевое значение, если нажали любую клавишу. 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 **Esc** (27), то выходим из цикла

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

Esc	27
Enter	13
пробел	32

Программа

Программа «в сборе» выглядит так:

```
#include <conio.h>
#include <graphics.h>
void Draw ( int x, int y, int color )
  {
  setfillstyle ( 1, color );  // сплошная заливка, цвет color
  bar ( x, y, x+20, y+20 );   // залитый прямоугольник
  }
main()
{
 int x, y;                    // координаты квадрата
 initwindow (400, 400);       // открыть окно для графики
 setfillstyle(1, COLOR(0,0,255)); // сплошная заливка, синий цвет
 bar (0, 0, 399, 399);        // залить фон
 x = 0; y = 240;              // начальные координаты квадрата
   /* анимация */
 closegraph();      // закрыть окно для графики
}
```

Осталось лишь дописать основной блок, который обозначен в программе комментарием

```
    /* анимация */
```

В нем нужно организовать цикл анимации, который заканчивается тогда, когда квадрат коснулся границ окна или нажата клавиша **Esc.**

```
while ( x + 20 < 400 ) // пока не коснулся границы окна
   {
   if ( kbhit() )                   // если нажата клавиша...
     if ( getch() == 27 ) break;     // если Esc, выход из цикла
   Draw ( x, y, COLOR(255,255,0) ); // рисуем желтый квадрат
   delay ( 20 );                    // смотрим на него (задержка)
   Draw ( x, y, COLOR(0,0,255) );   // стираем
   x ++;                            // перемещаем
   }
```

Цикл **while** выполняется до тех пор, пока фигура находится в пределах экрана. Нажатие на клавишу **Esc** 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 клавиши **Esc**, 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 затем стираем фигуру. После этого изменяем координату **x** и возвращаемся к началу цикла.

Что новенького?

- Чтобы определить нажата ли какая-нибудь клавиша, используется функция **kbhit**. Она возвращает 0, если никакая клавиша не нажата, и ненулевое значение, если была нажата какая-то клавиша.
- Если клавиша уже была нажата, ее код можно получить (без дополнительного ожидания) с помощью функции **getch**.
- Чтобы сделать задержку на заданное время, используется процедура **delay**. Параметром этой процедуры является величина задержки в миллисекундах. Если уменьшать задержку, фигура будет двигаться быстрее.

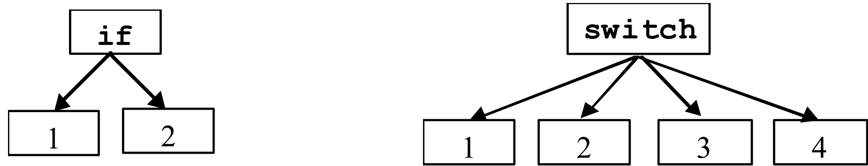
Управление клавишами-стрелками

Предварительный анализ

Принцип работы программы очень простой: получив код клавиши, надо сдвинуть объект в соответствующую сторону. Если обозначить изменения координат фигуры **x** и **y** за 1 шаг цикла через **dx** и **dy**, для движения в четырех направлениях получаем:

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. 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.

Здесь используется бесконечный цикл **while(1)**. Выйти из него можно только одним способом – через оператор **break** (досрочный выход из цикла).

```
while ( 1 )                           // бесконечный цикл
  {
  Draw ( x, y, COLOR(255,255,0) );    // рисуем квадрат
  code = getch();                     // ждем нажатия клавиши
  if ( code == 27 ) break;            // если Esc, то выход
  Draw ( x, y, COLOR(0,0,255) );      // стираем квадрат
  switch ( code ) {                   // выбор направления
    case 75: x --; break;             // влево
    case 77: x ++; break;             // вправо
    case 72: y --; break;             // вверх
    case 80: y ++;                    // вниз
    }
  }
```

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. Обратите внимание, что целую переменную **code** нужно объявить в начале основной программы.

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 **code**, и обрабатываем это нажатие с помощью оператора **switch**.

```
dx = 1;  dy = 0;      // сначала двигается вправо
while ( 1 )           // бесконечный цикл
  {
  if ( kbhit() ) {              // если нажата клавиша
    code = getch();               // получить ее код
    if ( code == 27 ) break;      // если Esc, то выход
    switch ( code ) {             // изменить направление движения
      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;
      }
   }
 Draw ( x, y, COLOR(255,255,0) ); // рисуем квадрат
 delay ( 10 );                    // ждем
 Draw ( x, y, COLOR(0,0,255) );   // стираем
 x += dx;                         // двигаем квадрат
 y += dy;
 }
```

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**.

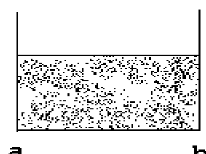
> **Псевдослучайные числа** – это последовательность чисел, обладающая свойствами, близкими к свойствам случайных чисел, в которой каждое следующее число вычисляется на основе предыдущих по некоторой математической формуле.

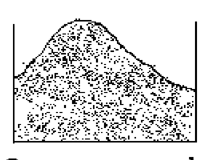
Таким образом, эта последовательность ведет себя так же, как и последовательность случайных чисел, хотя, зная формулу, мы можем получить следующее число в последовательности.

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)
srand (m);	установить начальное значение случайной последовательности, равное **m**

Случайные числа в заданном интервале

Для практических задач надо получать случайные числа в заданном интервале **[a,b]**. Если интервал начинается с нуля (**a=0**), можно использовать свойство операции взятия остатка от деления: остаток от деления числа на некоторое **N** всегда больше или равен нуля, но меньше **N**, то есть находиться в интервале **[0,N-1]**. Можно написать такую функцию

```
int random ( int N )
{
  return rand() % N;  // случайное число в интервале [0,N-1]
}
```

С ее помощью (вызывая ее много раз подряд) можно получать последовательность случайных чисел в интервале **[0,N-1]** с равномерным распределением.

Теперь попытаемся использовать эту функцию для интервала **[a,b]**. Очевидно, что формула

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

дает последовательность в интервале **[a,a+N-1]**. Поскольку нам нужно получить интервал **[a,b]**, сразу 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. Если разделить результат функции **rand()** на **RAND_MAX**:

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

мы получим 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 **[0,1)** такой последовательности равна **1**, а нам надо получить интервал длиной **b-a**. 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 **[0,399]**, случайное значение **y** в интервале **[0,299]** и проверяет цвет точки с координатами **(x,y).** Если эта точка черная, то ее цвет устанавливается случайный, а если нет — черный. Случайный цвет строится с помощью стандартной функции **COLOR** из красной (**R**), зеленой (**G**) и синей (**B**) составляющих, которые выбираются случайно в интервале [0,255].

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
int random (int N) { return rand() % N; } // функция
main()
{
int  x, y, R, G, B;
initwindow (500, 500);
while ( ! kbhit () ) {           // пока не нажата клавиша
  x = random(400);               // случайные координаты
  y = random(300);
  R = random(256);               // случайный цвет (R,G,B)
  G = random(256);
  B = random(256);
  if ( getpixel(x,y) != 0 )      // если точка не черного цвета
    putpixel ( x, y, 0 );        // делаем ее черной
  else                           // иначе...
    putpixel ( x, y, COLOR(R,G,B) );  // случайный цвет
  }
getch();
closegraph();
}
```

Что новенького?

- для определения того, была ли нажата какая-нибудь клавиша, используется функция **kbhit,** которая возвращает 0, если клавиша не была нажата, и ненулевое значение, если нажата любая 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.**