

## Глава 8. Алгоритмизация и программирование. Язык Си

### § 54. Алгоритм и его свойства

#### Что такое алгоритм?

Происхождение слова «алгоритм» связывают с именем учёного аль-Хорезми (перс. خوارزمی [al-Khwārazmī]), который описал десятичную систему счисления (придуманную в Индии) и предложил правила выполнения арифметических действий с десятичными числами.

**Алгоритм** — это точное описание порядка действий, которые должен выполнить исполнитель для решения задачи.

Здесь **исполнитель** — это устройство или одушевленное существо (человек), способное понять и выполнить команды, составляющие алгоритм.

Человек как исполнитель часто действует неформально, по-своему понимая команды. Несмотря на это, ему тоже часто приходится действовать по тому или иному алгоритму. Например, рецепт приготовления какого-либо блюда можно считать алгоритмом. На уроках русского языка, выполняя разбор слова или предложения, вы тоже действуете по определенному алгоритму. Много различных алгоритмов в математике (постарайтесь вспомнить известные вам). На производстве, рабочий, вытачивая деталь в соответствии с чертежом, действует по алгоритму, который разработал технолог. И таких примеров может быть множество.

В информатике рассматривают только **формальных исполнителей**, которые не понимают (и не могут понять) смысл команд. К этому типу относятся все технические устройства, в том числе и компьютер.

Каждый формальный исполнитель обладает собственной **системой команд**. В алгоритмах для такого исполнителя нельзя использовать команды, которых нет в его системе команд.

#### Свойства алгоритма

- **Дискретность** — алгоритм состоит из отдельных команд (шагов), каждая из которых выполняется ограниченное время.
- **Детерминированность (определенность)** — при каждом запуске алгоритма с одними и теми же исходными данными должен быть получен один и тот же результат.
- **Понятность** — алгоритм содержит только команды, входящие в систему команд исполнителя, для которого он предназначен.
- **Конечность (результативность)** — для корректного набора данных алгоритм должен завершаться через конечное время с вполне определенным результатом (результатом может быть сообщение о том, что задача не имеет решений).
- **Корректность** — для допустимых исходных данных алгоритм должен приводить к правильному результату.
- **Массовость** — алгоритм, как правило, предназначен для решения множества однотипных задач с различными исходными данными.

Эти свойства не равноправны. Дискретность, детерминированность и понятность — фундаментальные свойства алгоритма, то есть ими обладают все алгоритмы для формальных исполнителей. Остальные свойства можно рассматривать как требования к «правильному» алгоритму.

Иными словами, алгоритм получает на вход некоторый дискретный входной объект (например, набор чисел или слово) и обрабатывает входной объект по шагам (дискретно), строя промежуточные дискретные объекты. Этот процесс может закончиться или не закончиться. Если процесс выполнения алгоритма заканчивается, то объект, полученный на последнем шаге работы, является результатом работы алгоритма при данном входе. Если процесс выполнения не заканчивается, говорят, что алгоритм зациклился. В этом случае результат его работы не определен.

## Способы записи алгоритмов

Алгоритмы можно записывать разными способами:

- на **естественном языке**, обычно такой способ применяют, записывая основные идеи алгоритма на начальном этапе;
- на **псевдокоде**, так называется смешанная запись, в которой используется естественный язык и операторы какого-либо языка программирования; в сравнении с предыдущим вариантом такая запись гораздо более строгая;
- в виде **блок-схемы** (графическая запись);
- в виде **программы** на каком-либо языке программирования.

Из всех перечисленных здесь способов мы будем использовать два: псевдокод и запись алгоритма в виде программы на двух языках программирования, С и С++.



### Контрольные вопросы

1. Что такое алгоритм?
2. Что такое исполнитель?
3. Чем отличаются формальные и неформальные исполнители?
4. Что такое «система команд исполнителя»? Придумайте исполнителя с некоторой системой команд.
5. Перечислите и объясните свойства алгоритма.
6. Какие существуют способы записи алгоритмов? Какие из них, по вашему мнению, чаще применяются на практике? Почему?

## § 55. Простейшие программы

### Пустая программа

Пустая программа – это программа, которая ничего не делает, но удовлетворяет требованиям выбранного языка программирования. Она полезна, прежде всего, для того, чтобы понять общую структуру программы.

```
main()
{
    // это основная программа
    /* здесь записывают
       операторы */
}
```

Язык С++ основан на С, поэтому в данном случае программы на С и С++ выглядят одинаково. Если же они отличаются, мы будем слева приводить программу на языке С, а справа – эквивалентную программу на языке С++.

Основная программа всегда называется именем **main** (от англ. *main* – основной, главный). Пустые скобки означают, что она не принимает никаких дополнительных данных (параметров) от пользователя.

Тело (основная часть) программы ограничено фигурными скобками. Между ними записывают *операторы* – команды языка программирования. Всё, что следует после символов `//` до конца строки, это *комментарии* – пояснения, которые не обрабатываются транслятором. Комментарий можно ограничивать парами символов `/*` и `*/`, в этом случае комментарий может занимать несколько строк.

### Вывод текста на экран

Напишем программу, которая выводит на экран такие строки:

**2+2=?**

**Ответ: 4**

Вот как она выглядит:

```
#include <stdio.h>
main()
```

```
#include <iostream>
using namespace std;
```

```
{
    printf("2+");
    printf("2=?\n");
    printf("Ответ: 4");
    getchar();
}
```

```
main()
{
    cout << "2+";
    cout << "2=?\n";
    cout << "Ответ: 4";
    cin.get();
}
```

Перед заголовком основной программы добавились строки, начинающиеся с символов **#include** (от англ. *include* – включить). Это команды (инструкции, директивы) для *препроцессора* – программы, которая обрабатывает текст нашей программы до того, как за него «примется» транслятор, генерирующий машинные коды. С помощью команд **#include** в программу включаются *заголовочные файлы*, в которых описаны стандартные функции языка программирования и дополнительных библиотек.

В программе на языке С мы используем две функции: **printf** (вывод на экран) и **getchar** (ожидание нажатия клавиши и ввод символа). Обе функции описаны в файле **stdio.h**, который поставляется со средой программирования.

Как понятно из программы, функция **printf** выводит на экран текст, заключенный в кавычки. Сочетание «\n» внутри строки на самом деле обозначает один специальный управляющий символ – переход на новую строку.

Задача функции **getchar** (в этой программе!) – организовать задержку в конце работы программы до нажатия любой клавиши<sup>1</sup>, чтобы мы смогли увидеть результаты и окно ввода-вывода (так называемое консольное окно) не закрылось бы автоматически сразу после окончания работы программы.

В программе на С++ ввод и вывод организован по-другому. Подключается заголовочный файл **iostream** (от англ. *input output stream* – *поток ввода и вывода*). *Поток вывода* (то есть, последовательность символов, выводимая на экран) называется **cout** (от англ. *console output* – вывод на консоль), а поток ввода (последовательность символов, вводимая с клавиатуры) имеет имя **cin** (от англ. *console input* – ввод с консоли). Справа от оператора << записывают данные, которые нужно вывести на экран, в нашей программе – это строки в кавычках. Можно записывать несколько операторов << в одной команде, например:

```
cout << "2+" << "2=?" << "\n" << "Ответ: 4";
```

Вместо символов \n для перехода на новую строку можно использовать специальную инструкцию (*манипулятор*) **endl**:

```
cout << "2+" << "2=?" << endl << "Ответ: 4";
```

Обратите внимание, что любая команда заканчивается точкой с запятой.

Последняя инструкция **cin.get()** в программе на С++ обеспечивает задержку до нажатия на клавишу *Enter*. Дословно она означает «получить символы из входного потока и ничего с ними не делать».

Строчка

```
using namespace std;
```

говорит о том, что будет использоваться пространство имен **std**, в котором определена вся стандартная библиотека языка С++. Здесь пространство имен – это область действия объектов программы, в том числе переменных. В области **std** описаны стандартные потоки ввода и вывода с именами **cin** и **cout**. Если не объявлять используемое пространство имен, нужно было бы явно указывать его при любом обращении к этим потокам:

```
#include <iostream>
main()
{
    std::cout << "2+";
    std::cout << "2=?\n";
    std::cout << "Ответ: 4";
    std::cin.get();
}
```

<sup>1</sup> Эта функция также определяет код нажатой клавиши.

## Переменные

Напишем программу, которая выполняет сложение двух чисел:

- 1) запрашивает у пользователя два целых числа;
- 2) складывает их;
- 3) выводит результат сложения.

Программу на псевдокоде (смеси русского языка и языка C) можно записать так:

```
main()
{
    // ввести два числа
    // сложить их
    // вывести результат
}
```

Компьютер не может выполнить псевдокод, потому что команд «ввести два числа» и ей подобных, которые записаны в комментариях, нет в его системе команд. Поэтому нам нужно «расшифровать» все такие команды через операторы языка программирования.

В отличие от предыдущих задач, данные нужно хранить в памяти. Для этого используют *переменные*.

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

Переменная (как и любая ячейка памяти) может хранить только одно значение. При записи в неё нового значения «старое» стирается и его уже никак не восстановить.

Переменные в программе необходимо объявлять. При объявлении указывается тип переменной и её имя (*идентификатор*). Значение переменной сразу после объявления не определено: переменной выделяется некоторая область памяти, и там могло быть до этого записано любое число.

Вот так объявляются целочисленные переменные (в которых могут храниться только целые значения) с именами **a**, **b** и **c**:

```
int a, b, c;
```

Описание переменных начинается с ключевого слова, которое определяет тип переменных, а после него записывают список имён переменных. В данном случае указан тип **int** (от англ. *integer* – целочисленный).

В именах переменных можно использовать латинские буквы (строчные и заглавные буквы различаются), цифры (но имя не может начинаться с цифры, иначе транслятору будет сложно различить, где начинается имя, а где – число) и знак подчеркивания «\_».

Желательно давать переменным «говорящие» имена, чтобы можно было сразу понять, какую роль выполняет та или иная переменная.

Тип переменной нужен для того, чтобы

- определить область допустимых значений переменной;
- определить допустимые операции с переменной;
- определить, какой объем памяти нужно выделить переменной и в каком формате будут храниться данные (вспомните, что целые и вещественные числа хранятся по-разному, см. главу 4);
- предотвратить случайные ошибки (опечатки в именах переменных).

После объявления переменной её значение не определено (в ней записан «мусор» – неизвестное значение) и использовать её нельзя. Если нужно, прямо при объявлении можно задать начальные значения переменных:

```
int a = 1, b, c = 123;
```

В этом примере переменной **b** не присваивается начальное значение, в ней содержится «мусор».

Приведем полную программу сложения двух чисел:

```
#include <stdio.h>
main()
{
    int a, b, c;
    scanf("%d%d", &a, &b);
```

```
#include <iostream>
using namespace std;
main()
{
    int a, b, c;
```

```
c = a + b;
printf("%d", c);
getchar();
}
```

```
cin >> a >> b;
c = a + b;
cout << c;
cin.get(); cin.get();
}
```

В языке С ввод данных выполняет функция **scanf** (от англ. *scan formatted* – сканирование по формату). В скобках на первом месте записывают *формат ввода* – символьную строку, в которой формат «%d» означает ввести целое число (для других типов данных используют другие форматы, о которых речь пойдет далее). Далее через запятую записывают *адреса* переменных, в которые нужно записать введенные значения. Для определения адреса переменной используется операция «&». Например, «&a» означает «адрес переменной **a**». Записанная команда ввода означает «ввести два целых числа, первое записать в переменную **a**, второе – в переменную **b**».

В языке С++ ввод данных из входного потока **cin** записывается значительно проще:

```
cin >> a >> b;
```

Вы, наверное, заметили, что команда **cin.get()** повторяется два раза. Это сделано потому, что при первом вызове она прочитает все, что осталось во входном потоке после записи значения переменной **b** (хотя бы символ «новая строка»), и при однократном вызове программа не будет ждать нажатия на клавишу *Enter*.

Оператор, содержащий символ «=» – это **оператор присваивания**, с его помощью изменяют значение переменной. Он выполняется следующим образом: вычисляется выражение справа от символа «=», а затем результат записывается в переменную, записанную слева. Поэтому, например, оператор

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

увеличивает значение переменной **i** на 1. Часто используется также сокращенная запись

```
i ++;
```

которая обозначает то же самое. Аналогичная запись для уменьшения значения переменной на 1 выглядит так:

```
i --;
```

У приведенной выше программы сложения чисел есть два недостатка:

- 1) перед вводом данных пользователь не знает, что от него требуется (сколько чисел нужно вводить и каких);
- 2) результат выдается в виде числа, которое означает неизвестно что.

Хотелось бы, чтобы диалог программы с пользователем выглядел так:

```
Введите два целых числа: 2 3
2+3=5
```

Подсказку для ввода вы можете сделать самостоятельно. При выводе результата ситуация несколько усложняется, потому что нужно вывести значения трёх переменных и два символа: «+» и «=». Для этого строится список вывода:

```
printf("%d+%d=%d", a, b, c);      cout << a << "+" << b << "=" << c;
```

Обратите внимание, что имена переменных записаны без кавычек.

В программе на С форматная строка содержит как символы для вывода на экран (+, =), так и форматы для вывода данных. Первое значение из списка (значение переменной **a**) будет выведено там, где в форматной строке записан первый формат %d, то есть, перед знаком «+», второе значение – на месте второго формата %d и т.д. Элементы в списке вывода разделены запятыми.

В принципе, можно было бы обойтись и без переменной **c**, потому что элементом списка вывода может быть арифметическое выражение, которое сразу вычисляется и на экран выводится результат:

```
printf("%d+%d=%d", a, b, a+b);    cout << a << "+" << b << "=" << a+b;
```

В обоих языках можно использовать *форматный вывод*: указать общее количество знакомест, отводимое на число. В языке С его записывают в коде формата между знаками «%» и «d», а в языке С++ – при вызове специальной команды-манипулятора **setw**, которая определена в заголовочном файле **iomanip**. Например, в результате выполнения команд

```
a = 123;
printf("%5d", a);      | #include <iomanip>
                        | ...
```

```
a = 123;
cout << setw(5) << a;
```

значение целой переменной **a** займет на экране ровно 5 знакомест:

◦◦123

Поскольку само число занимает только 3 знакоместа, перед ним выводятся два пробела, которые обозначены как «◦».

В пределах заданной ширины поля можно выравнивать число не только вправо (как по умолчанию), но и влево:

```
a = 123;
printf("%-5d", a);
```

```
#include <iomanip>
...
a = 123;
cout << setw(5) << left << a;
```

В обоих случаях получаем такой результат:

123◦◦



## Контрольные вопросы

1. Как вы думаете, почему основная программа всегда называется **main**?
2. Что такое идентификатор?
3. Как записываются комментарии? Подумайте, как комментирование можно использовать при поиске ошибок в алгоритме?
4. Сравните операторы вывода в С и С++. Как выполняется переход на новую строку?
5. Что такое переменная? Как вы думаете, зачем нужно объявлять переменные?
6. Каковы правила построения имён переменных?
7. Зачем нужен тип переменной?
8. Какое значение записано в переменной сразу после объявления? Можно ли его использовать?
9. Как задать начальные значения переменных?
10. Что такое оператор присваивания?
11. Почему желательно выводить на экран подсказку перед вводом данных?
12. Подумайте, когда можно вычислять результат прямо в операторе вывода, а когда нужно заводить отдельную переменную.
13. Что такое форматный вывод? Как вы думаете, где он может быть полезен?



## Задачи и задания

1. Используя оператор вывода, постройте на экране следующие рисунки из символов:

```

      Ж      Ж      Ж      Ж      Ж      Ж      Ж
    ЖЖЖ      ЖЖ      Ж      Ж      ЖЖ      ЖЖ      ЖЖ
  ЖЖЖЖЖ      ЖЖЖЖЖ      ЖЖЖЖЖ      ЖЖЖ      ЖЖЖЖЖ
    Ж Ж      ЖЖ      Ж Ж Ж      ЖЖЖЖЖ      ЖЖ      ЖЖ
    ЖЖЖ      Ж      ЖЖЖЖЖ      ЖЖЖЖЖЖЖ      Ж      Ж

```

2. Выберите правильные имена переменных:

1	Vasya	СУ-27	@mail_ru
m11	Петя	СУ_27	lenta.ru
1m	Митин брат	_27	"Pes barbos"
m 1	Quo vadis	СУ(27)	<Ладья>

3. Пусть **a** и **b** – целые переменные. Что будет выведено в результате работы фрагмента программы:

а) 

```
int a = 5, b = 3;
printf("%d=Z(%d)", a, b);
```

```
int a = 5, b = 3;
cout << a << "=Z(" << b << ")";
```

б) 

```
int a = 5, b = 3;
printf("Z(a)=(b)");
```

```
int a = 5, b = 3;
cout << "Z(a)=(b)";
```

в) 

```
int a = 5, b = 3;
printf("Z(%d)=(%d)", a, a+b);
```

```
int a = 5, b = 3;
cout << "Z(" << a << ")=(" << a+b << ")";
```



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

- а) `3+5=?`
- б) `z(5)=f(3)`
- в) `a=5; b=3;`
- г) `Ответ: (5;3)`

## § 56. Вычисления

В курсе программирования можно выделить две важнейшие составляющие – алгоритмы и способы организации данных<sup>2</sup>. В этом параграфе мы познакомимся с простейшими типами данных в языках C и C++. В следующих разделах рассматриваются основные алгоритмические конструкции: ветвления, циклы, подпрограммы. В конце главы подробно изучаются сложные (составные) типы данных: массивы, символьные строки, работа с файлами.

### Типы данных

Любая переменная относится к какому-либо типу, то есть может хранить данные только того типа, который был указан при её объявлении.

В языках C и C++ используются следующие основные типы данных:

- **int** – целые значения;
- **float** – вещественные значения;
- **bool** – логические значения;
- **char** – символ (в памяти хранится код символа);

На переменные типа **char** и **bool** в памяти выделяется 1 байт, на переменную типа **int** – 2 или 4 байта (в зависимости от версии языка), на переменную типа **float** – 4 байта.

Существуют также расширенные типы данных для работы с большими числами и повышения точности вычислений:

- **long int** – длинное целое число (4 или 8 байт);
- **double** – вещественное число двойной точности<sup>3</sup> (8 байт).

Как мы обсуждали в главе 4, большинство вещественных чисел хранится в памяти неточно, и в результате операций с ними накапливается вычислительная ошибка. Поэтому для работы с целочисленными данными не нужно использовать вещественные переменные.

Логические переменные относятся к типу **bool** и принимают значения **true** (истина) и **false** (ложь). Несмотря на то, что теоретически для хранения логического значения достаточно одного бита памяти, такая переменная занимает в памяти один байт. Так как процессор может читать и записывать в память только целые байты, операции с логическими переменными в этом случае выполняются быстрее.

### Арифметические выражения и операции

Арифметические выражения в любом языке программирования записываются в строчку. Они могут содержать константы (постоянные значения), имена переменных, знаки арифметических операций, круглые скобки (для изменения порядка действий) и вызовы функций. Например,

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

При определении порядка действий используется *приоритет* (старшинство) операций. Они выполняются в следующем порядке:

- действия в скобках;
- умножение и деление, слева направо;
- сложение и вычитание, слева направо.

<sup>2</sup> Знаменитая книга швейцарского специалиста Никлауса Вирта, разработчика языков Паскаль, Модула-2 и Оберон, так и называлась «Алгоритмы + структуры данных = программы».

<sup>3</sup> Для того, чтобы не задумываться о потере точности вычисления, многие авторы и разработчики рекомендуют для работы с вещественными числами всегда использовать тип **double**.

Таким образом, умножение и деление имеют одинаковый приоритет, более высокий, чем сложение и вычитание. Поэтому в приведенном примере значение выражения, заключенного в скобки, сначала разделится на 2, а потом результат деления умножится на **d**.

В языках С и С++ часто используют сокращенную запись арифметических операций:

сокращенная запись

```
a += b;
a -= b;
a *= b;
a /= b;
```

полная запись

```
a = a + b;
a = a - b;
a = a * b;
a = a / b;
```

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

В языках С и С++ результат деления целого числа на целое – это всегда **целое** число, остаток при делении отбрасывается. Когда нужно получить вещественный результат деления, одно из чисел (делимое или делитель) нужно преобразовать к вещественному типу: для числа поставить десятичную точку, а для переменной или выражения использовать явное приведение типа:

```
int a = 3, b = 4;
float x;
x = 3 / 4; // = 0
x = 3. / 4; // = 0.75
x = 3 / 4.; // = 0.75
x = a / 4; // = 0
x = a / 4.; // = 0.75
x = a / b; // = 0
x = float(a) / 4; // = 0.75
x = a / float(b); // = 0.75
```

Если нужно получить остаток от деления, применяют операцию «%», которая имеет такой же приоритет, как умножение и деление:

```
d = 85;
a = d / 10; // = 8
b = d % 10; // = 5
```

Нужно учитывать, что для отрицательных чисел эти операции выполняются, строго говоря, не совсем корректно<sup>4</sup>. Дело в том, что с точки зрения теории чисел остаток – это неотрицательное число, поэтому  $-7 = (-4) \cdot 2 + 1$ , то есть частное от деления  $(-7)$  на 2 равно  $-4$ , а остаток – 1. В то же время во многих языках (в том числе, в Паскале и в С) при целочисленном делении используется модуль числа, а затем к частному и остатку добавляется знак «минус»:

$$7 = 3 \cdot 2 + 1 \Rightarrow -7 = (-3) \cdot 2 - 1.$$

При таком подходе частное от деления  $(-7)$  на 2 равно  $-3$ , а остаток  $-(-1)$ .

Операции возведения в степень в С и С++ нет. Для вещественных чисел можно использовать функцию **pow(x, y)**, которая возводит значение **x** в степень **y**. В языке С описание этой функции находится в файле **math.h**, его нужно подключить в начале программы с помощью команды **#include**. В С++ подключается аналогичный файл **cmath**.

## Вещественные значения

При записи вещественных чисел в программе целую и дробную часть разделяют не запятой (как принято в отечественной математической литературе), а точкой. Например

```
float x;
x = 123.456;
```

Вещественное значение можно записывать в целочисленную переменную, при этом дробная часть значения отсекается.

При выводе на экран вещественных значений в языке С можно использовать разные форматы:

**%f** – с фиксированной точкой, по умолчанию выводятся 6 знаков дробной части;

<sup>4</sup> Корректное вычисление остатка реализовано в школьном алгоритмическом языке системы КуМир.



**%e** – научный (или *экспоненциальный*) формат, предназначенный для записи как очень больших, так и очень маленьких чисел;

**%g** – формат вывода выбирается автоматически в зависимости от значения числа.

Например, программа

```
float x;
x = 123.456;
printf("%f\n", x);
printf("%e\n", x);
printf("%g\n", x);
```

выведет на экран результат в виде

```
123.456001
1.234560e+002
123.456
```

Обратите внимание, что в первой строке выведено не то, число, которое мы записывали в переменную, оно отличается от верного значения на  $10^{-6}$ . Дело в том, что большинство вещественных чисел, в том числе и число 123,456, не могут быть точно представлены с помощью конечного числа двоичных разрядов и поэтому записываются в память с ошибкой (см. главу 4). Для данных типа **float** верными можно считать 7 десятичных разрядов.

Результат вывода во второй строке означает  $1,234560 \cdot 10^2 = 123,456$ , то есть до буквы **E** указывают значащую часть числа, а после нее – порядок (см. главу 4).

Во всех форматах для вещественных чисел можно указывать общее количество знакомест и количество знаков в дробной части, например, программа

```
float x;
x = 123.456;
printf("%10.2f\n", x);
printf("%10.2e\n", x);
printf("%10.2g\n", x);
```

выведет на экран

```
○○○○123.46
○○1.23e+002
○○1.2e+002
```

Обратите внимание, что для формата **%10.2g** второе число обозначает количество значащих цифр (в результате число пришлось округлить и использовать научный формат вывода).

В языке C++ можно задать два значения: общее количество знакомест для вывода (с помощью функции **cout.width**) и число значащих десятичных цифр (с помощью функции **cout.precision**). Программа

```
float x;
x = 123.456;
cout.width(10);
cout.precision(5);
cout << x << endl;
cout.width(10);
cout.precision(2);
cout << x << endl;
cout.width(10);
cout.precision(3);
cout << x;
```

выведет на экран

```
○○○○123.46
○○1.2E+002
○○○○○○123
```

Если сравнивать эти результаты с выводом программы на языке C, можно заметить, что по умолчанию при выводе фактически используется формат **%g**.

## Стандартные функции

В стандартную математическую библиотеку, которая подключается с помощью команды

```
#include <math.h>
```

```
#include <cmath>
```

включены математические функции:

- **abs(a)** – модуль целого числа  $a$ ;
- **fabs(x)** – модуль вещественного числа  $x$ ;
- **sqrt(x)** – квадратный корень числа  $x$ ;
- **sin(x)** – синус угла  $x$ , заданного в радианах;
- **cos(x)** – косинус угла  $x$ , заданного в радианах;
- **exp(x)** –  $e^x$ , экспонента числа  $x$ ;
- **ln(x)** – натуральный логарифм числа  $x$ ;
- **pow(x,y)** – возведение числа  $x$  в степень  $y$ ;

Для перехода от вещественных значений к целым используется явное преобразование, при котором отсекается дробная часть:

```
a = (int)-1.6; // = -1
a = (int)1.6;  // = 1
```

Кроме того, можно использовать стандартные функции:

- **floor(x)** – округление числа  $x$  до ближайшего целого, не большего, чем  $x$ ; возвращает вещественное число;
- **ceil(x)** – округление числа  $x$  до ближайшего целого, не меньшего, чем  $x$ ; возвращает вещественное число.

Вот примеры использования этих функций:

```
float x;
x = floor(-1.6); // = -2
x = ceil(-1.6);  // = -1
x = floor(1.6);  // = 1
x = ceil(1.6);   // = 2
```

## Случайные числа

В некоторых задачах необходимо моделировать случайные явления, например, результат бросания игрального кубика (на нём может выпасть число от 1 до 6). Как сделать это на компьютере, который по определению «неслучаен», то есть строго выполняет заданную ему программу?

Случайные числа – это последовательность чисел, в которой невозможно предсказать следующее число, даже зная все предыдущие. Чтобы получить истинно случайные числа, можно, например, бросать игральный кубик или измерять какой-то естественный шумовой сигнал (например, радиосум или электромагнитный сигнал, принятый из космоса). На основе этих данных составлялись и публиковались таблицы случайных чисел, которые использовали в разных областях науки.

Вернёмся к компьютерам. Ставить сложную аппаратуру для измерения естественных шумов или космического излучения на каждый компьютер очень дорого, и повторить эксперимент будет невозможно – завтра все значения будут уже другие. Существующие таблицы слишком малы, когда, скажем, нужно получать 100 случайных чисел каждую секунду. Для хранения больших таблиц требуется много памяти.

Чтобы выйти из положения, математики придумали алгоритмы получения *псевдослучайных* («как бы случайных») чисел. Для «постороннего» наблюдателя псевдослучайные числа практически неотличимы от случайных, но они вычисляются по некоторой математической формуле<sup>5</sup>: зная первое число («зерно») можно по формуле вычислить второе, затем третье и т.п.

<sup>5</sup> В стандартные библиотеки языков программирования обычно входит *линейный конгруэнтный* генератор псевдослучайных целых чисел, использующий формулу

$$X_{k+1} = (aX_k + c) \bmod m,$$

В языке C существует функция **rand** для получения случайных (точнее, псевдослучайных) целых чисел в диапазоне **[0, RAND\_MAX]**, где **RAND\_MAX** – постоянная, определённая в заголовочном файле **stdlib.h** (для C++ вместо него используется файл **cstdlib**). Целое число в заданном диапазоне **[a, b]** можно получить с помощью операции взятия остатка и простой арифметики:

```
n = a + rand()%(b-a+1);
```

Например, для того, чтобы записать в целую переменную **n** случайное число в диапазоне от 1 до 6 (результат бросания кубика), можно использовать оператор

```
n = 1 + rand() % 6;
```

Вещественное случайное число на отрезке **[x, y]** получается так:

```
z = x + (y-x)*rand()/RAND_MAX;
```

Важно, чтобы при использовании этой формулы хотя бы одна из граничных переменных, **x** или **y**, была вещественной (подумайте, почему).



### Контрольные вопросы

1. Какие типы данных вы знаете?
2. Как вы думаете, почему во многих языках программирования есть несколько целочисленных и вещественных типов данных?
3. Какие данные записываются в логические переменные? Почему они обычно занимают целое число байтов?
4. Что такое приоритет операций? Зачем он нужен?
5. В каком порядке выполняются операции, если они имеют одинаковый приоритет?
6. Зачем используются скобки?
7. Что происходит, если в выражения входят переменные разных типов? Какого типа будет результат?
8. Опишите операции деления и взятия остатка. В чем их особенность в языках C и C++? Подумайте, почему в математике они не определены для вещественных чисел.
9. Расскажите о проблеме вычисления остатка от деления в различных языках программирования. Обсудите в классе этот вопрос.
10. Какие стандартные математические функции вы знаете? В каких единицах задается аргумент тригонометрических функций?
11. Как выполнить округление (к ближайшему целому) в языках C и C++?
12. Какие числа называют случайными? Зачем они нужны?
13. Как получить «естественное» случайное число? Почему такие числа почти не используются в цифровой технике?
14. Чем отличаются псевдослучайные числа от случайных?
15. Какие функции для получения псевдослучайных чисел вы знаете?



### Задачи и задания

1. Найдите в справочной системе или в Интернете диапазон значений для вещественных типов данных.
2. Напишите программу, которая находит сумму, произведение и среднее арифметическое трёх целых чисел, введённых с клавиатуры. Например, при вводе чисел 4, 5 и 7 мы должны получить ответ

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

где  $X_{k+1}$  и  $X_k$  – следующее и предыдущее псевдослучайные числа; запись  $n \bmod k$  означает остаток от деления  $n$  на  $k$ ;  $a$ ,  $c$  и  $m$  – целые числа, подобранные так, чтобы в получаемой цепочке чисел было как можно меньше закономерностей. Например, в библиотеке *Microsoft C* использовались значения  $a = 214013$ ,  $c = 2531011$  и  $m = 2^{31}$ .

3. Напишите программу, которая вводит радиус круга и вычисляет его площадь и длину окружности. Можно использовать константу `M_PI`, равную числу  $\pi$  (эта константа определена в математической библиотеке).
4. Напишите программу, которая меняет местами значения двух переменных в памяти.
5. \*В предыдущей задаче попробуйте найти решение, которое не использует дополнительные переменные.
6. Напишите программу, которая возводит введённое число в степень 10, используя только четыре операции умножения. Что произойдет, если ввести большое число, например, 78? Попробуйте объяснить полученный результат.
7. Вычислите значение вещественной переменной `c` при `a = 2` и `b = 3`:
  - а) `c = a + 1 / 3;`
  - б) `c = a + 4 / 2 * 3 + 6;`
  - в) `c = (a + 4) / 2 * 3;`
  - г) `c = a + 4) / (b + 3) * a;`
8. Вычислите значение целочисленной переменной `c` при `a = 26` и `b = 6`:
  - а) `c = a % b + b;`
  - б) `c := a / b + a;`
  - в) `b = a / b;`  
`c = a / b;`
  - г) `b = a / b + b;`  
`c = a % b + a;`
  - д) `b = a % b + 4;`  
`c = a % b + 1;`
  - е) `b = a / b;`  
`c = a % (b + 1);`
  - ж) `b = a % b`  
`c = a / (b + 1)`
9. Выполните предыдущее задание при `a = -22` и `b = 4`.
10. Напишите программу, которая вводит трёхзначное число и разбивает его на цифры. например, при вводе числа 123 программа должна вывести «1,2,3».
11. Напишите программу, которая вводит координаты двух точек на числовой оси и выводит расстояние между ними.
12. Напишите программу, которая округляет вещественное число до ближайшего целого.
13. Напишите программу, которая вводит два целых числа, `a` и `b` (`a < b`), и выводит на экран 5 случайных целых чисел на отрезке `[a,b]`.
14. Напишите программу, которая моделирует бросание двух игральных кубиков: при запуске выводит случайное число в диапазоне от 2 до 12.
15. Напишите программу, которая случайным образом выбирает дежурных: выводит два различных случайных числа в диапазоне от 1 до `N`, где `N` – количество учеников вашего класса. Какая проблема может при этом возникнуть?
16. Напишите программу, которая вводит два вещественных числа, `a` и `b` (`a < b`), и выводит на экран 5 случайных вещественных чисел в полуинтервале `[a,b)`.

## § 57. Ветвления

### Условный оператор

Возможности, описанные в предыдущих параграфах, позволяют писать *линейные* программы, в которых операторы выполняются последовательно друг за другом, и порядок их выполнения не зависит от входных данных.

В большинстве реальных задач порядок действий может несколько изменяться, в зависимости от того, какие данные поступили. Например, программа для системы пожарной сигнализации

должна выдавать сигнал тревоги, если данные с датчиков показывают повышение температуры или задымленность.

Для этой цели в языках программирования предусмотрены условные операторы. Например, для того, чтобы записать в переменную **M** максимальное из значений переменных **a** и **b**, можно использовать оператор:

```
if ( a > b )
    M = a;
else
    M = b;
```

Здесь использованы два ключевых слова: **if** означает «если», а **else** – «иначе». Если условие в скобках после **if** истинно, выполняется оператор, записанный после скобок, а если условие ложно, то выполняется оператор после слова **else**. Обратите внимание, что после слова **else** никакого условия нет.

В приведенном примере условный оператор записан в полной форме: в обоих случаях (истинно условие или ложно) нужно выполнить некоторые действия. Программа выбора максимального значения может быть написана иначе:

```
M = a;
if ( b > a )
    M = b;
```

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

Для того, чтобы сделать текст программы более понятным, всё тело условного оператора сдвинуто вправо. Вообще говоря, это не обязательно: вся программа на языках C и C++ может быть записана в одну строку. Тем не менее, запись с отступами значительно повышает читаемость программ, и мы далее будем её использовать<sup>6</sup>.

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

```
if ( a > b )
{
    c = a;
    a = b;
    b = c;
}
```

В этом случае нужно записать *составной оператор*, в котором между фигурными скобками может быть сколько угодно команд.

Кроме знаков **<** и **>**, в условиях можно использовать другие знаки отношений: **<=** (меньше или равно), **>=** (больше или равно), **==** (равно) и **!=** (не равно).

Внутри условного оператора могут находиться любые операторы, в том числе и другие условные операторы. Например, пусть возраст Андрея записан в переменной **a**, а возраст Бориса – в переменной **b**. Нужно определить, кто из них старше. Одним условным оператором тут не обойтись, потому что есть три возможных результата: старше Андрей, старше Борис и оба одного возраста. Решение задачи можно записать так:

```
if ( a > b )
    printf("Андрей старше");
else
    if ( a == b )
        printf("Одного возраста");
    else
        printf("Борис старше");
```

```
if ( a > b )
    cout << "Андрей старше";
else
    if ( a == b )
        cout << "Одного возраста";
    else
        cout << "Борис старше";
```

<sup>6</sup> В некоторых языках, например, в языке Python, отступы обязательны, и все строки одного уровня вложенности должны иметь одинаковые отступы.

Условный оператор, проверяющий равенство, находится внутри блока **else**, поэтому он называется *вложенным* условным оператором. Как видно из этого примера, использование вложенных условных операторов позволяет выбрать один из *нескольких* (а не только из двух) вариантов.

Нужно помнить правило: любой блок **else** относится к ближайшему предыдущему оператору **if**, у которого такого блока еще не было. Например, оператор

```
if (a > b) printf("A"); else if (a == b) printf("=");
else printf("B");
```

может быть записан с выделением структуры так:

```
if (a > b)
    printf("A");
else
    if (a == b)
        printf("=");
    else
        printf("B");
```

Здесь второй блок **else** относится к ближайшему (второму, вложенному) условному оператору, поэтому буква Б будет выведена только тогда, когда оба условия окажутся ложными.

## Сложные условия

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

На качестве условия в условном операторе можно указать любое логическое выражение, в том числе сложное условие, составленное из простых отношений с помощью логических операций (связок) «И», «ИЛИ» и «НЕ» (см. главу 3). В языках С и С++ они записываются так: «И» – **&&**, «ИЛИ» – **||** и «НЕ» – **!** (восклицательный знак).

Пусть в переменной **v** записан возраст сотрудника. Тогда нужный фрагмент программы будет выглядеть следующим образом:

```
if ( v >= 25 && v <= 40 )
    printf("подходит");
else
    printf("не подходит");
```

```
if ( v >= 25 && v <= 40 )
    cout << "подходит";
else
    cout << "не подходит";
```

Обратите внимание, что каждое простое условие не обязательно заключать в скобки. Это связано с тем, что в языках С и С++ отношения имеют более *высокий* приоритет, чем логические операции, которые выполняются в таком порядке: сначала все операции «НЕ», затем – «И», и в самом конце – «ИЛИ» (во всех случаях – слева направо). Для изменения порядка действий используют круглые скобки.

## Множественный выбор

Условный оператор предназначен, прежде всего, для выбора одного из двух вариантов (простого ветвления). Иногда нужно сделать выбор из нескольких возможных вариантов.

Пусть, например, в переменной **m** хранится номер месяца, и нужно вывести на экран его русское название. Конечно, в этом случае можно использовать 12 условных операторов:

```
if (m == 1) printf("январь");
if (m == 2) printf("февраль");
...
if (m == 12) printf("декабрь");
```

```
if (m == 1) cout << "январь";
if (m == 2) cout << "февраль";
...
if (m == 12) cout << "декабрь";
```

Вместо многоточия могут быть записаны аналогичные операторы для остальных значений **m**. Но в языках С и С++ для подобных случаев есть специальный оператор выбора:

```
switch ( m ) {
    case 1: printf("январь");
            break;
    case 2: printf("февраль");
            break;
    ...
}
```

```
switch ( m ) {
    case 1: cout << "январь";
            break;
    case 2: cout << "февраль";
            break;
    ...
}
```



```
case 12: printf("декабрь");
        break;
default: printf("ошибка");
}
```

```
case 12: cout << "декабрь";
        break;
default: cout << "ошибка";
}
```

Кроме очевидных 12 блоков здесь добавлен еще один, который сигнализирует об ошибочном номере месяца. Он начинается ключевым словом **default**.

Обратите внимание, что каждый блок заканчивается оператором **break** (в переводе с английского – «прервать»). Если его не поставить, программа перейдет на следующий блок. Поэтому программа

```
switch ( m ) {
case 1:  printf("январь");
case 2:  printf("февраль");
case 3:  printf("март");
default: printf("ошибка");
}
```

```
switch ( m ) {
case 1:  cout << "январь";
case 2:  cout << "февраль";
case 3:  cout << "март";
default: cout << "ошибка";
}
```

для случая  $m = 2$  выведет на экран текст  
февральмартошибка



### Контрольные вопросы

1. Чем отличаются разветвляющиеся алгоритмы от линейных?
2. Как вы думаете, почему не все задачи можно решить с помощью линейных алгоритмов?
3. Как вы думаете, хватит ли линейных алгоритмов и ветвлений для разработки любой программы?
4. Почему нельзя выполнить обмен значений двух переменных в два шага:  $a = b$ ;  $b = a$ ?
5. Чем отличаются условные операторы в полной и неполной формах? Как вы думаете, можно ли обойтись только неполной формой?
6. Какие отношения вы знаете? Как обозначаются отношения «равно» и «не равно»?
7. Что такое сложное условие?
8. Как определяется порядок вычислений в сложном условии?
9. Зачем нужен оператор выбора? Как можно обойтись без него?
10. Как в операторе выбора определить, что нужно делать, если ни один вариант не подошёл?
11. Зачем в операторе выбора используется оператор **break**?
12. Как выполнить для какого-то варианта несколько операторов?



### Задачи и задания

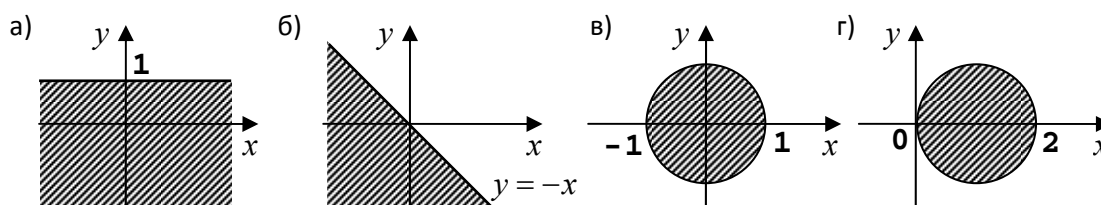
1. Покажите, что приведенная программа не всегда верно определяет максимальное из трёх чисел, записанных в переменные  $a$ ,  $b$  и  $c$ :

```
if ( a > b ) M = a;
else      M = b;
if ( c > b ) M = c;
else      M = b;
```

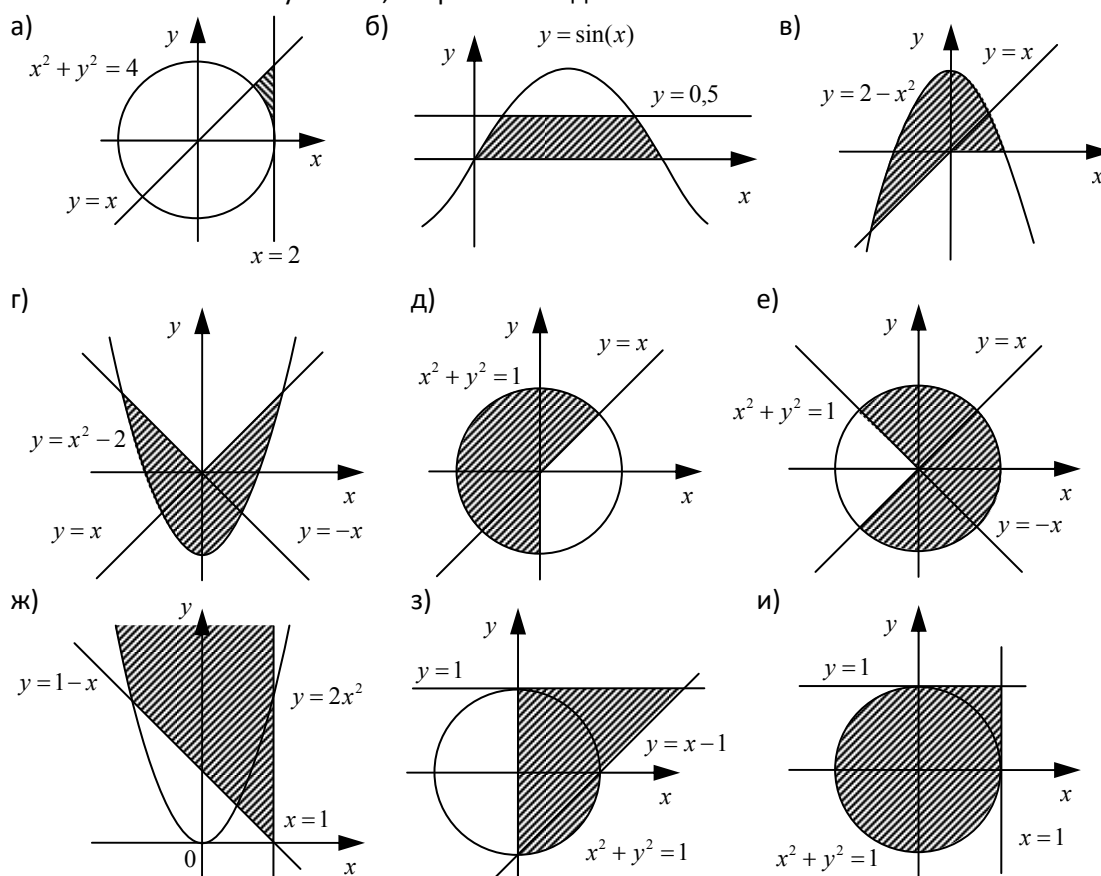
Приведите контрпример, то есть значения переменных, при котором в переменной  $M$  будет получен неверный ответ. Как нужно доработать программу, чтобы она всегда работала правильно?

2. Напишите программу, которая выбирает максимальное и минимальное из пяти введённых чисел.
3. Напишите программу, которая определяет, верно ли, что введённое число – трёхзначное.
4. Напишите программу, которая вводит номер месяца и выводит название времени года. Оператор выбора использовать не разрешается. При вводе неверного номера месяца должно быть выведено сообщение об ошибке.
5. Решите предыдущую задачу с помощью оператора выбора.
6. Напишите программу, которая вводит с клавиатуры номер месяца и определяет, сколько дней в этом месяце. При вводе неверного номера месяца должно быть выведено сообщение об ошибке.

7. Напишите программу, которая вводит с клавиатуры номер месяца и день, и определяет, сколько дней осталось до Нового года. При вводе неверных данных должно быть выведено сообщение об ошибке.
8. Напишите программу, которая вводит возраст человека (целое число, не превышающее 120) и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «25 лет».
9. Напишите программу, которая вводит целое число, не превышающее 100, и выводит его прописью, например, 21 → «двадцать один».
10. Напишите программу, которая вводит координаты точки на плоскости и определяет, попала ли эта точка в заштрихованную область.



11. Напишите два варианта программы, которая вводит координаты точки на плоскости и определяет, попала ли эта точка в заштрихованную область. Один вариант программы должен использовать сложные условия, второй – обходиться без них.



## § 58. Циклические алгоритмы

### Как организовать цикл?

**Цикл** – это многократное выполнение одинаковых действий. Доказано, что любой алгоритм может быть записан с помощью трёх алгоритмических конструкций: циклов, условных операторов и последовательного выполнения команд (линейных алгоритмов).

Простейший цикл, который 10 раз выводит на экран слово «привет», на псевдокоде записывается так:

```
сделай 10 раз
вывод "привет"
```

Подумаем, как можно организовать такой цикл. Вы знаете, что программа после запуска выполняется автоматически. И при этом на каждом шаге нужно знать, сколько раз уже выполнен цикл и сколько ещё осталось выполнить. Для этого необходимо использовать ячейку памяти, в которой будет запоминаться количество выполненных шагов цикла (счётчик шагов). Сначала можно записать в неё ноль (ни одного шага не сделано), а после каждого шага цикла увеличивать значение ячейки на единицу. На псевдокоде алгоритм можно записать так (здесь и далее операции, входящие в тело цикла, выделяются отступами):

```
счётчик = 0
пока счётчик < 10
{
    вывод "привет"
    увеличить счётчик на 1
}
```

Возможен и другой вариант: сразу записать в счётчик нужное количество шагов, и после каждого шага цикла *уменьшать* счётчик на 1. Тогда цикл должен закончиться при нулевом значении счётчика:

```
счётчик = 10
пока счётчик > 0
{
    вывод "привет"
    уменьшить счётчик на 1
}
```

Этот вариант несколько лучше, чем предыдущий, поскольку счётчик сравнивается с нулём, а такое сравнение выполняется в процессоре автоматически (см. главу 4).

В этих примерах мы использовали *цикл с условием*, который выполняется до тех пор, пока некоторое условие не становится ложно.

## Циклы с условием

Рассмотрим следующую задачу: определить количество цифр в десятичной записи целого положительного числа. Будем предполагать, что исходное число записано в переменную **n** целого типа.

Сначала составим алгоритм решения этой задачи. Чтобы считать что-то в программе, нужно использовать переменную, которую называют *счётчиком*. Для подсчёта количества цифр необходимо как-то отсекают эти цифры по одной, с начала или с конца, каждый раз увеличивая счётчик. Начальное значение счётчика должно быть равно нулю, так как до выполнения алгоритма ещё не найдено ни одной цифры.

Для отсекающей первой цифры необходимо заранее знать, сколько цифр в десятичной записи числа, то есть нужно заранее решить ту задачу, которую мы решаем. Следовательно, этот метод не подходит.

Отсечь последнюю цифру проще – достаточно разделить число нацело на 10 (поскольку речь идет о десятичной системе). Операции отсекающей и увеличения счётчика нужно выполнять столько раз, сколько цифр в числе. Как же «поймать» момент, когда цифры кончатся? Несложно понять, что в этом случае результат очередного деления на 10 будет равен нулю, это и говорит о том, что отброшена последняя оставшаяся цифра. Изменение переменной **n** и счётчика для начального значения 1234 можно записать в виде таблицы, показанной справа. Псевдокод выглядит так:

```
счётчик = 0
пока n > 0
{
    отсечь последнюю цифру n
    увеличить счётчик на 1
}
```

n	счётчик
1234	0
123	1
12	2
1	3
0	4

Запись такого цикла на языке C выглядит так:

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

Здесь целочисленная переменная-счётчик имеет имя **count**. Слово **while** переводится с английского как «пока», за ним в скобках записывается условие работы цикла (в данном случае – «пока  $n > 0$ »). Фигурные скобки ограничивают составной оператор. Если в теле цикла нужно выполнить только один оператор, эти скобки можно не ставить. Напомним, что операция деления для целых чисел всегда даёт целое число (остаток отбрасывается).

Обратите внимание, что проверка условия выполняется в начале очередного шага цикла. Такой цикл называется *циклом с предусловием* (то есть с предварительной проверкой условия) или циклом «пока». Если в начальный момент значение переменной **n** будет нулевой или отрицательное, цикл не выполнится ни одного раза.

В данном случае количество шагов цикла «пока» неизвестно, оно равно количеству цифр введенного числа, то есть зависит от исходных данных. Кроме того, этот же цикл может быть использован и в том случае, когда число шагов известно заранее или может быть вычислено:

```
k = 0;
while ( k < 10 )
{
    printf ( "привет\n" );
    k ++;
}
```

```
k = 0;
while ( k < 10 )
{
    cout << "привет\n";
    k ++;
}
```

Если условие в заголовке цикла никогда не нарушится, цикл будет работать бесконечно долго. В этом случае говорят, что «программа зациклилась». Например, если забыть увеличить переменную **k** в предыдущем цикле, программа зациклится:

```
k = 0;
while ( k < 10 )
{
    printf ( "привет\n" );
}
```

```
k = 0;
while ( k < 10 )
{
    cout << "привет\n";
}
```

Во многих языках программирования существует *цикл с постусловием*, в котором условие проверяется после завершения очередного шага цикла. Это полезно в том случае, когда нужно обязательно выполнить цикл хотя бы один раз. Например, пользователь должен ввести с клавиатуры положительное число. Для того, чтобы защитить программу от неверных входных данных, можно использовать такой цикл с постусловием:

```
do {
    printf("Введите n > 0: ");
    scanf ( "%d", &n );
}
while ( n <= 0 );
```

```
do {
    cout << "Введите n > 0: ";
    cin >> n;
}
while ( n <= 0 );
```

Этот цикл закончится тогда, когда условие  $n \leq 0$  нарушится, то есть станет *истинным* условие  $n > 0$ . А это значит, что пользователь ввел допустимое (положительное) значение.

Обратите внимание на важную особенность этого вида цикла: при входе в цикл условие не проверяется, поэтому *цикл всегда выполняется хотя бы один раз*.

## Цикл с переменной

В информатике важную роль играют степени числа 2 (2, 4, 8, 16 и т.д.) Чтобы вывести все степени двойки от  $2^1$  до  $2^{10}$  мы уже можем написать такую программу с циклом «пока»:

```
k = 1;
n = 2;
while ( k <= 10 )
{
    printf ( "%d\n", n );
```

```
k = 1;
n = 2;
while ( k <= 10 )
{
    cout << n << endl;
```

```
n = n * 2;
k++;
}
```

```
n = n * 2;
k++;
}
```

Вы наверняка заметили, что переменная **k** используется трижды (см. выделенные блоки): в операторе присваивания начального значения, в условии цикла и в теле цикла (увеличение на 1). Чтобы собрать все действия с ней в один оператор, во многие языки программирования введен особый вид цикла – цикл с переменной. В заголовке этого цикла задается начальное значение этой переменной, условие продолжения цикла и изменение переменной в конце каждого шага цикла:

```
n = 2;
for ( k = 1; k <= 10; k++)
{
    printf ( "%d\n", n );
    n *= 2;
}
```

```
n = 2;
for ( k = 1; k <= 10; k++)
{
    cout << n << endl;
    n *= 2;
}
```

С каждым шагом цикла переменная цикла может не только увеличиваться, но и уменьшаться. Следующая программа печатает квадраты натуральных чисел от 10 до 1 в порядке убывания:

```
for ( k = 10; k >= 1; k--)
    printf ("%d\n", k*k);
```

```
for ( k = 10; k >= 1; k--)
    cout << k*k << endl;
```

Шаг изменения переменной цикла может быть любым числом. Например, следующий цикл выводит квадраты нечётных чисел:

```
for ( k = 1; k <= 10; k+=2 )
    printf ("%d\n", k*k);
```

```
for ( k = 1; k <= 10; k+=2 )
    cout << k*k << endl;
```

## Вложенные циклы

В более сложных задачах часто бывает так, что на каждом шаге цикла нужно выполнять обработку данных, которая также представляет собой циклический алгоритм. В этом случае получается конструкция «цикл в цикле» или «вложенный цикл».

Предположим, что нужно найти все простые числа в интервале от 2 до 1000. Простейший (но не самый быстрый) алгоритм решения такой задачи на псевдокоде выглядит так:

```
сделать для n от 1 до 1000
если число n простое то
    вывод n
```

Как же определить, что число простое? Как известно, простое число делится только на 1 и само на себя. Если число **n** не имеет делителей в диапазоне от 2 до **n-1**, то оно простое, а если хотя бы один делитель в этом интервале найден, то составное.

Чтобы проверить делимость числа **n** на некоторое число **k**, нужно взять остаток от деления **n** на **k**. Если этот остаток равен нулю, то **n** делится на **k**. Таким образом, программу можно записать так (здесь **n**, **k** и **count** – целочисленные переменные, **count** обозначает счётчик делителей):

```
for ( n = 2; n <= 1000; n++)
{
    count = 0;
    for ( k = 2; k < n; k++ )
        if ( n % k == 0 ) count ++;
    if ( count == 0 )
        printf ("%d\n", n);
}
```

```
for ( n = 2; n <= 1000; n++)
{
    count = 0;
    for ( k = 2; k < n; k++ )
        if ( n % k == 0 ) count ++;
    if ( count == 0 )
        cout << n << endl;
}
```

Попробуем немного ускорить работу программы. Делители числа обязательно идут в парах, причём в любой паре меньший из делителей не превосходит  $\sqrt{n}$  (иначе получается, что произведение двух делителей, каждый из которых больше  $\sqrt{n}$ , будет больше, чем **n**). Поэтому внутренний цикл можно выполнять только до значения  $\sqrt{n}$  вместо **n-1**. Для того, чтобы работать только с целыми числами (и таким образом избежать вычислительных ошибок), лучше заменить условие

$k \leq \sqrt{n}$  на равносильное ему условие  $k^2 \leq n$ . При этом потребуется перейти к внутреннему циклу с условием:

```
count = 0;
k = 2;
while ( k*k <= n )
{
    if ( n % k == 0 ) count ++;
    k ++;
}
```

Чтобы еще ускорить работу цикла, заметим, что когда найден хотя бы один делитель, число уже заведомо составное, и искать другие делители в данной задаче не требуется. Поэтому можно закончить цикл. Для этого в условие работы цикла добавляется условие  $n \% k \neq 0$ , связанное с имеющимся условием с помощью операции «И», при этом можно обойтись без переменной `count`:

```
k = 2;
while ( k*k <= n && n % k != 0 )
    k ++;
if ( k*k > n )
    printf( "%d\n", n );
```

После выхода из цикла мы проверяем, какое условие было нарушено. Если  $k*k > n$  (нарушено первое условие в заголовке цикла), то число  $n$  простое.

В любом вложенном цикле переменная внутреннего цикла изменяется быстрее, чем переменная внешнего цикла. Рассмотрим такой вложенный цикл:

```
for ( i = 1; i <= 4; i++ )
{
    for ( k = 1; k <= i; k++ )
    {
        ...
    }
}
```

На первом шаге (при  $i=1$ ) переменная  $k$  принимает единственное значение 1. Далее, при  $i=2$  переменная  $k$  принимает последовательно значения 1 и 2. На следующем шаге при  $i=3$  переменная  $k$  проходит значения 1, 2 и 3, и т.д.



## Контрольные вопросы

1. Что такое цикл?
2. Сравните цикл с переменной и цикл с условием. Какие преимущества и недостатки есть у каждого из них?
3. Что означает выражение «цикл с предусловием»?
4. В каком случае цикл с предусловием не выполняется ни разу?
5. В каком случае программа, содержащая цикл с условием, может заглохнуть?
6. В каком случае цикл с переменной не выполняется ни разу?
7. Верно ли, что любой цикл с переменной можно заменить циклом с условием? Верно ли обратное утверждение?
8. В каком случае можно заменить цикл с условием на цикл с переменной?
9. Как будет работать приведенная программа, которая считает количество цифр введенного числа, при вводе отрицательного числа? Если вы считаете, что она работает неправильно, укажите, как её нужно доработать.



## Задачи и задания

1. Найдите ошибку в программе:

```
k = 0;
while ( k < 10 )
    printf( "привет\n" );
```

```
k = 0;
while ( k < 10 )
    cout << "привет" << endl;
```



Как её можно исправить?

2. Напишите программу, которая вводит два целых числа и находит их произведение, не используя операцию умножения. Учтите, что числа могут быть отрицательными.
3. Напишите программу, которая вводит натуральное число **N** и находит сумму всех натуральных чисел от 1 до **N**. Используйте сначала цикл с условием, а потом – цикл с переменной.
4. Напишите программу, которая вводит натуральное число **N** и выводит первые **N** чётных натуральных чисел.
5. Напишите программу, которая вводит натуральные числа **a** и **b**, и выводит квадраты натуральных чисел в интервале от **a** до **b**. Например, если ввести 4 и 5, программа должна вывести
 
$$4 * 4 = 16$$

$$5 * 5 = 25$$
6. Напишите программу, которая вводит натуральные числа **a** и **b**, и выводит сумму квадратов натуральных чисел в интервале от **a** до **b**.
7. Напишите программу, которая вводит натуральное число **N** и выводит на экран **N** псевдослучайных чисел. Запустите её несколько раз, объясните результаты опыта.
8. Напишите программу, которая строит последовательность из **N** случайных чисел на отрезке от 0 до 1 и определяет, сколько из них попадает в полуинтервалы  $[0; 0,25)$ ,  $[0,25; 0,5)$ ,  $[0,5; 0,75)$  и  $[0,75; 1)$ . Сравните результаты, полученные при  $N = 10, 100, 1000, 10000$ .
9. Найдите все пятизначные числа, которые при делении на 133 дают в остатке 125, а при делении на 134 дают в остатке 111.
10. Напишите программу, которая вводит натуральное число **N** и выводит на экран все натуральные числа, не превосходящие **N** и делящиеся на каждую из своих цифр.
11. *Числа Армстронга*. Натуральное число называется числом Армстронга, если сумма цифр числа, возведенных в **N**-ную степень (где **N** – количество цифр в числе) равна самому числу. Например,  $153 = 1^3 + 5^3 + 3^3$ . Найдите все трёхзначные и четырёхзначные числа Армстронга.
12. *Аutomorphic числа*. Натуральное число называется автоморфным, если оно равно последним цифрам своего квадрата. Например,  $25^2 = 625$ . Напишите программу, которая вводит натуральное число **N** и выводит на экран все автоморфные числа, не превосходящие **N**.
13. Напишите программу, которая считает количество чётных цифр введённого числа.
14. Напишите программу, которая считает сумму цифр введённого числа.
15. Напишите программу, которая определяет, верно ли, что введённое число содержит две одинаковых цифры, стоящие рядом (как, например, 221).
16. Напишите программу, которая определяет, верно ли, что введённое число состоит из одинаковых цифр (как, например, 222).
17. \*Напишите программу, которая определяет, верно ли, что введённое число содержит по крайней мере две одинаковых цифры, возможно, не стоящие рядом (как, например, 212).
18. Используя сначала цикл с условием, а потом – цикл с переменной, напишите программу, которая выводит на экран чётные степени числа 2 от  $2^{10}$  до  $2^2$  в порядке убывания.
19. *Алгоритм Евклида* для вычисления наибольшего общего делителя двух натуральных чисел, формулируется так: нужно заменять большее число на разность большего и меньшего до тех пор, пока одно из них не станет равно нулю; тогда второе и есть НОД. Напишите программу, которая реализует этот алгоритм. Какой цикл тут нужно использовать?
20. Напишите программу, использующую *модифицированный алгоритм Евклида*: нужно заменять большее число на остаток от деления большего на меньшее до тех пор, пока этот остаток не станет равен нулю; тогда второе число и есть НОД.
21. Добавьте в решение двух предыдущих задач вычисление количества шагов цикла. Заполните таблицу (шаги-1 и шаги-2 означают количество шагов двух версий алгоритма Евклида):

<b>a</b>	<b>64168</b>	<b>358853</b>	<b>6365133</b>	<b>17905514</b>	<b>549868978</b>
<b>b</b>	<b>82678</b>	<b>691042</b>	<b>11494962</b>	<b>23108855</b>	<b>298294835</b>
<b>НОД (a, b)</b>					
<b>шаги-1</b>					
<b>шаги-2</b>					

22. Напишите программу, которая вводит с клавиатуры 10 чисел и вычисляет их сумму и произведение.
23. Напишите программу, которая вводит с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится сумма и произведение введенных чисел (не считая 0).
24. Напишите программу, которая вводит с клавиатуры числа до тех пор, пока не будет введено число 0. В конце работы программы на экран выводится минимальное и максимальное из введенных чисел (не считая 0).
25. Напишите программу, которая вводит с клавиатуры натуральное число **N** и определяет его *факториал*, то есть произведение натуральных чисел от 1 до **N**:  $N! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot N$ . Что будет, если ввести большое значение **N** (например, 20)?
26. Напишите программу, которая вводит натуральные числа **A** и **N** и вычисляет  $A^N$ .
27. Напишите программу, которая выводит на экран все цифры числа, начиная с первой.
28. Ряд чисел Фибоначчи задается следующим образом: первые два числа равны 1 ( $F_1 = F_2 = 1$ ), а каждое следующее равно сумме двух предыдущих:  $F_n = F_{n-1} + F_{n-2}$ . Напишите программу, которая вводит натуральное число **N** и выводит на экран первые **N** чисел Фибоначчи.
29. Напишите программу, которая вводит натуральные числа **a** и **b** и выводит все простые числа в диапазоне от **a** до **b**.
30. Совершенным называется число, равное сумме всех своих делителей, меньших его самого (например, число  $6 = 1 + 2 + 3$ ). Напишите программу, которая вводит натуральное число **N** и определяет, является ли число **N** совершенным.
31. Напишите программу, которая вводит натуральное число **N** и находит все совершенные числа в диапазоне от 1 до **N**.
32. В магазине продается мастика в ящиках по 15 кг, 17 кг, 21 кг. Как купить ровно 185 кг мастики, не вскрывая ящики? Сколькими способами можно это сделать?
33. \*Ввести натуральное число **N** и вывести значение числа  $1/N$ , выделив период дроби. Например,  $1/2 = 0,5$  или  $1/7 = 0,(142857)$ .
34. \*В телевикторине участнику предлагают выбрать один из трёх закрытых чёрных ящиков, причём известно, что в одном из них – приз, а в двух других – пусто. После этого ведущий открывает один пустой ящик (но не тот, который выбрал участник) и предлагает заново сделать выбор, но уже между двумя оставшимися ящиками. Используя псевдослучайные числа, выполните моделирование 1000 раундов этой игры и определите, что выгоднее делать участнику викторины: выбрать тот же ящик, что и в начале игры, или другой<sup>7</sup>.

## § 59. Процедуры

### Что такое процедура?

Предположим, что в нескольких местах программы требуется выводить на экран сообщение об ошибке: «*Ошибка программы*». Это можно сделать, например, так:

```
printf ( "Ошибка программы" );      cout << "Ошибка программы";
```

<sup>7</sup> Эта задача известна как парадокс Монти Холла, потому что её решение противоречит интуиции и «здравому смыслу».

Конечно, можно вставить этот оператор вывода везде, где нужно вывести сообщение об ошибке. Но тут есть две сложности. Во-первых, строка-сообщение хранится в памяти много раз. Во-вторых, если мы задумаем поменять текст сообщения, нужно будет искать эти операторы вывода по всей программе. Для таких случаев в языках программирования предусмотрены *процедуры* – вспомогательные алгоритмы, которые выполняют некоторые действия.

```
void Error()
{
    printf("Ошибка программы");
}

main()
{
    int n;
    scanf( "%d", &n );
    if ( n < 0 ) Error();
    ...
}
```

```
void Error()
{
    cout << "Ошибка программы";
}

main()
{
    int n;
    cin >> n;
    if ( n < 0 ) Error();
    ...
}
```

Фактически мы ввели в язык программирования новую команду **Error**, которая была расшифрована прямо в теле программы. Для того, чтобы процедура заработала, в основной программе (или в другой процедуре) необходимо ее *вызвать* по имени.

Процедура начинается с ключевого слова **void** («пустой», то есть алгоритм, не возвращающий никакого значения). Тело процедуры заключается в фигурные скобки. Процедура расположена выше основной программы, так чтобы в момент *вызова* процедуры транслятор уже знал о ней. Обратите внимание, что и в заголовке процедуры, и при её вызове после имени процедуры нужно ставить круглые скобки (в данном случае – пустые).

Как мы видели, использование процедур сокращает код, если какие-то операции выполняются несколько раз в разных местах программы. Кроме того, большую программу разбивают на несколько процедур для удобства, оформляя в виде процедур отдельные этапы сложного алгоритма. Такой подход делает всю программу более понятной.

## Процедура с параметрами

Процедура **Error** при каждом вызове делает одно и то же. Более интересны процедуры, которым можно передавать аргументы – данные, которые изменяют выполняемые действия. Внутри процедуры эти данные рассматриваются как внутренние (локальные) переменные и называются *параметрами*.

Предположим, что в программе требуется многократно выводить на экран запись целого числа (0..255) в 8-битном двоичном коде. Старшая цифра в такой записи – это частное от деления числа на 128. Далее возьмем остаток от этого деления и разделим на 64 – получается вторая цифра и т.д. Алгоритм, решающий эту задачу для переменной **n**, можно записать так:

```
k = 128;
while ( k > 0 )
{
    printf ( "%d", n / k );
    n = n % k;
    k = k / 2;
}
```

```
k = 128;
while ( k > 0 )
{
    cout << n / k;
    n = n % k;
    k = k / 2;
}
```

Писать такой цикл каждый раз, когда нужно вывести двоичное число, очень утомительно. Кроме того, легко сделать ошибку или опечатку, которую будет сложно найти. Поэтому лучше оформить этот вспомогательный алгоритм в виде процедуры. Но этой процедуре нужно передать *аргумент* – число для перевода в двоичную систему. Программа получается такая:

```
void printBin ( int n )
{
    int k;
    k = 128;
    while ( k > 0 )
    {
```

```
void printBin ( int n )
{
    int k;
    k = 128;
    while ( k > 0 )
    {
```

```

    printf ( "%d", n / k );
    n = n % k;
    k = k / 2;
}
main()
{
    printBin ( 99 );
}

```

```

    cout << n / k;
    n = n % k;
    k = k / 2;
}
main()
{
    printBin ( 99 );
}

```

Основная программа содержит всего одну команду – вызов процедуры **printBin** с аргументом 99. В заголовке процедуры в скобках записывают тип и внутреннее имя параметра (то есть имя, по которому к нему можно обращаться в процедуре).

В процедуре объявлена *локальная* (внутренняя) переменная **k** – она известна только внутри этой процедуры, обратиться к ней из основной программы и из других процедур невозможно.

Параметров может быть несколько, в этом случае они перечисляются в заголовке процедуры через запятую (тип данных для каждого параметра указывается отдельно). Например, процедуру, которая выводит на экран среднее арифметическое двух чисел, можно записать так:

```

void printSred ( int a, int b )
{
    printf ( "%f", (a+b)/2. );
}

```

```

void printSred ( int a, int b )
{
    cout << (a+b)/2.;
}

```

## Изменяемые параметры

Напишем процедуру, которая меняет местами значения двух переменных. Проще всего для этого использовать третью переменную:

```

void Swap ( int a, int b )
{
    int c;
    c = a; a = b; b = c;
}

main()
{
    int x=2, y=3;
    Swap ( x, y );
    printf ( "%d %d", x, y );
}

```

После запуска этой программы обнаружится, что значения переменных **x** и **y** остались прежние: на экран будет выведено «2 3». Дело в том, что процедуры работают с *копиями* переданных ей аргументов. Это значит, что процедура **Swap** создает в памяти временные локальные переменные с именами **a** и **b** и копирует в них переданные значения переменных **x** и **y** основной программы. Поэтому и все перестановки в нашей программе были сделаны именно с копиями, а значения переменных **x** и **y** не изменились.

Чтобы решить проблему, нужно явно сказать, чтобы процедура работала с теми же ячейками памяти, что и основная программа. Проще всего это можно сделать в языке C++: для этого в заголовке процедуры перед именем изменяемой переменной ставят знак **&**:

```

void Swap ( int &a, int &b )
{
    ...
}

```

Теперь процедура решает поставленную задачу: на выходе мы увидим «3 2», что и требовалось. В подобных случаях говорят, что параметры передаются по ссылке, а не по значению. Это означает, что фактически в процедуру передается адрес переменной, и можно изменять значение этой переменной, записывая новые данные по этому адресу.

При вызове такой процедуры **Swap** можно передавать только параметры-переменные, но не константы (постоянные) и не арифметические выражения. Например, вызовы **Swap (2, 3)** и

**Swap(x, y+3)** противоречат правилам языка программирования, и программа выполняться не будет.

В языке С все несколько более сложно: процедуре нужно передавать не сами переменные, а их адреса, чтобы она смогла работать с памятью. Вспомните, что с таким приёмом мы уже встречались при изучении функции **scanf**. Адрес переменной обозначается знаком **&** и процедура должна вызываться так:

```
Swap(&a, &b);
```

Необходимо внести изменения и в саму процедуру:

```
void Swap ( int *adrA, int *adrB )
{
    int c;
    c = *adrA;
    *adrA = *adrB;
    *adrB = c;
}
```

Здесь изменены названия параметров, чтобы показать, что мы работаем с адресами переменных. Для того, чтобы получить значение переменной по адресу **adrA**, нужно использовать запись **\*adrA**.



### Контрольные вопросы

1. Что такое процедуры? В чем смысл их использования?
2. Как оформляются процедуры в языках С и С++?
3. Достаточно ли включить процедуру в текст программы, чтобы она «сработала»?
4. Что такое параметры? Зачем они используются?
5. Какие переменные называются локальными? Где они объявляются?
6. Как оформляются процедуры, имеющие несколько параметров?
7. Что такое изменяемые параметры? Зачем они используются?
8. Как в заголовке процедуры отличить изменяемый параметр от неизменяемого?
9. Чем отличается оформление процедур с изменяемыми параметрами в языках С и С++?



### Задачи и задания

1. Напишите процедуру, которая выводит на экран в столбик все цифры переданного ей числа, начиная с последней.
2. Напишите процедуру, которая выводит на экран в столбик все цифры переданного ей числа, начиная с первой.
3. Напишите процедуру, которая выводит на экран все делители переданного ей числа (в одну строку).
4. \*Напишите процедуру, которая выводит на экран запись переданного ей числа в римской системе счисления.
5. Напишите процедуру, которая выводит на экран запись числа, меньшего, чем  $8^{10}$ , в виде 10 знаков в восьмеричной системе счисления.
6. Напишите процедуру, которая выводит на экран запись числа, меньшего, чем  $2^4 = 65536$ , в виде 4-х знаков в шестнадцатеричной системе счисления.
7. Напишите процедуру, которая принимает параметр – натуральное число  $N$  – и выводит на экран линию из  $N$  символов '- '.
8. Напишите процедуру, которая принимает параметр – натуральное число  $N$  – и выводит на экран квадрат из звездочек со стороной  $N$ .
9. Напишите процедуру, которая принимает числовой параметр – возраст человека в годах, и выводит этот возраст со словом «год», «года» или «лет». Например, «21 год», «22 года», «12 лет».
10. Напишите процедуру, которая выводит переданное ей число прописью. Например, 21 → «двадцать один».

11. Напишите процедуру, которая принимает параметр – натуральное число  $N$  – и выводит первые  $N$  чисел Фибоначчи (см. задания к § 58. ).
12. Напишите процедуру, которая определяет, верно ли, что переданное ей число – простое. (Используйте изменяемые параметры).

## § 60. Функции

### Пример функции

С функциями вы уже знакомы, потому что наверняка применяли стандартные функции языка программирования (например, **abs**, **sin**, **cos**). Функция, как и процедура – это вспомогательный алгоритм, который может принимать аргументы. Но, в отличие от процедуры, функция всегда возвращает *значение-результат*. Результатом может быть число, символ, или объект другого типа.

Составим функцию, которая вычисляет сумму цифр числа. Будем использовать следующий алгоритм (предполагается, что число записано в переменной **n**):

```
сумма = 0
пока n != 0
{
    сумма = сумма + n % 10
    n = n / 10;
}
```

Чтобы получить последнюю цифру числа (которая добавляется к сумме) нужно взять остаток от деления числа на 10. Затем последняя цифра отсекается, и мы переходим к следующей цифре. Цикл продолжается до тех пор, пока значение **n** не становится равно нулю.

Пока остается неясно, как указать в программе, чему равно значение функции? Для этого в языках C и C++ используют специальный оператор **return** (от англ. «вернуть»), после которого записывают значение-результат:

```
int sumDigits ( int n )
{
    int sum = 0;
    while ( n != 0 )
    {
        sum += n % 10;
        n /= 10;
    }
    return sum;
}

main()
{
    printf ( "%d", sumDigits(12345) );
}
```

Обратим внимание на особенности записи: тип возвращаемого значения (**int**) указывается в заголовке функции перед именем функции. Так же как и в процедурах, в функциях можно объявлять и использовать локальные переменные. Они входят в «зону видимости» только этой функции, для всех остальных функций и процедур они недоступны.

В функции может быть несколько операторов **return**, после выполнения любого из них работа функции заканчивается.

Функции, созданные в программе таким образом, применяются точно так же, как и стандартные функции. Их можно вызывать везде, где может использоваться выражение того типа, который возвращает функция. Приведем несколько примеров на языке C:

```
x = 2*sumDigits(n+5);
z = sumDigits(k) + sumDigits(m);
if ( sumDigits(n) % 2 == 0 )
{
    printf ( "Сумма цифр чётная\n" );
}
```



```
printf ( "Она равна %d", sumDigits(n) );
}
```

## Логические функции

Достаточно часто применяют специальный тип функций, которые возвращают *логическое значение* (да или нет, **true** или **false**). Иначе говоря, такая *логическая* функция отвечает на вопрос «да или нет?» и возвращает 1 бит информации.

Вернёмся к задаче, которую мы уже рассматривали: вывести на экран все простые числа в диапазоне от 2 до 1000. Алгоритм определения простоты числа оформим в виде функции. При этом его можно будет легко вызвать из любой точки программы.

Запишем основную программу на псевдокоде:

```
для i от 2 до 100
  если i - простое то
    вывод i
```

Предположим, что у нас уже есть логическая функция **isPrime**, которая определяет простоту числа, переданного ей как параметр, и возвращает **true** («истинно»), если число простое, и **false** («ложно») в противном случае. Такую функцию можно использовать вместо выделенного блока алгоритма:

```
if ( isPrime(i) )
  printf ( "%d\n", i );
```

```
if ( isPrime(i) )
  cout << i << endl;
```

Остается написать саму функцию **isPrime**. Будем использовать уже известный алгоритм: если число **n** в интервале от 2 до  $\sqrt{n}$  не имеет ни одного делителя, то оно простое<sup>8</sup>:

```
bool isPrime ( int n )
{
  int k = 2;
  while ( k*k <= n && n%k != 0 )
    k ++;
  return ( k*k > n );
}
```

Эта функция возвращает логическое значение, на это указывает ключевое слово **bool** (от *boolean* – булевский, логический, в честь Дж. Буля). Значение функции определяется оператором

```
return ( k*k > n );
```

Это оператор возвращает результат логического выражения **k\*k > n**. Если это выражение истинно (значение счётчика равно 0), то возвращается значение **true**, иначе – значение **false**.

Для того, чтобы использовать переменные типа **bool** в языке C, в начале программы нужно подключить файл **stdbool.h** с помощью директивы **#include**:

```
#include <stdbool.h>
```

В этом файле константе **true** присваивается значение 1, а константе **false** – значение 0. Дело в том, что в языке C любое значение, отличное от нуля, соответствует истинному условию, а нулевое значение – ложному. Поэтому логическая функция может возвращать и целое значение – 1 как ответ «да» и 0 как ответ «нет». Форма вызова такой функции не меняется.

Логические функции можно использовать так же, как и любые условия: в условных операторах и циклах с условием. Например, такой цикл останавливается на первом введённом составном числе:

```
scanf ( "%d", &n );
while ( isPrime(n) )
{
  printf ("%d - простое число\n");
  scanf ( "%d", &n );
}
```

```
cin >> n;
while ( isPrime(n) )
{
  cout << n <<
    " - простое число" << endl;
  cin >> n;
}
```

<sup>8</sup> Эту программу можно еще немного усовершенствовать: после числа 2 имеет смысл проверять только нечётные делители, увеличивая на каждом шаге значение **k** сразу на 2.



## Контрольные вопросы

1. Что такое функция? Чем она отличается от процедуры?
2. Как оформляются функции в тексте программы?
3. Как по тексту программы определить, какое значение возвращает функция?
4. Какие функции называются логическими? Зачем они нужны?
5. Обязательно ли логическая функция должна возвращать значение типа **bool**?



## Задачи и задания

1. Напишите функцию, которая вычисляет максимальное из трёх чисел.
2. На соревнованиях выступление спортсмена оценивают 5 экспертов, каждый из них выставляет оценку в баллах (целое число). Для получения итоговой оценки лучшая и худшая из оценок экспертов отбрасываются, а для оставшихся трёх находится среднее арифметическое. Напишите функцию, которая вводит 5 оценок экспертов и возвращает итоговую оценку выступления спортсмена.
3. Напишите функцию, которая вычисляет количество цифр числа.
4. Напишите функцию, которая вычисляет наибольший общий делитель двух чисел.
5. Напишите функцию, которая вычисляет наименьшее общее кратное двух чисел.
6. Напишите функцию, которая «разворачивает» десятичную запись числа наоборот, например, из 123 получается 321, и из 3210 – 0123.
7. Напишите функцию, которая моделирует бросание двух игральных кубиков (на каждом может выпасть от 1 до 6 очков). (Используйте генератор псевдослучайных чисел.)
8. Напишите функцию, которая вычисляет факториал натурального числа **N**.
9. Напишите функцию, которая вычисляет **N**-ое число Фибоначчи.
10. Дружественные числа – это два натуральных числа, таких, что сумма всех делителей одного числа (меньших самого этого числа) равна другому числу, и наоборот. Найдите все пары дружественных чисел, каждое из которых меньше 10000. Используйте функцию, которая вычисляет сумму делителей числа.
11. Напишите программу, которая вводит натуральное число **N** и находит все числа в диапазоне  $[0, N]$ , сумма цифр которых не меняется при умножении на 2, 3, 4, 5, 6, 7, 8 и 9 (например, число 9). Используйте функцию для вычисления суммы цифр числа.
12. Напишите логическую функцию, которая определяет, верно ли, что число **N** – совершенное, то есть равно сумме своих делителей, меньших его самого.
13. Простое число называется гиперпростым, если любое число, получающееся из него откидыванием нескольких цифр с конца, тоже является простым. Например, число 733 – гиперпростое, так как и оно само, и числа 73 и 7 – простые. Напишите логическую функцию, которая определяет, верно ли, что число **N** – гиперпростое. Используйте уже готовую функцию **isPrime**.

## § 61. Рекурсия

### Что такое рекурсия?

Вспомним определение натуральных чисел. Оно состоит из двух частей:

- 1) 1 – натуральное число;
- 2) если  $n$  – натуральное число, то  $n + 1$  – тоже натуральное число.

Вторая часть этого определения в математике называется *индуктивным*: натуральное число определяется через другое натуральное число, и таким образом определяется всё множество натуральных чисел. В программировании этот приём называют *рекурсией*.

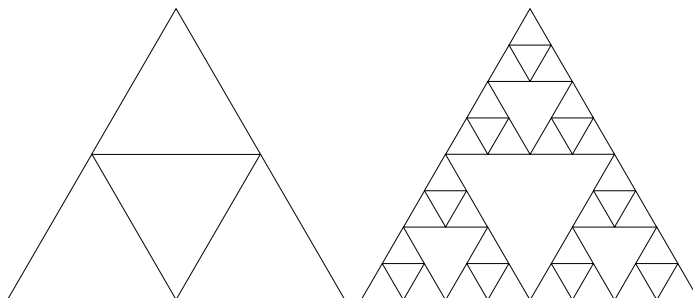
**Рекурсия** — это способ определения множества объектов через само это множество на основе заданных простых базовых случаев.

Первая часть в определении натуральных чисел – это и есть тот самый базовый случай. Если убрать первую часть из определения, оно будет неполно: вторая часть даёт только метод перехода к следующему значению, но не даёт никакой «зацепки», не отвечает на вопрос «откуда начать».

Похожим образом задаются *числа Фибоначчи*: первые два числа равны 1, а каждое из следующих чисел равно сумме двух предыдущих:

- 1)  $F_1 = F_2 = 1$ ,
- 2)  $F_n = F_{n-1} + F_{n-2}$  для  $n > 2$ .

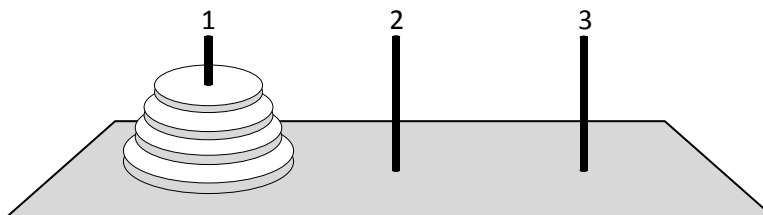
Популярные примеры рекурсивных объектов – *фракталы*. Так в математике называют геометрические фигуры, обладающие *самоподобием*. Это значит, что они составлены из фигур меньшего размера, каждая из которых подобна целой фигуре. На рисунке показан треугольник Серпинского – один из первых фракталов, который предложил в 1915 году польский математик В. Серпинский.



Равносторонний треугольник делится на 4 равных треугольника меньшего размера (левый рисунок), затем каждый из полученных треугольников, кроме центрального, снова делится на 4 ещё более мелких треугольника и т.д. На правом рисунке показан треугольник Серпинского с тремя уровнями деления.

## Ханойские башни

Согласно легенде, конец света наступит тогда, когда монахи Великого храма города Бенарас смогут переложить 64 диска разного диаметра с одного стержня на другой. Вначале все диски наизаны на первый стержень. За один раз можно перекладывать только один диск, причем разрешается класть только меньший диск на больший, но не наоборот. Есть также и третий стержень, который можно использовать в качестве вспомогательного.



Решить задачу для 2, 3 и даже 4-х дисков довольно просто. Проблема в том, чтобы составить алгоритм для любого числа дисков.

Пусть нужно перенести  $n$  дисков со стержня 1 на стержень 3. Представим себе, что мы как-то смогли переместить  $n-1$  дисков на вспомогательный стержень 2. Тогда остается перенести самый большой диск на стержень 3, а затем  $n-1$  меньших диска со вспомогательного стержня 2 на стержень 3, и задача будет решена. Получается такой псевдокод:

```
перенести ( n-1, 1, 2 )
1 -> 3
перенести ( n-1, 2, 3 )
```

Здесь запись  $1 \rightarrow 3$  обозначает «перенести один диск со стержня 1 на стержень 3». Процедура **перенести** (которую нам предстоит написать) принимает 3 параметра: число дисков, начальный и конечный стержни. Таким образом, мы свели задачу переноса  $n$  дисков к двум задачам переноса  $n-1$  дисков и одному элементарному действию – переносу 1 диска. Заметим, что при известных номерах начального и конечного стержней легко рассчитать номер вспомогательного, так как сумма номеров равна  $1 + 2 + 3 = 6$ . Получается такая (пока не совсем верная) процедура:

```
void Hanoi ( int n, int k, int m )
{
    int p;
    p = 6 - k - m;
    Hanoi ( n-1, k, p );
    printf ( "%d -> %d\n", k, m );
    Hanoi ( n-1, p, m );
}
```

Эта процедура вызывает сама себя, но с другими параметрами. Такая процедура называется рекурсивной.

**Рекурсивная процедура (функция)** — это процедура (функция), которая вызывает сама себя напрямую или через другие процедуры и функции.

Основная программа для решения задачи, например, с четырьмя дисками, будет содержать всего одну строку (перенести 4 диска со стержня 1 на стержень 3):

```
Hanoi( 4, 1, 3 );
```

Если вы попытаете запустить эту программу, она заикнется, то есть будет работать бесконечно долго. В чем же дело? Вспомните, что определение рекурсивного объекта состоит из двух частей. В первой части определяются базовые объекты (например, первое натуральное число), именно эта часть «отвечает» за остановку рекурсии в программе. Пока мы не определили такое условие остановки, и процедура бесконечно вызывает сама себя (это можно проверить, выполняя программу по шагам). Когда же остановить рекурсию?

Очевидно, что если нужно переложить 0 дисков, задача уже решена, ничего перекладывать не надо. Это и есть условие окончания рекурсии: если значение параметра **n**, переданное процедуре, равно 0, нужно выйти из процедуры без каких-либо действий (с помощью оператора **return**). Правильная версия процедуры выглядит так:

```
void Hanoi ( int n, int k,
             int m )
{
    int p;
    if ( n == 0 ) return;
    p = 6 - k - m;
    Hanoi ( n-1, k, p );
    printf ( "%d -> %d\n",
            k, m );
    Hanoi ( n-1, p, m );
}
```

```
void Hanoi ( int n, int k,
             int m )
{
    int p;
    if ( n == 0 ) return;
    p = 6 - k - m;
    Hanoi ( n-1, k, p );
    cout << k << " -> "
         << m << endl;
    Hanoi ( n-1, p, m );
}
```

Чтобы переложить **N** дисков, нужно выполнить  $2^N - 1$  перекладываний. Для **N=64** это число равно 18 446 744 073 709 551 615. Если бы монахи, работая день и ночь, каждую секунду перемещали один диск, им бы потребовалось 580 миллиардов лет.

## Примеры

**Пример 1.** Составим процедуру, которая переводит натуральное число в двоичную систему. Мы уже занимались вариантом этой задачи, где требовалось вывести 8-битную запись числа из диапазона 0..255, сохранив лидирующие нули. Теперь усложним задачу: лидирующие нули выводить не нужно, а натуральное число может быть любым (в пределах допустимого диапазона для выбранного типа данных).

Стандартный алгоритм перевода числа в двоичную систему можно записать, например, так:

```
пока n != 0
{
    вывод n % 2
    n = n / 2
}
```

Проблема в том, что двоичное число выводится «задом наперёд», то есть первым будет выведен последний разряд. Как «перевернуть число»?

Есть разные способы решения этой задачи, которые сводятся к тому, чтобы запоминать остатки от деления (например, в символьной строке) и затем, когда результат полностью получен, вывести его на экран.

Однако можно применить красивый подход использующий рекурсию. Идея такова: чтобы вывести двоичную запись числа  $n$ , нужно сначала вывести двоичную запись числа  $n/2$ , а затем – последнюю цифру, то есть  $n \% 2$ . Если полученное число-параметр равно нулю, нужно выйти из процедуры. Такой алгоритм очень просто программируется:

```
void printBin ( int n )
{
    if ( n == 0 ) return;
    printBin( n / 2 );
    printf ( "%d", n % 2 );
}
```

```
void printBin ( int n )
{
    if ( n == 0 ) return;
    printBin( n / 2 );
    cout << n % 2;
}
```

Конечно, то же самое можно было сделать и с помощью цикла. Поэтому можно сделать важный вывод: *рекурсия заменяет цикл*. При этом программа во многих случаях становится более понятной.

**Пример 2.** Составим функцию, которая вычисляет сумму цифр числа. Будем рассуждать так: сумма цифр числа  $n$  равна значению последней цифры плюс сумма цифр числа  $n/10$ . Сумма цифр однозначного числа равна самому этому числу, это условие окончания рекурсии. Получаем следующую функцию:

```
int sumDig ( int n )
{
    int sum;
    sum = n % 10;
    if ( n >= 10 )
        sum += sumDig ( n / 10 );
    return sum;
}
```

**Пример 3.** Алгоритм Евклида, один из древнейших известных алгоритмов, предназначен для поиска наибольшего общего делителя (НОД) двух натуральных чисел. Формулируется он так:

**Алгоритм Евклида.** Чтобы найти НОД двух натуральных чисел, нужно вычитать из большего числа меньшее до тех пор, пока меньшее не станет равно нулю. Тогда второе число и есть НОД исходных чисел.

Этот алгоритм может быть сформулировать в рекурсивном виде. Во-первых, в нём для перехода к следующему шагу используется равенство  $\text{НОД}(a, b) = \text{НОД}(a - b, b)$  при  $a \geq b$ . Кроме того, задано условие останова: если одно из чисел равно нулю, то НОД совпадает со вторым числом. Поэтому можно написать такую рекурсивную функцию:

```
int NOD ( int a, int b )
{
    if ( a == 0 || b == 0 )
        return a + b;
    if ( a > b )
        return NOD ( a - b, b );
    else return NOD ( a, b - a );
}
```

Заметим, что при равенстве одного из чисел нулю второе число совпадает с суммой двух, поэтому в качестве результата функции принимается  $a + b$ .

Существует и более быстрый вариант алгоритма Евклида (*модифицированный алгоритм*), в котором большее число заменяется на остаток от деления большего на меньшее:

```
int NOD ( int a, int b )
{
    if ( a == 0 || b == 0 )
        return a + b;
    if ( a > b )
        return NOD ( a % b, b );
}
```

```

    else return NOD ( a, b%a );
}

```

Эту функцию можно еще значительно упростить. Дело в том, что остаток от деления **a** на **b** всегда меньше делителя **b**, поэтому при очередном рекурсивном вызове можно передавать функции **NOD** сначала большее число, а потом – меньшее; это позволит избавиться от условного оператора:

```

int NOD ( int a, int b )
{
    if ( b == 0 ) return a;
    return NOD ( b, a%b );
}

```

Заметьте, что условие окончания рекурсии тоже упростилось: достаточно проверить, что на очередном шаге остаток от деления (второй параметр) стал равен нулю, тогда результат – это значение первого параметра **a**.

## Как работает рекурсия

Рассмотрим вычисление *факториала*  $N!$ : так называют произведение всех натуральных чисел от 1 до заданного числа  $N$ :  $N! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot N$ . Факториал может быть также введен с помощью рекуррентной формулы, которая связывает факториал данного числа с факториалом предыдущего:

$$N! = \begin{cases} 1, & N = 1 \\ N \cdot (N-1)!, & N \geq 2 \end{cases}$$

Здесь первая часть описывает базовый случай (условие окончания рекурсии), а вторая – переход к следующему шагу. Запишем соответствующую функцию на языке C, добавив в начале и в конце операторы вывода:

<pre> int Fact ( int N ) {     int F;     printf ( "-&gt; N=%d\n", N );     if ( N &lt;= 1 )         F = 1;     else F = N * Fact ( N - 1 );     printf ( "&lt;- N=%d\n", N );     return F; } </pre>	<pre> -&gt; N=3 -&gt; N=2 -&gt; N=1 &lt;- N=1 &lt;- N=2 &lt;- N=3 </pre>
---	--

Справа от программы показан протокол ее работы при вызове **Fact (3)** (для наглядности сделаны отступы, показывающие вложенность вызовов). Из протокола видно, что вызов **Fact (2)** происходит раньше, чем заканчивается вызов **Fact (3)**. Это значит, что компьютеру нужно где-то (без помощи программиста) запомнить состояние программы (в том числе значения всех локальных переменных) и адрес, по которому нужно вернуться после завершения вызова **Fact (2)**. Для этой цели используется *стек*.

**Стек** (англ. *stack* – кipa, стопка) – особая область памяти, в которой хранятся локальные переменные и адреса возврата из процедур и функций.

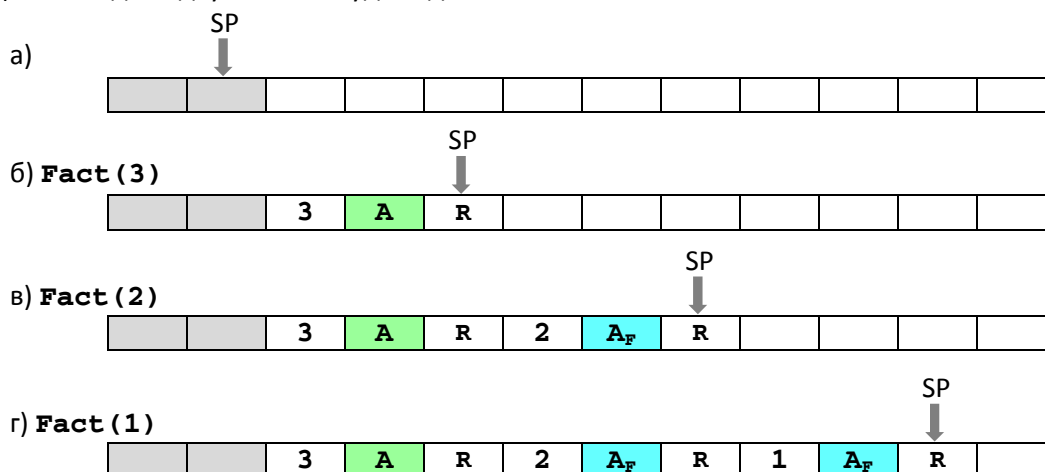
Один из регистров процессора называется *указателем стека* (англ. *stack pointer = SP*) – в нём записан адрес последней занятой ячейки стека. При вызове процедуры в стек помещаются значения всех ее параметров, адрес возврата и выделяется место под локальные переменные.

На рисунке *a* показано начальное состояние стека, серым цветом выделены занятые ячейки. Когда функция **Fact** вызывается из основной программы с параметром 3, в стек записывается значение параметра, затем – адрес возврата *A*, и выделяется место для результата  $R^9$  (рисунк *б*). При втором и третьем вложенных вызовах в стек добавляются аналогичные блоки данных (рисун-

<sup>9</sup> Результат может передаваться вызывающей программе также через регистры процессора, без использования стека.



ки в и г). Заметьте, что в нашем случае адрес возврата  $A_F$  (точка после вызова в рекурсивной функции) в последних двух блоках будет один и тот же.



Когда выполняется возврат из процедуры, состояние стека изменяется в обратную сторону: г – в – б – а.

Что же следует из этой схемы? Во-первых, с каждым новым вызовом расходуется дополнительная стековая память. Если вложенных вызовов будет очень много (или если процедура создает много локальных переменных), эта память закончится, и программа завершится аварийно.

Во-вторых, при каждом вызове процедуры некоторое время затрачивается на выполнение служебных операций (занесение данных в стек и т.п.), поэтому, как правило, рекурсивные программы выполняются несколько дольше, чем аналогичные нерекурсивные.

А всегда ли можно написать нерекурсивную программу? Оказывается всегда. Доказано, что любой рекурсивный алгоритм может быть записан без использования рекурсии (хотя часто при этом программа усложняется и становится менее понятной). Например, для вычисления факториала можно использовать обычный цикл:

```
F = 1;
for ( i = 2; i <= N; i++ )
    F = F * i;
```

В данном случае такой *итерационный* (то есть повторяющийся, циклический) алгоритм значительно лучше, чем рекурсивный: он не расходует стековую память и выполняется быстрее. Поэтому здесь нет никакой необходимости использовать рекурсию.

Итак, рекурсия – это мощный инструмент, заменяющий циклы в задачах, которые можно свести к более простым задачам того же типа. В сложных случаях использование рекурсии позволяет значительно упростить программу, сократить ее текст и сделать более понятной.

С другой стороны, если существует простое решение задачи без использования рекурсии, лучше применить именно его. Нужно стараться обходиться без рекурсии, если вложенность вызовов получается очень большой, или процедура использует много локальных данных.



## Контрольные вопросы

1. Что такое рекурсия? Приведите примеры.
2. Как вы думаете, почему любое рекурсивное определение состоит из двух частей?
3. Что такое рекурсивная процедура (функция)?
4. Расскажите о задаче «Ханойские башни». Попытайтесь придумать алгоритм ее решения, не использующий рекурсию.
5. Процедура А вызывает процедуру Б, а процедура Б – процедуру А и сама себя. Какую из них можно назвать рекурсивной?
6. В каком случае рекурсия никогда не остановится? Докажите, что в рассмотренных задачах этого не случится.
7. Что такое стек? Как он используется при выполнении программ?
8. Почему при использовании рекурсии может случиться переполнение стека?

9. Назовите достоинства и недостатки рекурсии. Когда ее следует использовать, а когда – нет?



### Задачи и задания

1. Найдите в Интернете информацию об использовании рекурсии в искусстве и рекламе. Сделайте сообщение в классе.
2. Найдите в Интернете информацию о фракталах. Сделайте сообщение в классе.
3. Используя материалы Интернета, ответьте на вопрос: «Как связаны числа Фибоначчи с кроликами»?
4. Придумайте свою рекурсивную фигуру и опишите её.
5. \*Используя графические возможности модуля **turtle**, постройте на экране треугольник Серпинского и другие фракталы.
6. Напишите рекурсивную процедуру для перевода числа в двоичную систему, которая правильно работала бы для нуля (выводила 0).
7. \*Напишите рекурсивную процедуру для перевода числа в шестнадцатеричную систему счисления.
8. \*Напишите рекурсивную процедуру для перевода числа в троичную уравновешенную систему счисления (см. § 14). Вместо цифры  $\bar{1}$  используйте символ «#».
9. \*Дано натуральное число N. Требуется получить и вывести на экран все возможные *различные* способы представления этого числа в виде суммы натуральных чисел (то есть,  $1 + 2$  и  $2 + 1$  – это один и тот же способ разложения числа 3). Решите задачу с помощью рекурсивной процедуры
10. Напишите рекурсивную процедуру для перевода числа из двоичной системы счисления в десятичную.
11. \*Напишите рекурсивную процедуру для перевода числа из шестнадцатеричной системы счисления в десятичную.
12. \*Напишите рекурсивную процедуру для перевода числа из троичной уравновешенной системы счисления (см. § 14) в десятичную. Вместо цифры  $\bar{1}$  используйте символ «#».
13. Напишите рекурсивную и нерекурсивную функции, вычисляющие НОД двух натуральных чисел с помощью модифицированного алгоритма Евклида. Какой вариант вы предпочтете?

## § 62. Массивы

### Что такое массив?

Основное предназначение современных компьютеров – обработка большого количества данных. При этом надо как-то обращаться к каждой из тысяч (или даже миллионов) ячеек с данными. Очень сложно дать каждой ячейке собственное имя и при этом не запутаться. Из этой ситуации выходят так: дают имя не ячейке, а группе ячеек, в которой каждая ячейка имеет собственный номер. Такая область памяти называется массивом.

**Массив** – это группа переменных одного типа, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка в массиве имеет уникальный номер.

Для работы с массивами нужно, в первую очередь, научиться:

- выделять память нужного размера под массив;
- записывать данные в нужную ячейку;
- читать данные из ячейки массива.

Чтобы использовать массив, надо его объявить – определить тип массива (тип входящих в него элементов), выделить место в памяти и присвоить имя. Имена массивов строятся по тем же правилам, что и имена переменных.

В языках C и C++ массивы объявляются почти так же, как и обычные переменные, только после имени массива в квадратных скобках указывают количество элементов:

```
int A[5];
double V[8];
bool L[10];
char S[80];
```

Индексы элементов массива всегда начинаются с нуля. Если, например, в массиве **A** пять элементов, то последний элемент будет иметь индекс 4.

Для того, чтобы обратиться к элементу массива, нужно записать имя массива и в квадратных скобках индекс нужного элемента, например, **A[3]**. Индексом может быть не только число, но значение целой переменной или арифметического выражения целого типа. В этом примере массив заполняется квадратами первых натуральных чисел:

```
main()
{
    const int N = 10;
    int A[N];
    int i;
    for ( i = 0; i < N; i++ )
        A[i] = i*i;
}
```

При объявлении границ индексов массивов можно использовать *константы* – постоянные величины, имеющие имя. В приведенном примере с помощью ключевого слова **const** объявлена целочисленная (**int**) константа **N**, равная 10. Константы обычно вводятся выше блока объявления переменных. Использование констант очень удобно, потому что при изменении размера массива в программе нужно поменять только одно число – значение этой константы.

Далее во всех примерах мы будем считать, что в программе объявлен целочисленный массив **A**, состоящий из **N** элементов (с индексами от 0 до **N-1**), а также целочисленная переменная **i**, которая будет обозначать индекс элемента массива. Чтобы ввести такой массив или вывести его на экран, нужно использовать цикл, то есть ввод и вывод массива выполняется поэлементно:

```
for ( i = 0; i < N; i++ )
{
    printf ( "A[%d]=", i );
    scanf ( "%d", &A[i] );
}
for ( i = 0; i < N; i++ )
    printf ( "%d ", A[i] );
```

```
for ( i = 0; i < N; i++ )
{
    cout << "A[" << i << "]=";
    cin >> A[i];
}
for ( i = 0; i < N; i++ )
    cout << A[i] << " ";
```

В этом примере перед вводом очередного элемента массива на экран выводится подсказка. Например, при вводе 3-го элемента будет выведено «**A[3]=**». После вывода каждого элемента ставится пробел, иначе все значения сольются в одну строку.

В учебных примерах массивы часто заполняют случайными числами:

```
for ( i = 0; i < N; i++ )
{
    A[i] = irand ( 20, 100 );
    printf ( "%d ", A[i] );
}
```

```
for ( i = 0; i < N; i++ )
{
    A[i] = irand ( 20, 100 );
    cout << A[i] << " ";
}
```

Здесь использована функция **irand**, которая возвращает псевдослучайное целое число в заданном диапазоне. Такой функции нет в стандартной библиотеке языка C, но её легко написать (вспомните материал § 56):

```
int irand ( int a, int b )
{
    return a + rand() % (b - a + 1);
}
```

Напомним, что для работы с датчиком случайных чисел в языке C нужно подключить с помощью директивы **include** заголовочный файл **stdlib.h**, а в языке C++ – файл **cstdlib**.

## Перебор элементов

Перебор элементов состоит в том, что мы в цикле просматриваем все элементы массива и, если нужно, выполняем с каждым из них некоторую операцию. Для этого удобнее всего использовать цикл с переменной, которая изменяется от минимального до максимального индекса. Поскольку мы будем работать с массивом, элементы которого имеют индексы от 0 до **N-1**, цикл выглядит так:

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

Здесь вместо многоточия можно добавлять операторы, работающие с элементом **A[i]**.

Во многих задачах нужно найти в массиве все элементы, удовлетворяющие заданному условию, и как-то их обработать. Простейшая из таких задач – подсчёт нужных элементов. Для решения этой задачи нужно ввести переменную-счётчик, начальное значение которой равно нулю. Далее в цикле (от 0 до **N-1**) просматриваем все элементы массива. Если для очередного элемента выполняется заданное условие, то увеличиваем счётчик на 1. На псевдокоде этот алгоритм выглядит так:

```
счётчик = 0;
for ( i = 0; i < N; i++ )
    if ( условие выполняется для A[i] )
        счётчик ++;
```

Предположим, что в массиве **A** записаны данные о росте игроков баскетбольной команды. Найдём количество игроков, рост которых больше 180 см, но меньше 190 см. В следующей программе используется переменная-счётчик **count**:

```
count = 0;
for ( i = 0; i < N; i++ )
    if ( 180 < A[i] && A[i] < 190 )
        count ++;
```

Теперь усложним задачу: требуется найти средний рост этих игроков. Для этого нужно дополнительно в отдельной переменной складывать все нужные значения, а после завершения цикла разделить эту сумму на количество. Начальное значение переменной **sum**, в которой накапливается сумма, тоже должно быть равно нулю.

```
count = 0;
sum = 0;
for ( i = 0; i < N; i++ )
    if ( 180 < A[i] && A[i] < 190 ) {
        count ++;
        sum += A[i];
    }
printf ( "%f", (float)sum / count );
```

Обратите внимание, что в последней строке для того, чтобы получить вещественный (а не округленный целочисленный) результат деления, значение переменной **sum** преобразовано к вещественному числу с помощью записи **(float)sum**.



## Контрольные вопросы

1. Что такое массив? Зачем нужны массивы?
2. Зачем нужно объявлять массивы?
3. Как объявляются массивы?
4. Как вы думаете, почему элементы массива расположены в памяти рядом?
5. Как обращаются к элементу массива?
6. Может ли нумерация элементов массива в языках C и C++ начинаться не с 0, а с другого числа?
7. Почему размер массива лучше вводить как константу, а не число?
8. Как ввести массив и вывести его на экран?
9. Как заполнить массив случайными числами в диапазоне от 100 до 200?



## Задачи и задания

1. Заполните массив элементами арифметической прогрессии. Её первый элемент и разность нужно ввести с клавиатуры.
2. Заполните массив степенями числа 2 (от  $2^1$  до  $2^N$ ).

3. Заполните массив первыми числами Фибоначчи.
4. \*Заполните массив из N элементов случайными целыми числами в диапазоне 1..N так, чтобы в массив обязательно вошли все числа от 1 до N (постройте случайную перестановку).
5. \*Постройте случайную перестановку чисел от 1 до N так, чтобы первое число обязательно было равно 5.
6. Заполните массив случайными числами в диапазоне 20..100 и подсчитайте отдельно число чётных и нечётных элементов.
7. Заполните массив случайными числами в диапазоне 1000.2000 и подсчитайте число элементов, у которых вторая с конца цифра – чётная.
8. Заполните массив случайными числами в диапазоне 0..100 и подсчитайте отдельно среднее значение всех элементов, которые <50, и среднее значение всех элементов, которые ≥50.

## § 63. Алгоритмы обработки массивов

### Поиск в массиве

Требуется найти в массиве элемент, равный значению переменной **X**, или сообщить, что его там нет. Алгоритм решения сводится к просмотру всех элементов массива с первого до последнего. Как только найден элемент, равный **X**, нужно выйти из цикла и вывести результат. Напрашивается такой алгоритм:

```
i = 0;
while ( A[i] != X )
    i ++;
printf ( "A[%d]=%d", i, X );
```

Он хорошо работает, если нужный элемент в массиве есть, однако приведет к ошибке, если такого элемента нет – получится заикливание и выход за границы массива. Поэтому в условие нужно добавить еще одно ограничение: **i < N**. Если после окончания цикла это условие нарушено, значит поиск был неудачным – элемента нет:

```
i = 0;
while ( i < N && A[i] != X )
    i ++;
if ( i < N )
    printf ( "A[%d]=%d", i, X );
else
    printf ( "Не нашли!" );
```

Отметим одну тонкость. В сложном условии **i < N** и **A[i] != X** первым должно проверяться именно отношение **i < N**. Если первая часть условия, соединенного с помощью операции «И», ложно, то вторая часть, как правило<sup>10</sup>, не вычисляется – уже понятно, что всё условие ложно. Заметим, что если **i ≥ N**, при проверке условия **A[i] != X** происходит *выход за границы массива*, то есть к обращению к ячейке памяти за пределами массива.

Возможен ещё один поход к решению этой задачи: используя цикл с переменной, перебрать все элементы массива и досрочно завершить цикл, если найдено требуемое значение.

```
nX = -1;
for ( i = 0; i < N; i++ )
    if ( A[i] == X )
    {
        nX = i;
        break;
    }
if ( nX >= 0 )
    printf ( "A[%d]=%d", nX, X );
else
```

<sup>10</sup> Во многих современных языках программирования (например, в C, C++, Python, Javascript, PHP) такое поведение гарантировано стандартом.

```
printf ( "Не нашли!" );
```

Для хранения номера найденного элемента вводим переменную **nX**, сначала записываем в неё значение  $-1$  (или любое другое число вне диапазона от 0 до  $N-1$ , то есть неправильный номер элемента). Для выхода из цикла используется оператор **break**. Если значение переменной **nX** осталось равным  $-1$  (не изменилось в ходе выполнения цикла), то в массиве нет элемента, равного **X**.

## Максимальный элемент

Найдем в массиве максимальный элемент. Для его хранения выделим целочисленную переменную **M**. Будем в цикле просматривать все элементы массива один за другим. Если очередной элемент массива больше, чем максимальный из предыдущих (находящийся в переменной **M**), запомним новое значение максимального элемента в **M**.

Остается решить, каково должно быть начальное значение **M**. Во-первых, можно записать туда значение, заведомо меньшее, чем любой из элементов массива. Например, если в массиве записаны натуральные числа, можно записать в **M** ноль или отрицательное число. Если содержимое массива неизвестно, можно сразу записать в **M** значение **A[0]**, а цикл перебора начать с **A[1]**, то есть со второго по счёту элемента:

```
M = A[0];
for ( i = 1; i < N; i++ )
    if ( A[i] > M ) M = A[i];
printf ( "%d", M );
```

Теперь предположим, что нужно найти не только значение, но и номер максимального элемента. Казалось бы, нужно ввести ещё одну переменную **nMax** для хранения номера, сначала записать в неё 0 (считаем элемент **A[0]** максимальным) и затем, когда нашли новый максимальный элемент, запоминать его номер в переменной **nMax**:

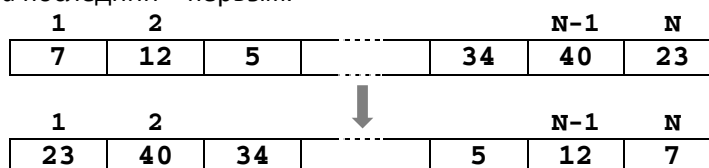
```
M = A[0]; nMax = 0;
for ( i = 1; i < N; i++ )
    if ( A[i] > M ) {
        M = A[i];
        nMax = i;
    }
printf ( "A[%d]=%d", nMax, M );
```

Однако это не самый лучший вариант. Дело в том, что по номеру элемента можно всегда определить его значение. Поэтому достаточно хранить только номер максимального элемента. Если этот номер равен **nMax**, то значение максимального элемента равно **A[nMax]**:

```
nMax = 0;
for ( i = 1; i < N; i++ )
    if ( A[i] > A[nMax] )
        nMax = i;
printf ( "A[%d]=%d", nMax, A[nMax] );
```

## Реверс массива

Реверс массива – это перестановка его элементов в обратном порядке: первый элемент становится последним, а последний – первым.



Из рисунка следует, что 0-й элемент меняется местами с  $(N-1)$ -м, 1-й – с  $(N-2)$ -м и т.д. Сумма индексов элементов, участвующих в обмене, для всех пар равна  $N-1$ , поэтому элемент с номером **i** должен меняться местами с  $(N-1-i)$ -м элементом. Кажется, что можно написать такой цикл:

```
сделать для i от 0 до N-1
    поменять местами A[i] и A[N-1-i]
```

однако это неверно. Посмотрим, что получится для массива из четырёх элементов:

	0	1	2	3
	7	12	40	23
$A[0] \leftrightarrow A[3]$	23	12	40	7
$A[1] \leftrightarrow A[2]$	23	40	12	7
$A[2] \leftrightarrow A[1]$	23	12	40	7
$A[3] \leftrightarrow A[0]$	7	12	40	23

Как видите, массив вернулся в исходное состояние: реверс выполнен дважды. Поэтому нужно остановить цикл на середине массива:

сделать для  $i$  от 0 до  $N/2$   
поменять местами  $A[i]$  и  $A[N-1-i]$

Для обмена используется вспомогательная целая переменная  $c$ :

```
for ( i = 0; i < (N/2); i++ )
{
    c = A[i];
    A[i] = A[N-1-i];
    A[N-1-i] = c;
}
```

## Сдвиг элементов массива

При удалении и вставке элементов необходимо выполнять сдвиг части или всех элементов массива в ту или другую сторону. Массив часто рисуют в виде таблицы, где первый элемент расположен слева. Поэтому сдвиг влево – это перемещение всех элементов на одну ячейку, при котором  $A[1]$  переходит на место  $A[0]$ ,  $A[2]$  – на место  $A[1]$  и т.д.

0	1				N-2	N-1
7	12	5		34	40	23

↓

0	1				N-2	N-1
12	5			34	40	23

Последний элемент остается на своем месте, поскольку новое значение для него взять неоткуда – массив кончился. Алгоритм выглядит так:

```
for ( i = 0; i < N-1; i++ )
    A[i] = A[i+1];
```

Обратите внимание, что цикл заканчивается при  $i=N-2$  (а не  $N-1$ ), чтобы не было выхода за границы массива, то есть обращения к несуществующему элементу  $A[N]$ .

При таком сдвиге первый элемент пропадает, а последний – дублируется. Можно старое значение первого элемента записать на место последнего. Такой сдвиг называется *циклическим* (см. § 28). Предварительно (до начала цикла) первый элемент нужно запомнить во вспомогательной переменной, а после завершения цикла записать его в последнюю ячейку массива:

```
c = A[0];
for ( i = 0; i < N-1; i++ )
    A[i] = A[i+1];
A[N-1] = c;
```

## Отбор нужных элементов

Требуется отобрать все элементы массива  $A$ , удовлетворяющие некоторому условию, в массив  $B$ . «Очевидное» решение:

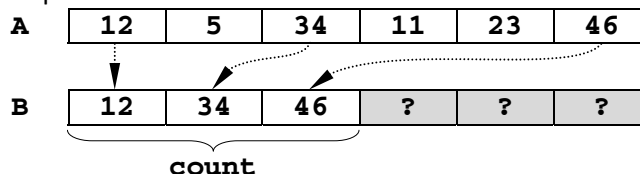
```
для i от 0 до N-1
    если условие выполняется для A[i] то
        B[i] = A[i];
```

На самом деле это решение плохое, потому что нужные элементы в массиве  $B$  оказываются расположены вразброс, на тех местах, где они стояли в массиве  $A$ . Поэтому работать с таким массивом  $B$  очень неудобно. На рисунке изображен случай, когда отбираются чётные элементы:





Хочется, чтобы все отобранные элементы стояли в начале массива **B**:



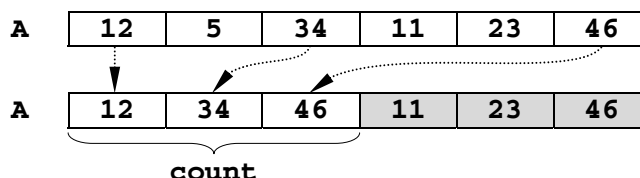
Для этого вводят переменную-счётчик **count**, в которой считают количество найденных элементов. Сначала её значение равно нулю. Когда очередной элемент найден, счётчик содержит номер первой свободной ячейки массива **B**, в неё записывается найденное значение, а затем значение счётчика увеличивается на 1. Программа, отбирающая все чётные элементы, выглядит так:

```
count = 0;
for ( i = 0; i < N; i++ )
    if ( A[i] % 2 == 0 )
    {
        B[count] = A[i];
        count++;
    }
```

Нужно помнить, что только первые **count** элементов массива **B** рабочие, остальные содержат неизвестные данные. Например, вот так можно вывести найденные элементы на экран:

```
for ( i = 0; i < count; i++ )
    printf ( "%d ", B[i] );
```

Если вместо массива **B** использовать тот же массив **A**, где находятся исходные числа, то все «нужные» элементы будут сгруппированы в начале, а их количество записано в переменной **count**:



## Контрольные вопросы

1. Почему при поиске индекса максимального элемента не нужно хранить само значение максимального элемента?
2. Что такое реверс массива?
3. Как вы думаете, какую ошибку чаще всего делают начинающие, программируя реверс массива?
4. Как вы думаете, какие проблемы (и ошибки) могут возникнуть при циклическом сдвиге массива *вправо*?
5. Что произойдет с массивом при выполнении следующего фрагмента программы:
 

```
for ( i = 0; i < N-1; i++ )
    A[i+1] = A[i];
```
6. Как (при использовании приведенного алгоритма поиска) определить, что элемент не найден?
7. Что такое выход за границы массива? Почему он может быть опасен?
8. Опишите «очевидный» алгоритм отбора части элементов одного массива в другой массив. Почему его не используют?



## Задачи и задания

1. Напишите программу, которая находит максимальный и минимальный из чётных положительных элементов массива. Если в массиве нет чётных положительных элементов, нужно вывести сообщение об этом.
2. Введите массив с клавиатуры и найдите (за один проход) количество элементов, имеющих максимальное значение.
3. Найдите за один проход по массиву три его различных элемента, которые меньше всех остальных («три минимума»).
4. \*Заполните массив случайными числами в диапазоне 10..12 и найдите длину самой длинной последовательности стоящих рядом одинаковых элементов.
5. Заполните массив случайными числами в диапазоне 0..4 и выведите на экран номера всех элементов, равных значению **X** (оно вводится с клавиатуры).
6. Заполните массив случайными числами и переставьте соседние элементы, поменяв 1-ый элемент со 2-м, 3-й – с 4-м и т.д.
7. Заполните массив из чётного количества элементов случайными числами и выполните реверс отдельно для 1-ой и 2-ой половин массива.
8. Заполните массив случайными числами и выполните реверс для части массива между элементами с индексами **K** и **M** (включая эти элементы).
9. Напишите программу для выполнения циклического сдвига массива вправо на 4 элемента.
10. Найдите в массиве все простые числа и скопируйте их в новый массив.
11. \*Найдите в массиве все числа Фибоначчи и скопируйте их в новый массив.

## § 64. Сортировка

### Введение

**Сортировка** – это расстановка элементов массива в заданном порядке.

Порядок сортировки может быть любым; для чисел обычно рассматривают сортировку по возрастанию (или убыванию) значений.

Возникает естественный вопрос: «зачем сортировать данные?». На него легко ответить, вспомнив, например, работу со словарями: сортировка слов по алфавиту облегчает поиск нужной информации.

Программисты изобрели множество способов сортировки. В целом их можно разделить на две группы: 1) простые, но медленно работающие (на больших массивах) и 2) сложные, но быстрые. Мы изучим два классических метода из первой группы и один метод из второй – знаменитую «быструю сортировку», предложенную Ч. Хоаром.

Далее мы будем, как принято в учебниках, рассматривать алгоритмы сортировки элементов массива по возрастанию (или убыванию) значений. Для массивов, в которых есть одинаковые элементы, используются понятия «сортировка по неубыванию» и «сортировка по невозрастанию».

### Метод пузырька (сортировка обменами)

Название этого метода произошло от известного физического явления – пузырёк воздуха в воде поднимается вверх. Если говорить о сортировке массива, сначала поднимается «наверх» (к началу массива) самый «лёгкий» (минимальный) элемент, затем следующий и т.д.

Сначала сравниваем последний элемент с предпоследним. Если они стоят неправильно (меньший элемент «ниже»), то меняем их местами. Далее так же рассматриваем следующую пару элементов и т.д. (см. рисунок).

4	4	4	4	1
5	5	5	1	4
2	2	1	5	5
1	1	2	2	2
3	3	3	3	3

Когда мы обработали пару ( $A[0]$ ,  $A[1]$ ), минимальный элемент стоит на месте  $A[0]$ . Это значит, что на следующих этапах его можно не рассматривать. Первый цикл, устанавливающий на свое место первый (минимальный) элемент, можно на псевдокоде записать так:

```

для  $j$  от  $N-2$  до  $0$  шаг  $-1$ 
    если  $A[j+1] < A[j]$  то
        поменять местами  $A[j]$  и  $A[j+1]$ 

```

Здесь  $j$  – целочисленная переменная. Обратите внимание, что на очередном шаге сравниваются элементы  $A[j]$  и  $A[j+1]$ , поэтому цикл начинается с  $j=N-2$ . Если начать с  $j=N-1$ , то на первом же шаге получаем выход за границы массива – обращение к элементу  $A[N]$ .

За один проход такой цикл ставит на место один элемент. Чтобы «подтянуть» второй элемент, нужно написать еще один почти такой же цикл, который будет отличаться только конечным значением  $j$  в заголовке цикла. Так как верхний элемент уже стоит на месте, его не нужно трогать:

```

сделать для  $j$  от  $N-2$  до  $1$  шаг  $-1$ 
    если  $A[j+1] < A[j]$  то
        поменять местами  $A[j]$  и  $A[j+1]$ 

```

При установке 3-го элемента конечное значение для  $j$  будет равно 2 и т.д.

Таких циклов нужно сделать  $N-1$  – на 1 меньше, чем количество элементов массива. Почему не  $N$ ? Дело в том, что если  $N-1$  элементов поставлены на свои места, то оставшийся автоматически встает на своё место – другого места для него нет. Поэтому полный алгоритм сортировки представляет собой такой вложенный цикл:

```

for (  $i = 0$ ;  $i < N-1$ ;  $i++$  )
    for (  $j = N-2$ ;  $j \geq i$ ;  $j--$  )
        if (  $A[j] > A[j+1]$  )
        {
            // поменять местами  $A[j]$  и  $A[j+1]$ 
        }

```

Записать полную программу вы можете самостоятельно.

## Метод выбора

Еще один популярный простой метод сортировки – метод выбора, при котором на каждом этапе выбирается минимальный элемент (из оставшихся) и ставится на свое место. Алгоритм в общем виде можно записать так:

```

для  $i$  от  $1$  до  $N-1$ 
    найти номер  $nMin$  минимального элемента из  $A[i]..A[N]$ 
    если  $i \neq nMin$  то
        поменять местами  $A[i]$  и  $A[nMin]$ 

```

Здесь перестановка происходит только тогда, когда найденный минимальный элемент стоит не на своём месте, то есть  $i \neq nMin$ . Поскольку поиск минимального элемента выполняется в цикле, этот алгоритм сортировки также представляет собой вложенный цикл:

```

for (  $i = 0$ ;  $i < N-1$ ;  $i++$  ) {
     $nMin = i$ ;
    for (  $j = i+1$ ;  $j < N$ ;  $j++$  )
        if (  $A[j] < A[nMin]$  )
             $nMin = j$ ;
    if (  $i \neq nMin$  )
    {
        // поменять местами  $A[i]$  и  $A[nMin]$ 
    }
}


```

## «Быстрая сортировка»



Ч.Э. Хоар (р. 1934)  
([en.wikipedia.org](http://en.wikipedia.org))

Методы сортировки, описанные в предыдущем параграфе, работают медленно для больших массивов данных (более 1000 элементов). Поэтому в середине XX века математики и программисты серьезно занимались разработкой более эффективных алгоритмов сортировки. Один из самых популярных «быстрых» алгоритмов, разработанный в 1960 году английским учёным Ч. Хоаром, так и называется – «быстрая сортировка» (англ. *quicksort*).



Будем исходить из того, что сначала лучше делать перестановки элементов массива на большом расстоянии. Предположим, что у нас есть  $N$  элементов и известно, что они уже отсортированы в обратном порядке. Тогда за  $N/2$  обменов можно отсортировать их как нужно – сначала поменять местами первый и последний, а затем последовательно двигаться с двух сторон к центру. Хотя это справедливо только тогда, когда порядок элементов обратный, подобная идея положена в основу алгоритма *Quicksort*.

Пусть дан массив **A** из **N** элементов. Выберем сначала наугад любой элемент массива (назовем его **X**). На первом этапе мы расставим элементы так, что слева от некоторой границы (в первой группе) находятся все числа, меньшие или равные **X**, а справа (во второй группе) – большие или равные **X**. Заметим, что элементы, равные **X**, могут находиться в обеих частях.

<b>A[i] ≤ X</b>	<b>A[i] ≥ X</b>
-----------------	-----------------

Теперь элементы расположены так, что ни один элемент из первой группы при сортировке не окажется во второй и наоборот. Поэтому далее достаточно отсортировать *отдельно* каждую часть массива. Такой подход называют «разделяй и властвуй» (англ. *divide and conquer*).

Лучше всего выбирать  $\mathbf{x}$  так, чтобы в обеих частях было равное количество элементов. Такое значение  $\mathbf{x}$  называется медианой массива. Однако для того, чтобы найти медиану, надо сначала отсортировать массив<sup>11</sup>, то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому обычно в качестве  $\mathbf{x}$  выбирают средний элемент массива или элемент со случайным номером.

Сначала будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше **X** (и, следовательно, должен стоять справа от **X**). Затем просматриваем массив справа до тех пор, пока не обнаружим элемент меньше **X** (он должен стоять слева от **X**). Теперь поменяем местами эти два элемента и продолжим просмотр до тех пор, пока два «просмотра» не встретятся где-то в середине массива. В результате массив окажется разбитым на 2 части: левую со значениями меньшими или равными **X**, и правую со значениями большими или равными **X**. На этом первый этап («разделение») закончен. Затем такая же процедура применяется к обеим частям массива до тех пор, пока в каждой части не останется один элемент (и таким образом, массив будет отсортирован).

Чтобы понять сущность метода, рассмотрим пример. Пусть задан массив

78	6	82	67	55	44	34
			↑ x			

Выберем в качестве **X** средний элемент массива, равный 67. Найдем первый слева элемент массива, который больше или равен **X** и должен стоять во второй части. Это число 78. Обозначим индекс этого элемента через **L**. Теперь находим самый правый элемент, который меньше **X** и должен стоять в первой части. Это число 34. Обозначим его индекс через **R**.

78	6	82	67	55	44	34
↑ L						↑ R

Теперь поменяем местами два этих элемента. Сдвигая переменную **L** вправо, а **R** – влево, находим следующую пару, которую надо переставить. Это числа 82 и 44.

<sup>11</sup> Хотя есть и другие, тоже достаточно непростые алгоритмы.

34	6	82	67	55	44	78
		↑ L			↑ R	

Следующая пара элементов для перестановки – числа 67 и 55.

34	6	44	67	55	82	78
			↑ L	↑ R		

После этой перестановки дальнейший поиск приводит к тому, что переменная **L** становится больше **R**, то есть массив разбит на две части. В результате все элементы массива, расположенные левее **A[L]**, меньше или равны **X**, а все правее **A[R]** – больше или равны **X**.

34	6	44	55	67	82	78
			↑ R	↑ L		

Таким образом, сортировка исходного массива свелась к двум сортировкам частей массива, то есть к двум задачам того же типа, но меньшего размера. Теперь нужно применить тот же алгоритм к двум полученным частям массива: первая часть – с 1-ого до **R**-ого элемента, вторая часть – с **L**-ого до последнего элемента. Как вы знаете, такой прием называется *рекурсией*.

Сначала предположим, что в нашей программе один *глобальный* массив **A** индексами от 0 до **N-1**, который нужно сортировать. Это значит, что массив доступен всем процедурам и функциям, в том числе и нашей процедуре сортировки. Глобальные данные объявляются выше основной программы:

```
const int N = 7;
int A[N];
```

Тогда процедура сортировки принимает только два параметра, ограничивающие её «рабочую зону»: **nStart** – номер первого элемента, и **nEnd** – номер последнего элемента. Если **nStart = nEnd**, то в «рабочей зоне» один элемент и сортировка не требуется, то есть нужно выйти из процедуры. В этом случае рекурсивные вызовы заканчиваются.

Приведем полностью процедуру быстрой сортировки:

```
void qSort( int nStart, int nEnd )
{
    int L, R, c, X;
    if ( nStart >= nEnd ) return; // массив отсортирован
    L = nStart; R = nEnd;
    X = A[(L+R)/2];
    while ( L <= R ) { // разделение
        while ( A[L] < X ) L++;
        while ( A[R] > X ) R--;
        if ( L <= R ) {
            c = A[L]; A[L] = A[R]; A[R] = c;
            L++; R--;
        }
    }
    qSort ( nStart, R ); // рекурсивные вызовы
    qSort ( L, nEnd );
}
```

Для того, чтобы отсортировать весь массив, нужно вызвать эту процедуру так:

```
qSort ( 0, N-1 );
```

К сожалению, эта процедура сортирует только глобальный массив **A**, чтобы использовать её для сортировки других массивов, добавим в заголовок еще один параметр – целочисленный массив:

```
void qSort( int A[], int nStart, int nEnd )
{
    ...
}
```

Размер массива указывать не обязательно, такая процедура будет работать с массивами любого размера. Изменяются также и рекурсивные вызовы (на первом месте в списке аргументов нужно указать имя массива):

```
void qSort( int A[], int nStart, int nEnd )
{
    ...
    qSort ( A, nStart, R );    // рекурсивные вызовы
    qSort ( A, L, nEnd );
}
```

а также вызов процедуры из основной программы:

```
qSort ( A, 0, N-1 );
```

Скорость работы быстрой сортировки зависит от того, насколько удачно выбирается вспомогательный элемент **X**. Самый лучший случай – когда на каждом этапе массив делится на две равные части. Худший случай – когда в одной части оказывается только один элемент, а в другой – все остальные. При этом глубина рекурсии достигает **N**, что может привести к переполнению стека (нехватке стековой памяти).

Для того, чтобы уменьшить вероятность худшего случая, в алгоритм вводят случайность: в качестве **X** на каждом шаге выбирают не середину рабочей части массива, а элемент со случайным номером:

```
X = A[irand(L,R)];
```

Здесь используется написанная нами ранее (см. § 62) функция **irand**, которая возвращает случайное целое число на отрезке **[L,R]**.

В таблице сравнивается время сортировки (в секундах) массивов разного размера, заполненных случайными значениями, с использованием трёх изученных алгоритмов.

N	метод пузырька	метод выбора	быстрая сортировка
1000	0,24 с	0,12 с	0,004 с
5000	5,3 с	2,9 с	0,024 с
15000	45 с	34 с	0,068 с

Как показывают эти данные, преимущество быстрой сортировки становится подавляющим при увеличении **N**.



## Контрольные вопросы

1. Что такое сортировка?
2. На какой идее основан метод пузырька? метод выбора?
3. Объясните, зачем нужен вложенный цикл в описанных методах сортировки.
4. Сравните метод пузырька и метод выбора. Какой из них требует меньше перестановок?
5. Расскажите про основные идеи метода «быстрой сортировки».
6. Как вы думаете, можно ли использовать метод «быстрой сортировки» для нечисловых данных, например, для символьных строк?
7. От чего зависит скорость «быстрой сортировки»? Какой самый лучший и самый худший случай?
8. Как вы думаете, может ли метод «быстрой сортировки» работать дольше, чем метод выбора (или другой «простой» метод)? Если да, то при каких условиях?
9. Как нужно изменить приведенные алгоритмы, чтобы элементы массива были отсортированы по убыванию?



## Задачи и задания

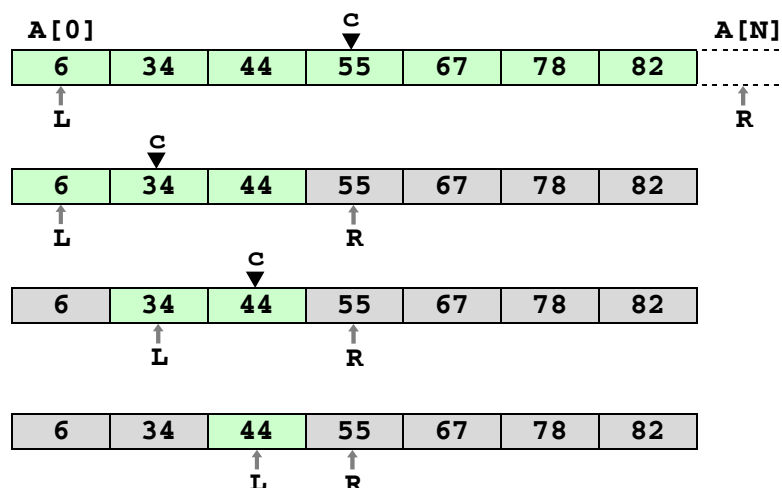
1. Отсортировать массив и найти количество различных чисел в нём.
2. Напишите программу, в которой сортировка выполняется «методом камня» – самый тяжёлый элемент опускается в конец массива.

3. Напишите программу, которая выполняет неполную сортировку массива: ставит в начало массива три самых меньших по величине элемента в порядке возрастания (неубывания). Положение остальных элементов не важно.
4. Напишите вариант метода пузырька, который заканчивает работу, если на очередном шаге внешнего цикла не было перестановок.
5. Напишите программу, которая сортирует массив по возрастанию последней цифры числа.
6. Напишите программу, которая сортирует массив по убыванию суммы цифр числа.
7. Напишите программу, которая сортирует первую половину массива по возрастанию, а вторую – по убыванию (элементы из первой половины не должны попадать во вторую и наоборот).
8. Напишите программу, которая сортирует массив, а затем находит максимальное из чисел, встречающихся в массиве несколько раз.
9. \*Напишите программу, которая сравнивает количество перестановок при сортировке одного и того же массива разными методами. Проведите эксперименты для возрастающей последовательности (уже отсортированной), убывающей (отсортированной в обратном порядке) и случайной.

## § 65. Двоичный поиск

Ранее мы уже рассматривали задачу поиска элемента в массиве и привели алгоритм, который сводится к просмотру всех элементов массива. Такой поиск называют *линейным*. Для массива из 1000 элементов нужно сделать 1000 сравнений, чтобы убедиться, что заданного элемента в массиве нет. Если число элементов (например, записей в базе данных) очень велико, время поиска может оказаться недопустимым, потому что пользователь не дожидется ответа.

Как вы помните, основная задача сортировки – облегчить последующий поиск данных. Вспомним, как мы ищем нужное слово (например, слово «гравицапа») в словаре. Сначала открываем словарь примерно в середине и смотрим, какие там слова. Если они начинаются на букву «Л», то слово «гравицапа» явно находится на предыдущих страницах, и вторую часть словаря можно не смотреть. Теперь так же проверяем страницу в середине первой половины, и т.д. Такой поиск называется *двоичным*. Понятно, что он возможен только тогда, когда данные предварительно отсортированы. Для примера на рисунке показан поиск числа  $X = 44$  в отсортированном массиве:



Серым фоном выделены ячейки, которые уже не рассматриваются, потому что в них не может быть заданного числа. Переменные  $L$  и  $R$  ограничивают «рабочую зону» массива:  $L$  содержит номер первого элемента, а  $R$  – номер элемента, *следующего* за последним. Таким образом, нужный элемент (если он есть) находится в части массива, которая начинается с элемента  $A[L]$  и заканчивается элементом  $A[R-1]$ .

На каждом шаге вычисляется номер среднего элемента «рабочей зоны», он записывается в переменную  $c$ . Если  $X < A[c]$ , то этот элемент может находиться только левее  $A[c]$ , и правая



граница **R** перемещается в **c**. В противном случае нужный элемент находится правее середины или совпадает с **A[c]**; при этом левая граница **L** перемещается в **c**.

Поиск заканчивается при выполнении условия **L = R - 1**, когда в рабочей зоне остался один элемент. Если при этом **A[L] = X**, то в результате найден элемент, равный **X**, иначе такого элемента нет.

```
int X, L, R, c;
L = 0; R = N;
while ( L < R-1 )
{
    c = (L+R) / 2;
    if ( X < A[c] )
        R = c;
    else L = c;
}
if ( A[L] == X )
    printf ( "A[%d]=%d", L, X );
else
    printf ( "Не нашли!" );
```

Двоичный поиск работает значительно быстрее, чем линейный. В нашем примере (для массива из 8 элементов) удастся решить задачу за 3 шага вместо 8 при линейном поиске. При увеличении размера массива эта разница становится впечатляющей. В таблице сравнивается количество шагов для двух методов при разных значениях **N**:

N	линейный поиск	двоичный поиск
2	2	2
16	16	5
1024	1024	11
1048576	1048576	21

Однако при этом нельзя сказать, что двоичный поиск лучше линейного. Нужно помнить, что данные необходимо предварительно отсортировать, а это может занять значительно больше времени, чем сам поиск. Поэтому такой подход эффективен, если данные меняются (и сортируются) редко, а поиск выполняется часто. Такая ситуация характерна, например, для баз данных.



### Контрольные вопросы

1. Почему этот алгоритм поиска называется «двоичным»?
2. Приведите примеры использования двоичного поиска в обычной жизни.
3. Как можно примерно подсчитать количество шагов при двоичном поиске?
4. Сравните достоинства и недостатки линейного и двоичного поиска.



### Задачи и задания

1. Напишите программу, которая сортирует массив по убыванию и ищет в нем все значения, равные введенному числу.
2. Напишите программу, которая считает среднее число шагов при двоичном поиске для массива из 32 элементов в диапазоне 0..100. Для поиска используйте 1000 случайных чисел в этом же диапазоне.

## § 66. Символьные строки

### Что такое символьная строка?

Если в середине XX века первые компьютеры использовались, главным образом, для выполнения сложных математических расчётов, сейчас их основная работа – обработка текстовой (символьной) информации.

Символьная строка – это последовательность символов, расположенных в памяти рядом (в соседних ячейках). Для работы с символами во многих языках программирования есть перемен-

ные специального типа: символы и символьные массивы. Казалось бы, массив символов – это и есть символьная строка, однако здесь есть некоторые особенности.

Дело в том, что массив – это группа символов, каждый из которых независим от других. Это значит, что вводить символьный массив нужно *посимвольно*, в цикле. Более того, размер массива задается при объявлении, и не очень ясно, как использовать массивы для работы со строками переменной длины. Таким образом, нужно

- работать с целой символьной строкой как с единым объектом;
- использовать строки переменной длины.

В языках С и С++ работа с символьными строками происходит по-разному. Сначала мы рассмотрим язык С (заметим, что все эти возможности работают и в С++).

В языке С строка объявляется как массив символов, то есть массив элементов типа **char** (от англ. *character* – символ):

```
char s[10];
```

Чем же отличается строка от массива? Только тем, что внутри этого массива есть символ с кодом 0, который обозначается как `'\0'`. Этот невидимый символ (для него нет никакого изображения) обозначает окончание символьной строки. Не нужно путать его с символом `'0'` (цифрой 0), который имеет код 48. Таким образом, в символьный массив из 10 символов можно записать строку длиной 9 символов: ещё один символ нужно оставить на признак окончания строки.

Рассмотрим такой массив:

	0	1	2	3	4	5	6	7	8	9
s	П	р	и	в	е	т	!	\0	?	?

Символ `'\0'` в элементе массива `s[7]` говорит о том, что здесь заканчивается символьная строка, состоящая в данном случае из 7 символов.

Для вывода строки на экран можно использовать функцию **printf** с форматом `%s` (от англ. *string* – строка):

```
printf( "%s", s );
```

Для массива, показанного выше, будет выведено

**Привет!**

Символы с индексами 8 и 9 хранятся в памяти, но не выводятся на экран, потому что при выводе в элементе `s[7]` встретился символ с кодом 0, означающий конец строки. Для вывода одной строки удобно использовать функцию **puts**:

```
puts( s );
```

Она выводит строку на экран с переходом на новую строку, то есть автоматически добавляет символ `'\n'`.

Строку можно записать в массив при объявлении:

```
char s[10] = "Привет!";
```

Можно даже не указывать размер массива – в этом случае транслятор вычислит его автоматически:

```
char s[] = "Привет!";
```

Размер этого массива будет равен 8 (7 символов строки + символ с кодом 0), записать в него строку длиннее 8 символов нельзя (в этом случае будет *выход за границы массива*, который приведет к изменению ячеек памяти, не принадлежащих этому массиву).

Вводить строку с клавиатуры тоже можно по формату `%s`:

```
scanf( "%s", s );
```

Обратите внимание, что знак `&` перед именем строки не ставится (в отличие от ввода целых переменных), потому что имя строки в языке С воспринимается как адрес её первого символа (символа с индексом 0). Если вводить таким образом строку, содержащую пробелы, вы увидите, что вводится только первое слово (символы до первого пробела), так работает функция **scanf**. Часто бывает нужно ввести строку с пробелами целиком, для этого удобно использовать функцию **gets**:

```
gets( s );
```

Длина строки (количество символов от начала строки до завершающего символа с кодом 0) определяется с помощью стандартной функции **strlen** (от англ. *string length* – длина строки). Вот так в целочисленную переменную **n** записывается длина строки **s**:

```
n = strlen ( s );
```

Чтобы использовать эту функцию (и другие функции для работы с символьными строками), нужно подключить заголовочный файл **string.h**:

```
#include <string.h>
```

Для того, чтобы работать с отдельными символами строки, к ним нужно обращаться так же, как к элементам массива: в квадратных скобках записывают индекс символа. Например, так можно изменить четвёртый символ на 'a' (при условии, что длина строки не менее 5 символов) :

```
s[4] = 'a';
```

Приведём полную программу, которая вводит строку с клавиатуры, заменяет в ней все буквы 'a' на буквы 'б' и выводит полученную строку на экран.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s[80];
    int i;
    printf ( "Введите строку" );
    gets ( s );
    for ( i = 0; i < strlen(s); i++ )
        if ( s[i] == 'a' )
            s[i] = 'б';
    puts ( s );
}
```

Теперь покажем, как сделать то же самое, используя возможности языка C++. Для работы с символьными строками в C++ введён специальный тип данных, который называется **string**:

```
main()
{
    string s;
    ...
}
```

Начальное значение строки можно задать прямо при объявлении:

```
string s = "Привет!";
```

Новое значение строки записывается с помощью оператора присваивания:

```
s = "Привет!";
```

Для того, чтобы ввести из входного потока строку с пробелами, применяется функция **getline**:

```
getline ( cin, s );
```

а вывод выполняется стандартным образом:

```
cout << s;
```

Для определения длины строки **s** используется запись **s.size()**. Это так называемая точечная нотация (подробнее мы познакомимся с ней в курсе 11 класса). Такая запись означает, что метод **size** применяется к объекту **s** типа **string**. В данном случае **size** – это функция (метод), связанная с типом данных **string**.

Таким образом, программа которая заменяет в строке все буквы 'a' на буквы 'б' на языке C++ выглядит так:

```
#include <iostream>
using namespace std;
main()
{
    string s;
    int i;
    cout << "Введите строку: ";
    getline ( cin, s );
```

```

for ( i = 0; i < s.size(); i++ )
    if ( s[i] == 'a' )
        s[i] = 'б';
cout << s;
}

```

## Операции со строками (язык C)

Для выполнения стандартных операций со строками (копирования, сцепления, удаления и вставки символов) применяют функции стандартной библиотеки языка C, которые подключаются с помощью заголовочного файла **string.h**.

Для того, чтобы объединить две строки в одну, используется функция **strcat** (от англ. *string concatenation* – сцепление строк). Она дописывает вторую строку в конец первой:

```

char s[80] = "Привет", s1[] = "Вася!";
strcat ( s, " ", " );
strcat ( s, s1 );

```

В результате в массиве **s** будет построена строка «Привет, Вася!». Первый вызов функции **strcat** добавит в конец строки **s** запятую и пробел, а второй – допишет ещё строку «Вася!», которая записана в массиве **s1**.

Функция **strcpy** (от англ. *string copy* – копировать строку) копирует строку из одного места в другое. Например, после выполнения фрагмента программы:

```

char s[80], s1[] = "Привет!";
strcpy ( s1, "Вася" );
strcpy ( s, s1 );

```

в обоих массивах будет записана строка «Вася»: первый вызов функции **strcpy** запишет строку «Вася» в массив **s1**, а второй вызов – скопирует строку из **s1** в **s**. Обратите внимание, что при вызове этой функции сначала указывают куда скопировать, а потом – что скопировать.

Функция **strcpy** (как и другие функции для работы со строками) не проверяет, есть ли в массиве место для копирования строки. Если в приведённом примере размер массива **s** будет меньше, чем 5 символов, произойдет выход за границы массива. Это опасная ошибка, в результате которой стираются данные за пределами массива.

Функции **strcpy** на самом деле нужно передать два адреса в памяти – адрес строки-приёмника и адрес строки-источника данных. Как мы уже говорили, указав имя строки, мы сообщаем функции адрес начала этой строки, но это не обязательно. В следующем примере адрес приёмника (куда будут скопированы данные) – это адрес символа **s[7]**:

```

char s[80] = "Прошёл поезд.";
strcpy ( &s[7], "пароход." );

```

В результате в массиве **s** будет построена строка «Прошёл пароход.». Вместо «**&s[7]**» можно использовать равносильную запись «**s+7**». Адрес источника данных (второй параметр) также не обязательно совпадает с началом строки. Например:

```

char s[80] = "Прошёл поезд.",
      s1[] = "Привет, Вася.";
strcpy ( s+7, s1+8 );

```

В массиве **s** будет построена строка «Прошёл Вася.».

Функцию **strcpy** можно использовать для удаления символов из строки. Вот так удаляются 4 символа с **s[2]** по **s[5]**:

```

char s[] = "0123456789";
strcpy ( s+2, s+6 );

```

«Хвост» строки, начиная с символа **s[6]**, переносится ближе к началу, на место символа **s[2]**. В массиве **s** остаётся строка «016789».

В библиотеке языка C есть ещё одна функция для копирования строк, которая называется **strncpy** (в середине названия есть дополнительная буква **n**). Она отличается от **strcpy** тем, что

- копирует не всю строку, а указанное количество символов (третий параметр функции, целое число);
- не добавляет в конец строки завершающий символ с кодом 0.

С помощью этой функции можно заменить часть строки:

```
char s[] = "Мухтар, ко мне!",
      s1[] = "Цезарь живет у нас дома.";
strncpy ( s, s1, 6 );
```

В начало массива **s** будет скопировано 6 первых символов из массива **s1**, получится «Цезарь, ко мне!». Можно также использовать адреса символов в середине массивов:

```
char s[] = "Вчера Мурзик вернулся.",
      s1[] = "Кота зовут Васька.";
strncpy ( s+6, s1+11, 6 );
```

В массиве **s** получается строка «Вчера Васька вернулся.».

В помощью функции **strncpy** несложно получить подстроку (часть строки) в другом массиве. Например, так можно скопировать в массив **s1** четыре символа из строки **s** с номерами от 2 до 5:

```
char s[] = "0123456789", s1[20];
strncpy ( s1, s+2, 4 );
s1[4] = '\0';
```

Поскольку функция **strncpy** не добавляет в конец строки завершающий символ с кодом 0, его приходится ставить вручную. В массиве **s1** будет записана строка «2345».

К сожалению, в языке C непросто выполнить операцию вставки фрагмента в строку. Для этого нужно использовать вспомогательный массив (*буфер*). В этом фрагменте после имени «Иван» в строку **s** вставляется отчество «Васильевич», так что получается строка «Иван Васильевич меняет профессию.»:

```
char s[80] = "Иван меняет профессию.", s1[30];
strcpy ( s1, s+4 ); // s1 = "меняет профессию."
strcpy ( s+4, "Васильевич" ); // s = "Иван Васильевич"
strcat ( s, s1 ); // s = "Иван Васильевич меняет профессию."
```

При первом вызове функции **strcpy** в буфер **s1** копируется «хвост» строки. Затем к начальной части добавляется фрагмент-вставка, и потом в конец строки дописывается «хвост» из буфера.

## Операции со строками (язык C++)

В языке C++ операции со строками выполняются значительно проще благодаря введённому типу **string**.

Оператор **+** используется для объединения (сцепления) строк, эта операция иногда называется *конкатенация*. Например:

```
s1 = "Привет";
s2 = "Вася";
s = s1 + ", " + s2 + "!";
```

Здесь и далее считаем, что в программе объявлены строки **s**, **s1** и **s2**. В результате выполнения приведённой программы в строку **s** будет записано значение **"Привет, Вася!"**.

Для того, чтобы выделить часть строки (*подстроку*, англ. *substring*), применяется метод **substr**, который тоже вызывается с помощью точечной нотации. Этот метод принимает два параметра: номер начального символа и *количество символов*. Следующий фрагмент копирует в строку **s1** пять символов строки **s** (с 3-го по 7-й):

```
s = "0123456789";
s1 = s.substr ( 3, 5 );
cout << s1 << endl;
```

В строку **s1** будет записано значение **«34567»**. Если второй параметр при вызове **substr** не указан, метод возвращает все символы до конца строки. Например,

```
s = "0123456789";
s1 = s.substr ( 3 );
```

вернёт **«3456789»**.

Для удаления части строки нужно вызвать метод **erase**, указав номер начального символа и число удаляемых символов:

```
s = "0123456789";
```

```
s.erase ( 3, 6 );
```

В строке **s** остаётся значение «0129» (удаляются 6 символов, начиная с 3-го). Обратите внимание, то процедура **erase** изменяет строку.

При вставке символов методу **insert** передают вставляемый фрагмент и номер символа, с которого начинается вставка:

```
s = "0123456789";
s.insert ( 3, "ABC" );
```

Переменная **s** получит значение «012ABC3456789».

## Поиск в строках

В библиотеке языка C есть отдельные функции для поиска одного символа и подстроки в строке. Им нужно передать строку, в которой надо искать, и образец для поиска.

Эти функции возвращают адрес в памяти, который можно записать в специальную переменную-указатель. Указатели в языке C объявляются с помощью знака \*. Например, так объявляют указатель на символ, то есть переменную, в которой можно хранить адрес любого символа в памяти (вспомните, где мы раньше уже фактически использовали указатели?):

```
char *p;
```

В такую переменную можно записать результат функции **strchr** (от англ. *string* – строка и *character* – символ), которая ищет один символ в строке:

```
char s[] = "Здесь был Вася.";
char *p;
p = strchr ( s, 'c' );
```

Обратите внимание, что символ заключается в апострофы, а не в двойные кавычки. Если символ найден, его номер вычисляется как разность указателя **p** и адреса начала строки **s**. Если такого символа в строке нет, в переменную **p** будет записано специальное значение **NULL** (нулевой указатель):

```
if ( p != NULL )
    printf ( "Номер первого символа 'c': %d\n", p-s );
else
    printf ( "Символ не найден.\n" );
```

Если указанных символов несколько, функция **strchr** найдет первое вхождение символа в строку. Если нужно искать символ с конца строки, применяют функцию **strrchr** (добавляется буква **r** в середине, от англ. *reverse* – обратный).

Поиск подстроки в строке выполняет функция **strstr**, которая работает аналогично:

```
char s[] = "Здесь был Вася.";
char *p;
p = strstr ( s, "Вася" );
if ( p != NULL )
    printf ( "Слово 'Вася' начинается с s[%d]\n", p-s );
else
    printf ( "Слово не найдено.\n" );
```

В языке C++ для поиска символа и подстроки используется метод **find**. Эта функция возвращает номер найденного символа (номер первого символа подстроки) или **-1**, если найти нужный фрагмент не удалось.

```
string s = "Здесь был Вася.";
int n;
n = s.find ( 'c' );
if ( n >= 0 )
    cout << "Номер первого символа 'c': " << n << endl;
else cout << "Символ не найден.\n";
n = s.find ( "Вася" );
if ( n >= 0 )
    cout << "Слово 'Вася' начинается с s[" << n << "]\n";
else
    cout << "Слово не найдено.\n";
```



Для поиска с конца строки используют метод `rfind`:

```
n=s.rfind ( 'c' );
if ( n >= 0 )
    cout << "Номер последнего символа 'с': " << n << endl;
else
    cout << "Символ не найден.\n";
```

## Пример обработки строк

Предположим, что с клавиатуры вводится строка, содержащая имя, отчество и фамилию человека, например:

**Василий Алибабаевич Хрюндиков**

Каждые два слова разделены одним пробелом, в начале строки пробелов нет. В результате обработки должна получиться новая строка, содержащая фамилию и инициалы:

**Хрюндиков В.А.**

Возможный алгоритм решения этой задачи может быть на псевдокоде записан так:

```
ввести строку s
найти в строке s первый пробел
имя = всё, что слева от первого пробела
удалить из строки s имя с пробелом
найти в строке s первый пробел
отчество = всё, что слева от первого пробела
удалить из строки s отчество с пробелом // осталась фамилия
s = s + ' ' + имя[1] + '.' + отчество[1] + '.'
```

Мы последовательно выделяем из строки три элемента: имя, отчество и фамилию, используя тот факт, что они разделены одиночными пробелами. После того, как имя сохранено в отдельной переменной, в строке `s` остались только отчество и фамилия. После «изъятия» отчества остается только фамилия. Теперь нужно собрать строку-результат из частей: «сцепить» фамилию и первые буквы имени и отчества, поставив пробелы и точки между ними. Для выполнения всех операций будем использовать стандартные функции, описанные выше.

Приведем полные программы на языке C:

```
#include <stdio.h>
#include <string.h>
main()
{
    char s[80], name[] = " .", name2[] = " .";
    char *p;
    printf ( "Введите имя, отчество и фамилию: " );
    gets ( s );
    name[0] = s[0]; // первая буква имени
    p = strchr ( s, ' ' ); // найти пробел
    strcpy ( s, p+1 ); // стереть имя
    name2[0] = s[0]; // первая буква отчества
    p = strchr ( s, ' ' ); // найти пробел
    strcpy ( s, p+1 ); // осталась фамилия
    strcat ( s, " " ); // добавить пробел
    strcat ( s, name ); // прицепить имя
    strcat ( s, name2 ); // прицепить отчество
    puts ( s );
}
```

и на C++:

```
#include <iostream>
using namespace std;
main()
{
    string s, name, name2;
    int n;
```



```

cout << "Введите имя, отчество и фамилию: ";
getline( cin, s );
n = s.find( ' ' );
name = s.substr(0,n) + '.';    // первая буква имени
s = s.substr ( n+1 );
n = s.find( ' ' );
name2 = s.substr(0,n) + '.';   // первая буква отчества
s = s.substr ( n+1 );         // осталась фамилия
s = s + ' ' + name + name2;
cout << s;
}

```

Используя описания стандартных функций, самостоятельно разберитесь, как работают эти программы.

## Преобразования число↔строка

В практических задачах часто нужно преобразовать число, записанное в виде цепочки символов, в числовое значение, и наоборот.

Для преобразования символьной строки в число в библиотеке языка C есть функции **atoi** (преобразование строки в целое число) и **atof** (преобразование строки в вещественное число). Для их использования нужно подключить заголовочный файл **stdlib.h**:

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

Преобразуем строку «123», записанную в массив **s**, в целое число:

```

char s[] = "123";
int N;
N = atoi ( s );    // N = 123

```

Если строку не удастся преобразовать в целое число (например, если строка содержит нецифровой символ), работа функции останавливается на первом ошибочном символе. Например, при преобразовании строки «12x3» будет получено число 12.

Аналогичная функция **atof** преобразует строку в вещественное число:

```

char s[] = "123.456";
float X;
X = atof ( s );    // X = 123.456

```

Обратное преобразование (из числа в символьную строку) проще всего сделать с помощью функции **sprintf**, которая очень похожа на известную нам функцию **printf**, но не выводит результат на экран, а записывает его в символьную строку:

```

char s[80];
int N = 123;
float X = 123.456;
sprintf ( s, "%d", N );    // s = "123"
sprintf ( s, "%e", X );    // s = "1.234560E+002"
sprintf ( s, "%10.3f", X ); // s = " 123.456"

```

Первый аргумент функции **sprintf** – это символьная строка, в которую нужно записать результат. При использовании формата **%e** вещественные числа представлены в научном формате ("1.234560E+002" означает  $1,23456 \cdot 10^2$ ). В последней строке примера используется форматный вывод: запись «%10.3f» означает «вывести вещественное число с фиксированной точкой в 10 позициях с 3-мя знаками в дробной части».

В языке C++ для преобразования строки (типа **string**) в число можно использовать те же функции **atoi** и **atof**:

```

string s = "123";
int N;
double X;
N = atoi ( s.c_str() );    // N = 123
X = atof ( s.c_str() );    // X = 123.456

```

Поскольку исходная строка – это объект типа **string**, его сначала нужно преобразовать к типу данных, который принимают функции **atoi** и **atof**, то есть к символьной строке с нулевым за-

вершающим символом на конце. Это делает метод **c\_str**, введенный для объектов типа **string**. В стандарте C++11 вместо **atoi** и **atof** можно использовать функции **stoi** и **stof**, которые не требуют такого преобразования и работают с объектами типа **string**.

Для обратного преобразования (из числа в строку) в C++ используется та же идея, что и в C: перенаправить результат вывода не на экран, а в символьную строку. Для этого в C++ нужно использовать строковые потоки вывода, которые подключаются с помощью заголовочного файла **sstream**:

```
#include <sstream>
main
{
    ostringstream ss;
    ...
}
```

Здесь **ss** – это объект типа **ostringstream**, выходной поток, направленный в строку. Итак, сначала выводим число в такой поток, а затем из этого потока преобразуем в строку (типа **string**) с помощью метода **str**:

```
string s;
int N=123;
double X=123.456;
ss << N;
s=ss.str();           // s = "123"
ss.str("");           // очистка потока
ss.width(13);         // ширина поля
ss.precision(6);      // число знаков в дробной части
ss << scientific << X; // вывести в научном формате
s=ss.str();           // s = "1.234560E+002"
ss.str("");           // очистка потока
ss.width(10);         // ширина поля
ss.precision(3);      // число знаков в дробной части
ss << fixed << X;      // вывести с фиксированной точкой
s=ss.str();           // s = " 123.456"
```

## Строки в процедурах и функциях

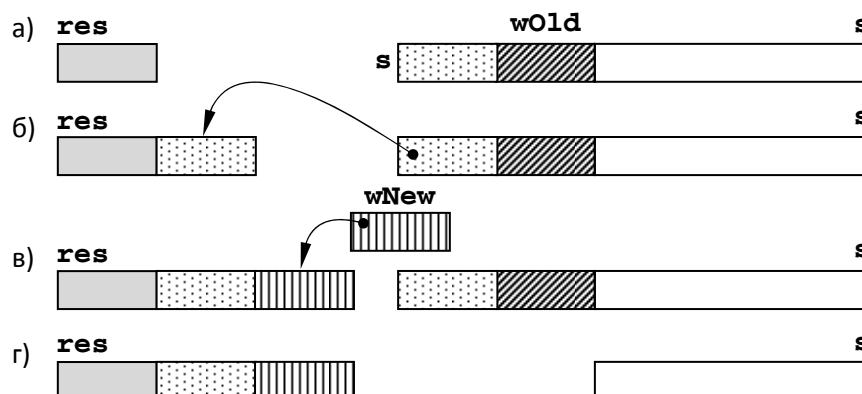
Строки можно передавать в процедуры и функции как аргументы. Построим процедуру, которая заменяет в строке **s** все вхождения слова-образца **wOld** на слово-замену **wNew** (здесь **wOld** и **wNew** – это имена переменных, а выражение «слово **wOld**» означает «слово, записанное в переменную **wOld**»).

Сначала разработаем алгоритм решения задачи. В первую очередь в голову приходит такой псевдокод

```
пока слово wOld есть в строке s
{
    удалить слово wOld из строки
    вставить на это место слово wNew
}
```

Однако такой алгоритм работает неверно, если слово **wOld** входит в состав **wNew**, например, нужно заменить '12' на 'A12B' (покажите самостоятельно, что это приведет к заикливанию).

Чтобы избежать подобных проблем, попробуем накапливать результат в другой символьной строке **res**, удаляя из строки **s** уже обработанную часть. Предположим, что на некотором шаге в оставшейся части строки **s** обнаружено слово **wOld** (рис. а).



Теперь нужно выполнить следующие действия:

- 1) ту часть строки **s**, которая стоит слева от образца, «прицепить» в конец строки **res** (рис. б);
- 2) «прицепить» в конец строки **res** слово-замену **wNew** (рис. в);
- 3) удалить из строки **s** начальную часть, включая найденное слово-образец (рис. г).

Далее все эти операции (начиная с поиска слова **wOld** в строке **s**) повторяются до тех пор, пока строка **s** не станет пустой. Если очередное слово найти не удалось, вся оставшаяся строка **s** приписывается в конец строки-результата, и цикл заканчивается.

В начале работы алгоритмы в строку **res** записывается пустая строка, не содержащая ни одного символа. В таблице приведен протокол работы алгоритма замены для строки «12.12.12», в которой нужно заменить слово «12» на «A12B»:

рабочая строка <b>s</b>	результат <b>res</b>
"12.12.12"	" "
".12.12"	"A12B"
".12"	"A12B.A12B"
" "	"A12B.A12B.A12B"

Теперь можно написать процедуру на языке С. Она принимает три символьные строки, размер которых мы не указываем:

```
void replaceAll ( char s[], char wOld[], char wNew[] )
{
    int lenOld, lenNew;
    char *p, *pS, *pRes;
    char res[200];
    lenOld = strlen(wOld);
    lenNew = strlen(wNew);
    res[0] = '\0';
    pS = s; pRes = res;
    while ( strlen(pS) > 0 )
    {
        p = strstr ( pS, wOld );
        if ( p == NULL )
        {
            strcat ( res, s );
            break;
        }
        if ( p > pS ) {
            strncpy ( pRes, pS, p-pS );
            pRes += p-pS;
            pS = p;
        }
        strcpy ( pRes, wNew );
        pRes += lenNew;
        pS += lenOld;
    }
    strcpy ( s, res );
}
```

Дадим некоторые пояснения. Ответ формируется в отдельной символьной строке **res** (это локальная переменная), и в конце процедуры копируется в строку **s**. В самом начале в первый по счёту символ массива **res** записывается символ с кодом 0 (строка **res** – пустая). Переменные **lenOld** и **lenNew** – это длины слова-образца и слова-замены соответственно. В процедуре используется также три указателя

- **p** – указывает на очередное найденное слово-образец;
- **pS** – указывает на рабочую точку внутри строки **s**, сначала устанавливается на начало этой строки;
- **pRes** – указывает на рабочую точку внутри строки-результата **res**; сначала устанавливается на начало этой строки.

Указатель **pS** будет сдвигаться вдоль строки **s**, обозначая начало зоны поиска. Цикл заканчивается тогда, когда в рабочей части строки **s** не осталось ни одного символа, то есть вызов функции **strlen(pS)** вернёт 0:

```
while ( strlen(pS) > 0 )
{
    ...
}
```

Кроме того, предусмотрен ещё один выход из цикла – если слово образец не найдено в оставшейся части строки, остаётся просто прицепить этот «хвост» к строке-результату выйти из цикла с помощью оператора **break**:

```
p = strstr ( pS, wOld );
if ( p == NULL )
{
    strcat ( res, s );
    break;
}
```

Если поиск завершился удачно, проверяем, есть ли какие-то символы слева от образца. Если они есть, копируем их в строку-результат:

```
if ( p > pS )
{
    strncpy ( pRes, pS, p-pS );
    pRes += p-pS;
    pS = p;
}
```

Здесь **p-pS** – это количество символов в строке **s** слева от найденного образца до рабочей точки.

Теперь остаётся прицепить строку-замену **wNew** к результату и передвинуть оба указателя:

```
strcpy ( pRes, wNew );
pRes += lenNew;
pS += lenOld;
```

После этого начинается новый шаг цикла.

Приведем пример использования процедуры:

```
main()
{
    char s[80] = "12.12.12";
    replaceAll ( s, "12", "A12B" );
    puts ( s );
}
```

В результате должна быть выведена строка «A12B.A12B.A12B».

Аналогичная процедура на языке C++ выглядит значительно проще благодаря более развитым средствам работы со строками:

```
void replaceAll ( string &s, string wOld, string wNew )
{
    string res = "";
    int p, len=wOld.size();
    while ( s.size() > 0 )
```

```

{
    p = s.find ( wOld );
    if ( p < 0 )
    {
        res = res + s;
        break;
    }
    if ( p > 0 )
        res = res + s.substr ( 0, p );
    res = res + wNew;
    if ( p + len > s.size() )
        s = "";
    else s.erase ( 0, p + len );
}
s = res;
}

```

Поскольку строка **s** изменяется, она передаётся по ссылке и в заголовке процедуры перед её именем ставится знак **&**. Переменная **len** хранит длину строки-образца, в остальном алгоритм не изменяется.

Построенную выше процедуру на языке C++ можно легко превратить в функцию. Для этого нужно

- в заголовке функции указать, что она возвращает строку (использовать ключевое слово **string** вместо **void**);
- убрать в заголовке процедуры символ **&** перед именем исходной строки (она не должна изменяться);
- вернуть результат с помощью оператора **return**.

Ниже показаны все изменённые части подпрограммы:

```

string replaceAll ( string s, string wOld, string wNew )
{
    ...
    return res;
}

```

Вызывать функцию можно таким образом:

```

main()
{
    string s = "12.12.12";
    s = replaceAll ( s, "12", "A12B" );
    cout << s;
    cin.get();
}

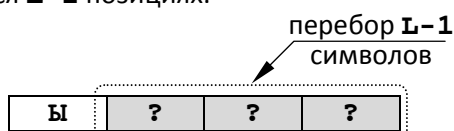
```

К сожалению, преобразовать процедуру на языке C в функцию не получится, потому что функции в этом языке не могут возвращать массив в качестве результата.

## Рекурсивный перебор

В алфавите языка племени «тумба-юмба» четыре буквы: «Ы», «Ш», «Ч» и «О». Нужно вывести на экран все слова, состоящие из **L** букв, которые можно построить из букв этого алфавита.

Это типичная задача на перебор вариантов, которую удобно свести к задаче меньшего размера. Будем определять буквы слова последовательно, одну за другой. Первая буква может быть любой из четырёх букв алфавита. Предположим, что сначала первой мы поставили букву **'Ы'**. Тогда для того, чтобы получить все варианты с первой буквой **'Ы'**, нужно перебрать все возможные комбинации букв на оставшихся **L-1** позициях:



Далее поочередно ставим на первое место все остальные буквы, повторяя процедуру:

```

перебор L символов
w[0]='Ы'; перебор последних L-1 символов
w[0]='Ш'; перебор последних L-1 символов
w[0]='Ч'; перебор последних L-1 символов
w[0]='О'; перебор последних L-1 символов

```

Здесь через **w** обозначена символьная строка, в которой хранится рабочее слово. Таким образом, задача для слов длины **L** свелась к 4-м задачам для слов длины **L-1**. Как вы знаете, такой прием называется *рекурсией*, а процедура – рекурсивной.

Когда рекурсия должна закончиться? Тогда, когда все символы расставлены, то есть количество установленных символов **N** равно **L**. При этом нужно вывести получившееся слово на экран и выйти из процедуры.

Подсчитаем количество всех возможных слов длины **L**. Очевидно, что слов длины 1 всего 4. Добавляя ещё одну букву, получаем  $4 \cdot 4 = 16$  комбинаций, для трех букв –  $4 \cdot 4 \cdot 4 = 64$  слова и т.д. Таким образом, слов из **L** букв может быть  $4^L$ .

Сначала напишем программу на языке C. В основной программе построим слово (символьную строку) **word** нужной длины (что в ней будет записано, не имеет значения). Процедуре **TumbaWords** передается алфавит в виде символьной строки, слово **word** и число уже установленных символов (в начале – 0) :

```

main()
{
    char word[] = "...";
    TumbaWords ( "ЫШЧО", word, 0 );
    getchar();
}

```

В процедуре используется описанный выше рекурсивный алгоритм:

```

void TumbaWords( char A[], char w[], int N )
{
    int i;
    if ( N == strlen(w) )
    {
        puts ( w );
        return;
    }
    for ( i = 1; i < strlen(A); i ++ )
    {
        w[N] = A[i];
        TumbaWords ( A, w, N+1 );
    }
}

```

Если условие в начале процедуры ложно (не все символы расставлены), в цикле перебираем все символы алфавита и поочередно ставим их на первое свободное место, а затем вызываем рекурсивно эту же процедуру, увеличивая третий аргумент на 1.

Приведем аналогичную программу на C++:

```

void TumbaWords( string A, string &w, int N )
{
    int i;
    if ( N == w.size() )
    {
        cout << w << endl;
        return;
    }
    for ( i = 1; i < A.size(); i ++ )
    {
        w[N] = A[i];
        TumbaWords ( A, w, N+1 );
    }
}

```

```

}
main()
{
    string word = "...";
    TumbaWords ( "ЪШЧО", word, 0 );
    cin.get();
}

```

Обратите внимание, что параметр **w** (строка-результат) – это изменяемый параметр.

## Сравнение и сортировка строк

Строки, как и числа, можно сравнивать. Для строк, состоящих из одних букв (русских или латинских), результат сравнения очевиден: меньше будет та строка, которая идет раньше в алфавитном порядке. Например, слово «паровоз» будет «меньше», чем слово «пароход»: они отличаются в пятой букве и «в» < «х». Более короткое слово, которое совпадает с началом более длинного, тоже будет стоять раньше в алфавитном списке, поэтому «пар» < «парк».

Но откуда компьютер «знает», что такое «алфавитный порядок»? И как сравнивать слова, в которых есть и строчные и заглавные буквы, а также цифры и другие символы. Что больше, «ПАР», «Пар» или «пар»?

Оказывается, при сравнении строк используются коды символов. Тогда получается, что

«ПАР» < «Пар» < «пар».

Возьмем пару «ПАР» и «Пар». Первый символ в обоих словах одинаков, а второй отличается – в первом слове буква заглавная, а во втором – такая же, но строчная. В таблице символов заглавные буквы стоят раньше строчных, и поэтому имеют меньшие коды. Поэтому «А» < «а», «П» < «а» и «ПАР» < «Пар» < «пар».

А как же с другими символами (цифрами, латинскими буквами)? Цифры стоят в кодовой таблице по порядку, причём раньше, чем латинские буквы; латинские буквы – раньше, чем русские; заглавные буквы (русские и латинские) – раньше, чем соответствующие строчные. Поэтому

«5STEAM» < «STEAM» < «Steam» < «steam» < «ПАР» < «Пар» < «пар».

Сравнение строк используется, например, при сортировке. Рассмотрим такую задачу: ввести с клавиатуры 10 фамилий и вывести их на экран в алфавитном порядке.

Для хранения данных удобно выделить массив строк. Поскольку в языке C строка хранится в массиве, то массив строк – это массив, составленный из массивов:

```

const int N = 10;
char S[N][80];

```

Здесь **S** – это массив из **N** строк, в каждой строке может храниться до 79 символов (+ один символ с кодом 0, завершающий строку).

Для сравнения строк в языке C используется функция **strcmp** (от англ. *string comparison* – сравнение строк). Она принимает два параметра (сравниваемые строки) и возвращает «разность» строк<sup>12</sup>:

- 0, если они равны;
- положительное число, если первая строка «больше» второй (стоит после второй в алфавитном списке);
- отрицательное число, если первая строка «меньше» второй.

Для обращения к строке с номером **i** используют, как обычно, форму **S[i]**. Ввод и вывод массива строк выполняется в цикле с помощью функций **gets** и **puts**:

```

main()
{
    const int N = 10;
    char s1[80], S[N][80];
    int i, j;
    printf ( "Введите строки: \n" );

```

<sup>12</sup> Значение, которое возвращает функция **strcmp** – это разность кодов первых различных символов двух строк или 0, если все символы совпали.



```

for ( i = 0; i < N; i ++ )
    gets ( S[i] );

for ( i = 0; i < N-1; i ++ )
    for ( j = N-2; j >= i; j -- )
        if ( strcmp(S[j], S[j+1]) > 0 )
        {
            strcpy (s1, S[j]);
            strcpy (S[j], S[j+1]);
            strcpy (S[j+1], s1);
        }

printf ( "После сортировки: \n" );
for ( i = 0; i < N; i ++ )
    puts ( S[i] );
}

```

Сортировка выполняется методом «пузырька», а для перестановки двух строк из массива используется вспомогательная строка `s1`.

В языке C++, где есть тип данных `string`, ситуация упрощается, и программа выглядит более понятной:

```

main()
{
    const int N = 10;
    string s1, S[N];
    int i, j;
    cout << "Введите строки: \n";
    for ( i = 0; i < N; i ++ )
        getline ( cin, S[i] );

    for ( i = 0; i < N-1; i ++ )
        for ( j = N-2; j >= i; j -- )
            if ( S[j] > S[j+1] )
            {
                s1 = S[j];
                S[j] = S[j+1];
                S[j+1] = s1;
            }

    cout << "После сортировки: \n";
    for ( i = 0; i < N; i ++ )
        cout << S[i] << endl;
}

```



### Контрольные вопросы

1. Что такое символьная строка?
2. Как хранятся строки в языках C и C++?
3. Как обращаться к элементу строки с заданным номером?
4. Как вычисляется длина строки?
5. Что обозначает оператор '+' применительно к строкам в C++?
6. Перечислите основные операции со строками и соответствующие им стандартные функции.
7. Как определить, что при поиске в строке образец не найден?
8. Чем отличаются средства языков C и C++ для работы со строками? Какой вариант вам больше нравится?
9. Как преобразовать число из символьного вида в числовой и обратно?
10. Почему строку не всегда можно преобразовать в число?
11. Объясните выражение «рекурсивный перебор».
12. Сравните рекурсивные и нерекурсивные методы решения переборных задач.



## Задачи и задания

1. Напишите программу, которая во введенной символьной строке заменяет все буквы «а» на буквы «б» и наоборот, как заглавные, так и строчные. При вводе строки «**абсАБС**» должен получиться результат «**басБАС**».
2. Ввести символьную строку и проверить, является ли она *палиндромом* (палиндром читается одинаково в обоих направлениях, например, «казак»).
3. Ввести адрес файла и «разобрать» его на части, разделенные знаком '/'. Каждую часть вывести в отдельной строке.
4. Ввести строку, в которой записана сумма натуральных чисел, например, «**1+25+3**». Вычислите это выражение.
5. Ввести с клавиатуры в одну строку фамилию, имя и отчество, разделив их пробелом. Вывести фамилию и инициалы. Например, при вводе строки «**Иванов Петр Семёнович**» должно получиться «**П.С. Иванов**».
6. Разберитесь, как работает еще одна функция замены на языке C++:

```
string replaceAll ( string s, string wOld, string wNew )
{
    int p, len=wOld.size();
    p=s.find ( wOld );
    while ( p >= 0 )
    {
        s.erase( p, len );
        s.insert ( p, len );
        p=s.find ( wOld );
    }
    return s;
}
```

Приведите пример входных данных, при которых эта функция работает неправильно.

7. Напишите рекурсивную версию процедуры **replaceAll**. Сравните две версии. Какую из них вы рекомендуете выбрать и почему?
8. Напишите функцию, которая изменяет в имени файла расширение на заданное (например, на «**.bak**»). Функция принимает два параметра: имя файла и нужно расширение. Учтите, что в исходном имени расширение может быть пустым.
9. Напишите функцию, которая определяет, сколько раз входит в символьную строку заданное слово.
10. С клавиатуры вводится число **N**, обозначающее количество футболистов команды «Бублик», а затем – **N** строк, в каждой из которых – информация об одном футболисте таком формате:  
 <Фамилия> <Имя> <год рождения> <голы>  
 Все данные разделяются одним пробелом. Нужно подсчитать, сколько футболистов, родившихся в период с 1998 по 2000 год, не забивали мячей вообще.
11. В условиях предыдущей задачи определите фамилию и имя футболиста, забившего наибольшее число голов, и количество забитых им голов.
12. В условиях предыдущей задачи выведите в алфавитном порядке фамилии и имена всех футболистов, которые забивали хотя бы один гол. В списке не более 100 футболистов.
13. Измените программу рекурсивного перебора так, чтобы длину слова можно было ввести с клавиатуры.
14. Выведите на экран все слова из **L** букв, в которых буква «Ы» встречается более 1 раза, и подсчитайте их количество.
15. Выведите на экран все слова из **L** букв, в которых есть одинаковые буквы, стоящие рядом (например, «ЫШШО»), и подсчитайте их количество.
16. В языке племени «тумба-юмба» запрещено ставить две гласные буквы подряд. Выведите все слова длины **L**, удовлетворяющие этому условию, и найдите их количество.

17. \*Напишите программу перебора слов заданной длины, не использующую рекурсию. Попробуйте составить функцию, которая на основе некоторой комбинации вычисляет следующую за ней.
18. \*Перестановки. К вам пришли **L** гостей. Напишите программу, которая выводит все *перестановки* – способы посадить их за столом.Guests можно обозначить латинскими буквами.

## § 67. Матрицы

### Что такое матрицы?

Многие программы работают с данными, организованными в виде таблиц. Например, при составлении программы для игры в крестики-нолики нужно запоминать состояние каждой клетки квадратной доски. Можно поступить так: пустым клеткам присвоить код  $-1$ , клетке, где стоит нолик – код  $0$ , а клетке с крестиком – код  $1$ . Тогда информация о состоянии поля может быть записана в виде таблицы:

	0	1	2
0	-1	0	1
1	-1	0	1
2	0	1	-1

Такие таблицы называются *матрицами* или *двухмерными массивами*. Каждый элемент матрицы, в отличие от обычного (линейного) массива имеет два индекса – номер строки и номер столбца. На рисунке фоном выделен элемент, находящийся на пересечении строки 1 и столбца 2 (в языках C и C++ строки и столбцы нумеруются с нуля<sup>13</sup>).

**Матрица** — это прямоугольная таблица, составленная из элементов одного типа (чисел, строк и т.д.). Каждый элемент матрицы имеет два индекса – номера строки и столбца.

При объявлении матриц в двух парах квадратных скобок указывают два размера (количество строк и количество столбцов)

```
const int N = 3, M = 4;
int A[N][M];
double X[10][12];
bool L[N][2];
```

Каждому элементу матрицы можно присвоить любое значение, допустимое для выбранного типа данных. Поскольку индексов два, для заполнения матрицы нужно использовать вложенный цикл. Далее в примерах будем считать, что объявлена матрица из **N** строк и **M** столбцов, а **i** и **j** – целочисленные переменные, обозначающие индексы строки и столбца. В этом примере матрица заполняется случайными числами и выводится на экран:

```
for ( i = 0; i < N; i++ )
{
    for ( j = 0; j < M; j++ )
    {
        A[i][j] = irand(20, 80);
        printf( "%3d", A[i][j] );
    }
    printf( "\n" );
}
```

```
for ( i = 0; i < N; i++ )
{
    for ( j = 0; j < M; j++ )
    {
        A[i][j] = irand(20, 80);
        cout.width(3);
        cout << A[i][j];
    }
    cout << endl;
}
```

Такой же двойной цикл нужно использовать для перебора всех элементов матрицы. Вот как вычисляется сумма всех элементов:

```
sum = 0;
```

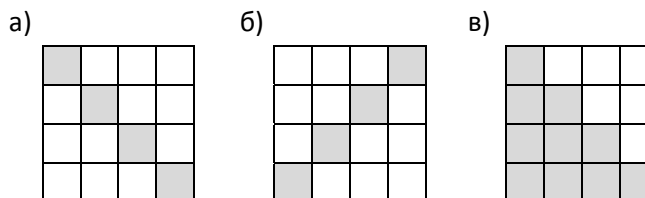
<sup>13</sup> Иногда бывает удобна нумерация с единицы, в этом случае можно выделить матрицу большего размера (на один столбец и одну строку больше) и не использовать элементы с нулевыми индексами.

```
for ( i = 0; i < N; i++ )
    for ( j = 0; j < M; j++ )
        sum += A[i][j];
```

## Обработка элементов матрицы

Покажем, как можно обработать (например, сложить) некоторые элементы квадратной матрицы **A**, содержащей **N** строк и **N** столбцов.

На рис. а выделена главная диагональ матрицы, на рис. б – вторая (побочная) диагональ, на рис. в – главная диагональ и все элементы под ней.



Главная диагональ – это элементы **A[0][0]**, **A[1][1]**, ..., **A[N-1][N-1]**, то есть номер строки равен номеру столбца. Для перебора этих элементов нужен один цикл:

```
for ( i = 0; i < N; i++ )
{
    // работаем с A[i][i]
}
```

Элементы побочной диагонали – это **A[0][N-1]**, **A[1][N-2]**, ..., **A[N-1][0]**. Заметим, что сумма номеров строки и столбца для каждого элемента равны **N-1**, поэтому получаем такой цикл перебора:

```
for ( i = 0; i < N; i++ )
{
    // работаем с A[i][N-1-i]
}
```

В случае «в» (обработка всех элементов на главной диагонали и под ней) нужен вложенный цикл: номер строки будет меняться от 0 до **N-1**, а номер столбца для каждой строки **i** – от 0 до **i**:

```
for ( i = 0; i < N; i++ )
    for ( j = 0; j <= i; j++ )
    {
        // работаем с A[i][j]
    }
```

Чтобы переставить строки или столбцы, достаточно одного цикла. Например, переставим строки 2 и 4, используя вспомогательную целую переменную **c**:

```
for ( j = 0; j < M; j++ )
{
    c = A[2][j];
    A[2][j] = A[4][j];
    A[4][j] = c;
}
```



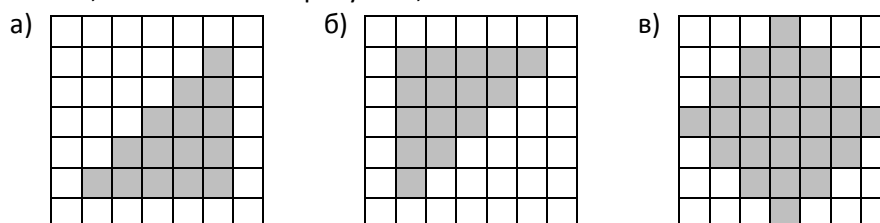
## Контрольные вопросы

1. Что такое матрицы? Зачем они нужны?
2. Сравните понятия «массив» и «матрица».
3. Как вы думаете, можно ли считать, что первый индекс элемента матрицы – это номер столбца, а второй – номер строки?
4. Могут ли индексы элементов матрицы принимать отрицательные значения?
5. Что такое главная и побочная диагонали матрицы?
6. Почему суммирование элементов главной диагонали требует одиночного цикла, а суммирование элементов под главной диагональю – вложенного?

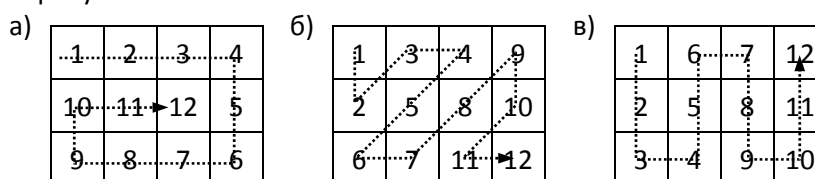


## Задачи и задания

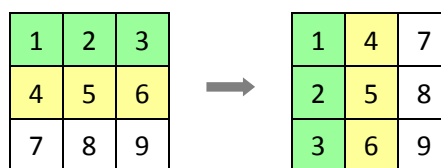
1. Напишите программу, которая находит минимальный и максимальный элементы матрицы и их индексы.
2. Напишите программу, которая находит минимальный и максимальный из чётных положительных элементов матрицы и их индексы. Учтите, что таких элементов в матрице может и не быть.
3. Напишите программу, которая выводит на экран строку матрицы, сумма элементов которой наибольшая.
4. Напишите программу, которая выводит на экран столбец матрицы, сумма элементов которой наименьшая.
5. Напишите программу, которая заполняет матрицу случайными числами, а затем записывает нули во все элементы выше главной диагонали.
6. Напишите программу, которая заполняет матрицу случайными числами, а затем записывает нули во все элементы выше побочной диагонали.
7. Напишите программу, которая заполняет матрицу  $7 \times 7$  случайными числами, а затем записывает в элементы, отмеченные на рисунках, число 99:



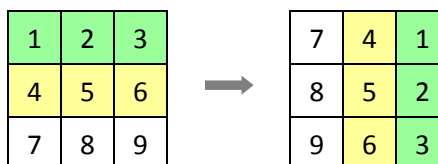
8. Заполните матрицу, содержащую  $N$  строк и  $M$  столбцов, натуральными числами по спирали и змейкой, как на рисунках:



9. Заполните квадратную матрицу случайными числами и выполните её *транспонирование*: так называется процедура, в результате которой строки матрицы становятся столбцами, а столбцы – строками:



10. Заполните квадратную матрицу случайными числами и выполните её поворот на 90 градусов:



11. \*Напишите программу, которая играет с человеком в крестики-нолики.
12. \*В матрице, содержащей  $N$  строк и  $M$  столбцов, записана карта островного государства Лимония (нули обозначают море, а единицы – сушу). Все острова имеют форму прямоугольника. Написать программу, которая по готовой карте определяет количество островов.

## § 68. Работа с файлами

### Как работать с файлами?

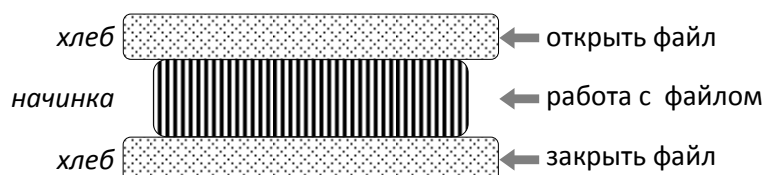
Файл – это набор данных на диске, имеющий имя. С точки зрения программиста, бывают файлы двух типов:

- 1) *текстовые*, которые содержат текст, разбитый на строки; таким образом, из всех специальных символов в текстовых файлах могут быть только символы перехода на новую строку;
- 2) *двоичные*, в которых могут содержаться любые данные и любые коды без ограничений; в двоичных файлах хранятся рисунки, звуки, видеофильмы и т.д.

Мы будем рассматривать только текстовые файлы.

Работа с файлом из программы включает три этапа. Сначала надо *открыть файл*, то есть сделать его активным для программы. Если файл не открыт, то программа не может к нему обращаться. При открытии файла указывают режим работы: чтение, запись или добавление данных в конец файла. Чаще всего открытый файл блокируется так, что другие программу не могут использовать его. Когда файл открыт (активен) программа выполняет все необходимые операции с ним. После этого нужно *заккрыть файл*, то есть освободить его, разорвать связь с программой. Именно при закрытии все последние изменения, сделанные программой в файле, записываются на диск.

Такой принцип работы иногда называют «принципом сэндвича», в котором три слоя: хлеб, затем начинка, и потом снова хлеб:



В большинстве языков программирования с файлами работают через вспомогательные переменные (их называют указатели, идентификаторы и т.п.). Например, в языке С используются указатели на файлы (англ. *file pointer*):

```
FILE *F;
```

Как вы знаете, указатель – это переменная, в которую можно записать адрес какого-то объекта. В данном случае в переменной **F** можно хранить адрес специального блока данных для управления файлом, который называется **FILE** и строится в памяти при открытии (активации) файла с помощью функции **fopen**:

```
F = fopen ( "input.txt", "r" );
```

В этом примере файл с именем **input.txt** из текущего каталога (то есть из каталога, откуда запускается программа) открывается в режиме чтения (на это указывает второй параметр «**r**», от англ. *read* – читать). Если второй параметр равен «**w**» (от англ. *write* – писать), файл открывается для записи. Можно ещё использовать режим добавления, указав вторым параметром «**a**» (от англ. *append* – добавить); в этом случае новые данные дописываются в конец файла, а если файла не было, он создается.

Если файл открыт неудачно (например, не обнаружен файл, открываемый для чтения), функция **fopen** вернет нулевой указатель (**NULL**). Этот факт можно использовать для обработки ошибок:

```
FILE *F;
F = fopen ( "input.txt", "r" );
if ( F )
{
    // здесь работаем с файлом
}
else
    printf("Открыть файл не удалось.");
```

Вспомните, что в языке С любое ненулевое значение воспринимается как истинное условие, а любое нулевое (в том числе и нулевой указатель **NULL**) – как ложное. Поэтому вместо **if (F != NULL)** часто пишут просто **if (F)**.

Функция **fclose** (от англ. *file close* – закрыть файл) закрывает файл, освобождая его. Например, если нужно читать данные из файла **input.txt** и записывать результаты в файл **output.txt**, можно использовать такую структуру:

```
FILE *Fin, *Fout;
Fin = fopen ( "input.txt", "r" );
Fout = fopen ( "output.txt", "w" );
    // здесь работаем с файлами
fclose (Fin);
fclose (Fout);
```

После закрытия файла файловую переменную можно использовать повторно, для работы с этим или другим файлом. После окончания работы программы все открытые файлы закрываются автоматически.

Если файл, который открывается на чтение, не найден, возникает ошибка. Если существующий файл открывается на запись, его содержимое уничтожается.

Чтение из текстовых файлов выполняет функция **fscanf**, а запись в файлы – функция **fprintf**, которые используются почти также, как и уже знакомые вам функции **scanf** и **printf**, но в первом параметре им передаётся указатель на открытый файл. В остальном чтение и запись происходит так же, как и для стандартных устройств – клавиатуры и текстового экрана. Если в переменных **Fin** и **Fout** записаны указатели на файлы, открытые, соответственно, на ввод и на вывод, можно написать так:

```
fscanf ( Fin, "%d%d", &a, &b );
fprintf ( Fout, "%d+%d=%d", a, b, a+b );
```

У нас получилась программа, которая складывает два целых числа, прочитанные из одного файла, и записывает результат в другой файл.

Как правило, текстовый файл – это «устройство» последовательного доступа к данным. Это значит, что для того, чтобы прочитать 100-е по счёту значение из файла, нужно сначала прочитать предыдущие 99. В своей внутренней памяти система хранит положение указателя (файлового курсора), который определяет текущее место в файле. При открытии файла указатель устанавливается в самое начало файла, при чтении смещается на позицию, следующую за прочитанными данными, а при записи – на следующую свободную позицию. Если нужно повторить чтение с начала файла, нужно закрыть его (с помощью функции **fclose**), а потом снова открыть.

Когда файловый курсор указывает на конец файла, логическая функция **feof** (от англ. *end of file* – конец файла) возвращает истинное значение:

```
if ( feof(Fin) )
    printf("Данные кончились");
```

В языке C++ для работы с текстовыми файлами используются файловые потоки ввода-вывода, которые подключаются с помощью заголовочного файла **fstream**:

```
#include <fstream>
```

Входной поток (файл для чтения) связан с переменной типа **ifstream** (от англ. *input file stream* – входной файловый поток), а выходной поток – с переменной типа **ofstream** (от англ. *output file stream* – выходной файловый поток):

```
ifstream Fin;
ofstream Fout;
```

Для открытия файла (потока) используется метод **open**, а для закрытия – метод **close**:

```
Fin.open ( "input.txt" );
Fout.open ( "output.txt" );
    // здесь работаем с файлами
Fin.close();
Fout.close();
```

Если поток открыт неудачно, значение переменной будет равно **NULL**, что можно использовать для обработки ошибки:

```
ifstream F;
F.open ( "input.txt" );
if ( F )
{
```



```
// здесь работаем с файлом
}
else
    printf ( "Открыть файл не удалось." );
```

Если в переменных **Fin** и **Fout** записаны указатели на потоки, открытые, соответственно, на ввод и на вывод, можно написать так:

```
Fin >> a >> b;
Fout << a << "+" << b << "=" << a+b;
```

В этом фрагменте программы складываются два целых числа, прочитанные из одного файла, и результат записывается в другой файл.

Когда файловый курсор указывает на конец файла, логическая функция-метод **eof** (от англ. *end of file* – конец файла) возвращает истинное значение:

```
if ( F.eof() )
    printf ( "Данные кончились" );
```

## Неизвестное количество данных

Предположим, что в текстовом файле записано в столбик неизвестное количество чисел и требуется найти их сумму. В этой задаче не нужно одновременно хранить все числа в памяти (и не нужно выделять массив!), достаточно читать по одному числу за раз:

```
пока не конец файла
    прочитать число из файла
    добавить его к сумме
```

Далее будем считать, что файловая переменная **Fin** связана с файлом, открытым на чтение. Основная часть программы (без объявления переменных, открытия и закрытия файлов) выглядит так:

```
S = 0;
while ( !feof(Fin) )
{
    fscanf ( Fin, "%d", &x );
    S = S + x;
}
```

```
S = 0;
while ( !Fin.eof() )
{
    Fin >> x;
    S = S + x;
}
```

К сожалению, в программе на языке C есть одна серьёзная проблема: если после последнего числа в файле стоит символ перевода строки (есть одна или несколько пустых строк), последнее прочитанное число будет учтено в сумме дважды. Дело в том, что после чтения этого числа файловый курсор еще не вышел на конец файла и условие цикла истинно. В то же время следующее чтение заканчивается неудачно, а в переменной **x** остается последнее прочитанное значение, которое ещё раз добавляется к сумме. Исправить ситуацию можно так:

```
int n;
S = 0;
while ( 1 )
{
    n = fscanf ( Fin, "%d", &x );
    if ( n < 1 ) break;
    S = S + x;
}
```

Здесь использован бесконечный цикл вида

```
while ( 1 )
{
    ...
}
```

Так как единица соответствует истинному условию, это условие никогда не будет нарушено, так что выйти из цикла можно только с помощью оператора **break**. Выход происходит в том случае, когда функция **fscanf** не смогла прочитать данные. Это можно определить по возвращаемому значению: функция **scanf** возвращает количество прочитанных значений. В нашем случае если прочитано меньше одного значения, то чтение закончено неудачно (данные закончились). Это

может случиться тогда, когда достигнут конец файла или встречен недопустимый (нецифровой) символ.

## Обработка массивов

В текстовом файле записано не более 100 целых чисел. Требуется вывести в другой текстовый файл те же числа, отсортированные в порядке возрастания.

Особенность этой задачи в том, что для сортировки нам нужно удерживать в памяти все числа, то есть для их хранения нужно выделить массив. Косвенно на это указывает ограничение – чисел не более 100. Поэтому массив должен иметь не менее 100 элементов:

```
const int MAX = 100;
int A[MAX];
```

Основная интрига состоит в том, что количество чисел точно неизвестно. Следовательно, нам нужно считать прочитанные значения и записывать их последовательно в первые ячейки массива. Если данные закончились, цикл чтения останавливается. Кроме того, нужно сделать защиту от слишком большого количества данных: если 100 чисел уже записаны в массив, цикл должен остановиться, потому что следующие числа записывать некуда. Ниже приведены циклы чтения на языке C:

```
N = 0;
while ( N < MAX )
{
    r = fscanf ( Fin, "%d", &A[N] );
    if ( r < 1 ) break;
    N ++;
}
```

и на C++:

```
N = 0;
while ( N < MAX && !Fin.eof() )
{
    Fin >> A[N];
    N ++;
}
```

Целая переменная **N** служит счётчиком прочитанных из файла чисел.

Теперь нужно отсортировать первые **N** значений массива **A** (этот код вы уже можете написать самостоятельно) и вывести их во второй файл, открытый на запись (**Fout** – указатель на файл):

```
Fout = fopen ( "output.txt", "w" );
for ( i = 0; i < N; i++ )
    fprintf ( Fout, "%d\n", A[i] );
fclose ( Fout );
```

Вариант на C++ (здесь **Fout** – выходной файловый поток):

```
Fout.open ( "output.txt" );
for ( i = 0; i < N; i++ )
    Fout << A[i] << endl;
Fout.close();
```

## Обработка строк

Как известно, современные компьютеры большую часть времени заняты обработкой символьной, а не числовой информации. Предположим, что в текстовом файле записаны данные о собаках, привезенных на выставку: в каждой строчке кличка собаки, ее возраст (целое число) и порода, например,

**Мухтар 4 немецкая овчарка**

Все элементы отделяются друг от друга одним пробелом. Нужно вывести в другой файл сведения о собаках, которым меньше 5 лет.

В этой задаче данные можно обрабатывать по одной строке (не нужно загружать все строки в оперативную память):

```

пока не конец файла (Fin)
  прочитать строку из файла Fin
  разобрать строку – выделить возраст
  если возраст < 5 то
    записать строку в файл Fout

```

Здесь, как и раньше, **Fin** и **Fout** – файловые переменные, связанные с файлами, открытыми на чтение и запись соответственно (для C++ – входной и выходной файловые потоки).

Сначала разберёмся с чтением символьных строк из файла. В языке C для этого используется функция **fgets**, которая принимает три параметра: имя строки, размер строки и указатель на файл, откуда нужно читать:

```
fgets( s, 80, Fin );
```

Кроме того, эта функция может определить, успешно ли выполнено чтение – в случае неудачи она возвращает нулевой указатель. Поэтому цикл чтения вида «пока не конец файла» можно записать так:

```

while( fgets(s, 80, Fin) )
{
  ...
}

```

В языке C++ аналогично работает со строками функция **getline**:

```

while( getline(Fin, s) )
{
  ...
}

```

Теперь подумаем, как обработать прочитанную строку. Будем считать, что все данные корректны, то есть первый пробел отделяет кличку от возраста, а второй – возраст от породы. Тогда разбор строки можно выполнить так:

```

найти в строке пробел
удалить из строки кличку с первым пробелом
найти в строке пробел
выделить возраст перед пробелом
преобразовать возраст в числовой вид

```

На языке C эти операции записываются всего в две строчки:

```

char s[80], *p;
int age;
...
p = strchr( s, ' ' );
sscanf( p+1, "%d", &age );

```

Функция **strchr** (см. § 66. ) находит первый пробел и записывает его адрес в переменную-указатель **p**. Затем функция **sscanf** читает из этой строки со следующего символа (первый аргумент) целое число (по формату **%d**) и записывает в память по адресу переменной **age** (третий аргумент – **&age**). Таким образом, функция **sscanf** работает точно так же, как и знакомая нам функция **scanf**, но в качестве источника данных использует не ввод с клавиатуры, а символьную строку из памяти.

Вариант на языке C++ получается несколько длиннее:

```

string s, s1;
int p, age;
...
p = s.find( ' ' );
s1 = s.substr( p+1 );
age = atoi( s1.c_str() );

```

В переменную **p** записывается номер первого пробела в строке, затем во временную переменную-строку **s1** записываем всю хвостовую часть исходной строки после пробела (возраст и породу собаки). Затем эта строка преобразуется в целое число с помощью функции **atoi**. Так как функция **atoi** остановится на первом нецифровом символе, всё, что записано после числа никак не влияет на результат.

В результате работы приведённого выше фрагмента возраст собаки оказывается в переменной **age**. Обратите внимание, что исходная строка *s* не изменяется, поэтому её можно использовать при выводе результата. Основной цикл будет выглядеть так:

```
Fin = fopen ( "input.txt", "r" );
Fout = fopen ( "output.txt", "w" );
while ( fgets( s, 80, Fin ) )
{
    p = strchr ( s, ' ' );
    sscanf ( p+1, "%d", &age );
    if ( age < 5 )
        fputs ( s, Fout );
}
fclose ( Fout );
fclose ( Fin );
```

```
Fin.open ( "input.txt" );
Fout.open ( "output.txt" );
while ( getline( Fin, s ) )
{
    p = s.find( ' ' );
    s1 = s.substr ( p+1 );
    age = atoi ( s1.c_str() );
    if ( age < 5 )
        Fout << s << endl;
}
Fout.close();
Fin.close();
```



### Контрольные вопросы

1. Чем отличаются текстовые и двоичные файлы по внутреннему содержанию? Можно ли сказать, что текстовый файл – это частный случай двоичного файла?
2. Объясните «принцип сэндвича» при работе с файлами.
3. Как вы думаете, почему открытый программой файл, как правило, блокируется и другие программы не могут получить к нему доступ?
4. Почему рекомендуется вручную закрывать файлы, хотя при закрытии программы они закроются автоматически? В каких ситуациях это может быть важно?
5. Что такое файловая переменная? Почему для работы с файлом используют не имя файла, а файловую переменную?
6. В каком случае одна и та же файловая переменная может быть использована для работы с несколькими файлами, а в каком – нет?
7. Что такое «последовательный доступ к данным»?
8. Как можно начать чтение данных из файла с самого начала?
9. Как определить, что данные в файле закончились?
10. В каких случаях нужно знать максимальное количество данных в файле, а в каких – нет?
11. В каких случаях нужно открывать одновременно несколько файлов?



### Задачи и задания

1. Напишите программу, которая находит среднее арифметическое всех чисел, записанных в файле в столбик, и выводит результат в другой файл.
2. Напишите программу, которая находит минимальное и максимальное среди чётных положительных чисел, записанных в файле, и выводит результат в другой файл. Учтите, что таких чисел может вообще не быть.
3. В файле в столбик записаны целые числа. Напишите программу, которая определяет длину самой длинной цепочки идущих подряд одинаковых чисел и выводит результат в другой файл.
4. В файле записано не более 100 чисел. Отсортировать их по возрастанию последней цифры и записать в другой файл.
5. В файле записано не более 100 чисел. Отсортировать их по возрастанию суммы цифр и записать в другой файл.
6. В двух файлах записаны отсортированные по возрастанию массивы неизвестной длины. Объединить их и записать результат в третий файл. Полученный массив также должен быть отсортирован по возрастанию.
7. Дополните решение задачи о собаках, так чтобы программа обрабатывала ошибки в исходных данных. При любых ошибках программа не должна завершаться аварийно.

8. В исходном файле записана речь подростка, в которой часто встречается слово-паразит «короче», например: «Мама, короче, мыла, короче, раму.» Убрать из текста все слова-паразиты (должно остаться «Мама мыла раму.»).
9. Прочитать текст из файла и подсчитать количество слов в нём.
10. Прочитать текст из файла и вывести в другой файл только те строки, в которых есть слова, начинающиеся с буквы А.
11. Прочитать текст из файла и вывести в другой файл в столбик все слова, которые начинаются с буквы А.
12. Прочитать текст из файла, заменить везде слово «паровоз» на слово «пароход» и записать в другой файл.
13. В файле записаны данные о результатах сдачи экзамена. Каждая строка содержит фамилию, имя и количество баллов, разделенные пробелами:  
**<Фамилия> <Имя> <Количество баллов>**  
Вывести фамилии и имена тех учеников, которые получили больше 80 баллов.
14. В предыдущей задаче добавить к списку нумерацию, например:  
**1) Иванов Вася**  
**2) Петров Петя**
15. В той же задаче сократить имя до одной буквы и поставить перед фамилией:  
**1) В. Иванов**  
**2) П. Петров**
16. В той же задаче отсортировать список по алфавиту (по фамилии).
17. \*В той же задаче отсортировать список по убыванию полученного балла (вывести балл в выходной файл).

## Самое важное в главе 8:

- Алгоритмы могут записываться на псевдокоде, в виде блок-схем и на языках программирования. Алгоритм, записанный на языке программирования, называется программой.
- Данные, с которыми работает программа, хранятся в переменных. Переменная - это величина, которая имеет имя, тип и значение. Значение переменной может изменяться во время выполнения программы.
- Любой алгоритм можно записать, используя последовательное выполнение операторов, ветвления и циклы. Ветвления предназначены для выбора вариантов действий в зависимости от выполнения некоторых условий. Цикл - это многократное повторение одинаковых действий.
- Подпрограммы - это вспомогательные алгоритмы, которые могут многократно вызываться из основной программы и других подпрограмм. Подпрограммы-процедуры выполняют описанные в них действия, а подпрограммы-функции в дополнение к этому возвращают вызвавшей программе результат этих действий (число, символ, логическое значение и т.д.). Данные, передаваемые в подпрограмму, называют аргументами. В подпрограмме эти данные представлены как локальные переменные, которые называются параметрами подпрограммы.
- Рекурсивные алгоритмы основаны на последовательном сведении исходной задачи ко всё более простым задачам такого же типа (с другими параметрами). Рекурсия служит заменой циклу. Любой рекурсивный алгоритм можно записать без рекурсии, но во многих случаях такая запись более длинная и менее понятная.
- Массив - это группа переменных одного типа, расположенных в памяти рядом (в соседних ячейках) и имеющих общее имя. Каждая ячейка массива имеет уникальный индекс (как правило, это номер элемента).
- Сортировка - это расстановка элементов массива в заданном порядке. Цель сортировки - облегчить последующий поиск. Для отсортированного массива можно применить двоичный поиск, который работает значительно быстрее, чем линейный.
- Символьная строка - это последовательность символов, расположенных в памяти рядом (в соседних ячейках). Строка представляет собой единый объект, она может менять свою длину во время работы программы.
- Матрица - это прямоугольная таблица, составленная из элементов одного типа (чисел, строк и т.д.). Каждый элемент матрицы имеет два индекса - номера строки и столбца.
- До выполнения операций с файлом нужно открыть файл (сделать его активным), а после завершения всех действий - закрыть (освободить).