

Глава II.

Хранение и обработка данных

1. Массивы.....	3
Основные понятия	3
Ввод с клавиатуры и вывод на экран	4
Заполнение случайными числами	5
Работа с текстовыми файлами	7
Работа с двоичными файлами	10
Простой поиск в массиве	11
Перестановка элементов массива	15
Сортировка массивов.....	16
Двоичный поиск в массиве	18
Массивы в процедурах и функциях	19
2. Символьные строки.....	21
Что такое символьная строка?	21
Стандартный ввод и вывод	22
Работа с файлами	23
Функции для работы со строками.....	25
Строки в функциях и процедурах.....	35
3. Матрицы (двухмерные массивы)	37
Что такое матрица?	37
Объявление матриц	38
Стандартный ввод и вывод	38
Работа с файлами	40
Алгоритмы для работы с матрицами	41
4. Массивы символьных строк.....	44
Объявление и инициализация	44
Ввод и вывод.....	44
Сортировка	45
5. Управление памятью	46
Указатели	46
Динамическое выделение памяти.....	47
Выделение памяти для матрицы	49
6. Рекурсия.....	51
Что такое рекурсия?	51
Не допустим бесконечную рекурсию!	52
Когда рекурсия не нужна	53
Рекурсивный поиск	54
Рекурсивные фигуры	55
Перебор вариантов	56
Быстрая сортировка (<i>QuickSort</i>)	59
7. Структуры.....	62
Что такое структуры?	62
Объявление и инициализация	62
Работа с полями структуры	63
Ввод и вывод.....	63
Копирование	64
Массивы структур	65
Динамическое выделение памяти.....	66

	Структуры как параметры процедур.....	67
	Сортировка по ключу	68

1. Массивы



Основные понятия



Что такое массив?

Основное предназначение компьютеров не вычисления, как считают многие, а **обработка больших объемов данных**. При размещении большого количества данных в памяти возникает такая проблема: надо научиться обращаться к каждой ячейке с данными отдельно. При этом очень сложно дать каждой ячейке собственное имя и при этом не запутаться. Выкручиваются из этой ситуации так: дают имя не ячейке, а группе ячеек, в которой каждая ячейка имеет номер. Такая область памяти называется массивом.

Массив — это группа ячеек памяти одинакового типа, расположенных рядом и имеющих общее имя. Каждая ячейка в группе имеет уникальный номер.

При работе с массивами надо научиться решать три задачи:

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



Объявление массива

Чтобы использовать массив, надо его объявить — выделить место в памяти. Типом массива называется тип массива это тип входящих в него элементов. Массивы могут быть разных типов — **int**, **float**, **char**, и т.д. Массив объявляют так же, как и обычные переменные, но после имени массива в квадратных скобках записывается его размер.

```
int A[10], B[20];    // 2 массива на 10 и 20 целых чисел
float C[12];         // массив из 12 вещественных чисел
```

При объявлении массива можно сразу заполнить его начальными значениями, перечисляя их внутри фигурных скобок:

```
int A[4] = { 2, 3, 12, 76 };
```

Если в списке в фигурных скобках записано меньше чисел, чем элементов в массиве, то оставшиеся элементы заполняются нулями. Если чисел больше, чем надо, транслятор сообщает об ошибке. Например,

```
int A[4] = { 2 };    // последние три элемента равны 0
```

Для повышения универсальности программы размер массива лучше определять через константу. В этом случае для переделки программы для массива другого размера надо только поменять значение этой константы:

```
const int N = 20;    // константа
main()
{
    int A[N];         // размер массива задан через константу
    ...
}
```

В таблице показаны примеры правильного и неправильного объявления массива.

правильно		неправильно	
<code>int A[20];</code>	размер массива указан явно	<code>int A[];</code>	размер массива неизвестен
<code>const int N = 20;</code> <code>int A[N];</code>	размер массива – постоянная величина	<code>int N = 20;</code> <code>int A[N];</code>	размер массива не может быть переменной



Обращение к элементу массива

Каждый элемент массива имеет свой порядковый номер. Чтобы обратиться к элементу массива, надо написать имя массива и затем в квадратных скобках номер нужного элемента. Важно запомнить одно важное правило:

Элементы массивов в языке Си нумеруются с нуля. Таким образом, если в массиве 10 элементов, он содержит элементы:

`A[0], A[1], A[2], ..., A[9]`

Номер элемента массива также называется его **индексом**. Вот примеры обращения к массиву **A**:

```
x = (A[3] + 5) * A[1]; // прочитать значения A[3] и A[1]
A[0] = x + 6;         // записать новое значение в A[0]
```

В языке Си не контролируется **выход за границы массива**, то есть формально вы можете записать что-то в элемент с несуществующим индексом, например в **A[345]** или в **A[-12]**. Однако при этом вы стираете какую-то ячейку в памяти, не относящуюся к массиву, поэтому последствия такого шага непредсказуемы и во многих случаях программа «зависает».



Ввод с клавиатуры и вывод на экран

Как же ввести данные в массив? Существует много способов в зависимости от вашей задачи:

- элементы массива вводятся с клавиатуры вручную;
- массив заполняется случайными числами (например, для моделирования случайных процессов);
- элементы массива читаются из файла;
- элементы массива поступают через порт с внешнего устройства (например, сканера, модема и т.п.);
- массив заполняется в процессе вычислений.

Задача. Ввести с клавиатуры массив из 10 элементов, умножить все элементы на 2 и вывести полученный массив на экран.

К сожалению, невозможно просто сказать компьютеру: «введи массив **A**». Мы должны каждый элемент прочитать отдельно.

Чтобы ввести массив в память, надо каждый его элемент обработать отдельно (например, вызвав для него функцию ввода **scanf**).

Ввод с клавиатуры применяется в простейших программах, когда объем вводимой информации невелик. Для ввода массива будем использовать цикл **for**. Напомним, что массив надо предварительно *объявить*, то есть выделить под него память.

Вводить можно столько элементов массива, сколько ячеек памяти выделено. Помните, что элементы массива нумеруются с нуля, поэтому если массив имеет всего 10 элементов, то последний элемент имеет номер 9. Если пытаться записывать в 10-ый элемент, произойдет выход за границы массива, и программа может работать неверно (а, возможно, и «зависнет»). При

вводе массива желательно выдать на экран общую подсказку для ввода всего массива и подсказки для каждого элемента.

Для умножения элементов массива на 2 надо снова использовать цикл, в котором за один раз обрабатывается 1 элемент массива.

Вывод массива на экран выполняется также в цикле **for**. Элементы выводятся по одному. Если в конце строки-формата в операторе **printf** поставить пробел, то элементы массива будут напечатаны в строчку, а если символ **\n** – то в столбик.

```
#include <stdio.h>
const int N = 10;          // размер массива
main()
{
    int i, A[N];            // объявление массива
    printf("Введите массив A\n"); // подсказка для ввода

    for ( i = 0; i < N; i ++ ) { // цикл по всем элементам
        printf("Введите A[%d]> ", i ); // подсказка для ввода A[i]
        scanf ("%d", &A[i]); // ввод A[i]
    }

    for ( i = 0; i < N; i ++ ) // цикл по всем элементам
        A[i] = A[i] * 2; // умножить A[i] на 2

    printf("\nРезультат:\n");

    for ( i = 0; i < N; i ++ ) // цикл по всем элементам
        printf("%d ", A[i]); // вывести A[i]
}
```



Заполнение случайными числами

Этот прием используется для моделирования случайных процессов, например, броуновского движения частиц. Пусть требуется заполнить массив равномерно распределенными случайными числами в интервале **[a,b]**. Поскольку для целых и вещественных чисел способы вычисления случайного числа в заданном интервале отличаются, рассмотрим оба варианта. Здесь и далее предполагается, что в начале программы есть строка

```
const int N = 10;
```

Описание функции-датчика случайных чисел находится в заголовочном файле **stdlib.h**. Удобно также добавить в свою программу функцию **random**:

```
int random (int N) { return rand() % N; }
```

которая выдает случайные числа с равномерным распределением в интервале **[0,N-1]**.

Как вы уже знаете из первой части курса, для получения случайных чисел с равномерным распределением в интервале **[a,b]** надо использовать формулу

```
k = random ( b - a + 1 ) + a;
```

Для вещественных чисел формула несколько другая:

```
x = rand() * (b - a) / RAND_MAX + a;
```

Здесь константа **RAND_MAX** – это максимальное случайное число, которое выдает стандартная функция **rand**.

В приведенном ниже примере массив **A** заполняется случайными *целыми* числами в интервале $[-5, 10]$, а массив **X** – случайными *вещественными* числами в том же интервале.

```
#include <stdlib.h>
const int N = 10;
main()
{
    int i, A[N], a = -5, b = 10;
    float X[N];

    for ( i = 0; i < N; i ++ )
        A[i] = random(b-a+1) + a;

    for ( i = 0; i < N; i ++ )
        X[i] = (float)rand()*(b-a)/RAND_MAX + a;

    // здесь что-то делаем с массивами
}
```

Возможно, в этом примере не вполне ясно, зачем перед вызовом функции **rand** поставлено слово (**float**). Это связано с тем, что у нас **a** и **b** – целые числа. Результат функции **rand** – тоже целое число. Здесь возможны две проблемы:

- При умножении результата функции **rand** на **b-a** может получиться очень большое число, которое не поместится в переменную типа **int**.
- В языке Си при делении целого числа на целое остаток отбрасывается, поэтому при делении результат будет неверным.

Когда массив заполняется случайными числами, обязательно вывести на экран исходный массив.

Задача. Заполнить массив случайными целыми числами в интервале $[-10, 15]$, умножить все элементы на 2 и вывести на экран исходный массив и результат.

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

const int N = 10;

main()
{
    int i, A[N];

    for ( i = 0; i < N; i ++ )    // заполнение массива сл. числами
        A[i] = random(26) - 10;

    printf("n Исходный массив:\n"); // вывод исходного массива
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);

    for ( i = 0; i < N; i ++ )    // умножить все элементы на 2
        A[i] = A[i] * 2;

    printf("n Результат:\n");
    for ( i = 0; i < N; i ++ )    // вывод результата
        printf("%d ", A[i]);

}
```



Работа с текстовыми файлами

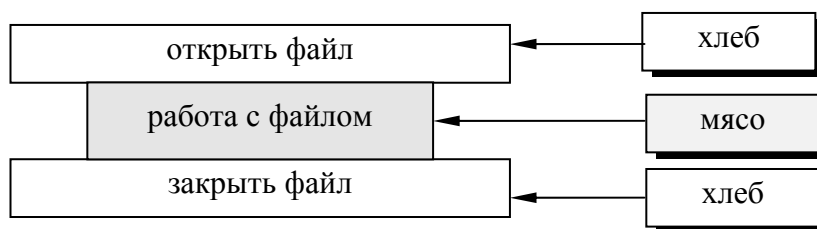
Наверняка при работе с предыдущими программами вы ощутили, что вводить с клавиатуры большое количество данных очень утомительно, особенно если это надо делать много раз. В таких случаях создают файл на диске, в который записывают нужные данные, а программа сама читает данные из файла.

Файлы бывают текстовые (в которых можно записывать только буквы, цифры, скобки и т.п.) и двоичные (в которых могут храниться любые символы из таблицы). В этом разделе мы рассмотрим только текстовые файлы.



Как работать с файлами из программы

Работа с файлами строит по принципу сэндвича:



Понятие «открыть файл» означает «начать с ним работу», сделать его активным и заблокировать обращение других программ к этому файлу. При закрытии файла он освобождается (теперь с ним могут работать другие программы) и все ваши изменения вносятся на диск.

Для работы с файлом используется специальная переменная, которая называется *указателем на файл*. Это адрес блока данных в памяти, в котором хранится вся информация об открытом файле. Объявляется указатель на файл так:

```
FILE *fp;
```

Чтобы открыть файл, надо вызвать функцию **fopen**, которая попытается открыть файл и записать его адрес в переменную **fp**. После этого все обращения к файлу выполняются не по имени файла, а через указатель **fp**.

```
fp = fopen ( "qq.dat", "r" );
```

Здесь файл **qq.dat** из текущего каталога открывается для чтения (режим **"r"** во втором параметре функции **fopen**). Если надо, можно указать полный (или относительный) путь к файлу, например так:

```
fp = fopen ( "c:\\data\\qq.dat", "r" );
```

Знак «наклонные черта» (слэш) в символьных строках всегда удваивается, потому что одиночный слэш – это специальный символ, например в сочетании **\\n**.

Кроме режима **"r"** (чтение из файла) есть еще несколько режимов:

- | | |
|-------------|---|
| "r" | Запись в новый файл. Если на диске уже есть файл с таким именем, он будет предварительно удален. |
| "a" | Добавление в конец файла. Если на диске уже есть файл с таким именем, новые данные дописываются в конец файла. Если такого файла нет, то он будет создан. |
| "r+" | Открыть существующий файл для изменения с возможностью записи и чтения |
| "w+" | Создание нового файла для записи и чтения (если файл с таким именем уже есть, он заменяется новым). |

Иногда программа не может открыть файл. Если файл открывается на чтение, это возможно в следующих случаях:

- неверно задано имя файла или файла нет на диске;
- файл используется другой программой и заблокирован.

Если файл открывается на запись, операция может закончиться неудачно, если

- на диске нет места;
- файл защищен от записи;
- неверно задано имя файла (например, оно содержит две точки, знак вопроса и т.п.).

Если файл не удалось открыть, функция **fopen** возвращает специальное нулевое значение (*нулевой указатель*), который обозначается **NULL**. Поэтому надо всегда проверять правильность открытия файла, особенно в режиме чтения. Если файл не был открыт, надо вывести сообщение об ошибке и выйти из программы.

```
if ( fp == NULL )
{
    printf("Нет файла с данными");
    return 1;    // выход по ошибке, код ошибки 1
}
```

Если файл открыт, можно читать из него данные. Для того используем функцию **fscanf**. Она полностью аналогична **scanf**, но служит для ввода из файла, а не с клавиатуры. Кроме того, ее первый параметр – указатель на файл, из которого читаются данные.

```
n = fscanf ( fp, "%d", &A[i] );
```

Функция **fscanf** возвращает результат – количество чисел, которые ей удалось прочитать. Если мы запрашивали одно число, то значение переменной **n** может быть равно единице (если все нормально) или нулю (если данные закончились или ошибочные, например, вместо чисел введено слово). Для успешного чтения данные в файле должны отделяться пробелом или символом перехода на новую строку (он вставляется в файл при нажатии на клавишу **Enter**).

Если файл открыт на запись, можно записать в него данные с помощью функции **fprintf**, которая полностью аналогична **printf**.

Когда работа с файлом закончена, надо закрыть его, вызвав функцию **fclose**:

```
fclose ( fp );
```

После этого указатель **fp** свободен и его можно использовать для работы с другим файлом.



Массив известного размера

Задача. Ввести массив из 10 целых чисел из файла **input.dat**, умножить каждый элемент на 2 и вывести в столбик в файл **output.dat**.

Эта задача решается с помощью функций **fopen**, **fscanf**, **fprintf** и **fclose**, описанных выше. В программе обрабатываются две ошибки:

- файла нет (его не удалось открыть);
- в файле мало данных или данные неверные (например, слова вместо целых чисел).


```
#include <stdio.h>

const int N = 10;

main()
{
    int i, A[N];

    FILE *fp;    // указатель на файл

    fp = fopen( "input.dat", "r" );    // открыть файл на чтение
    if ( fp == NULL ) {                // обработка ошибки
        printf("Нет файла данных");
        return 1;    // выход по ошибке, код ошибки 1
    }

    for ( i = 0; i < N; i ++ )
        if ( 0 == fscanf(fp,"%d",&A[i]) ) {    // чтение и обработка
            printf("Не хватает данных в файле"); // ошибки
            break;
        }
    fclose ( fp );    // закрыть файл

    for ( i = 0; i < N; i ++ )
        A[i] = A[i] * 2;

    fp = fopen( "output.dat", "w" );    // открыть файл на запись
    for ( i = 0; i < N; i ++ )          // вывести массив в файл
        fprintf ( fp, "%d\n", A[i] );    // в столбик
    fclose ( fp );
}
```

В отличие от предыдущих, эта программа выдает результаты не на экран, а в файл **output.dat** в текущем каталоге.



Массив неизвестного размера

Задача. В файле **input.dat** записаны в два столбика пары чисел (**x**, **y**). Записать в файл **output.dat** в столбик суммы **x+y** для каждой пары.

Сложность этой задачи состоит в том, что мы не можем прочитать все данные сразу в память, обработать их и записать в выходной файл. Не можем потому, что не знаем, сколько пар чисел в массиве. Конечно, если известно, что в файле, скажем, не более 200 чисел, можно выделить массив «с запасом», прочитать столько данных, сколько нужно, и работать только с ними. Однако в файле могут быть миллионы чисел и такие массивы не поместятся в памяти.

Однако, если подумать, становится понятно, что для вычисления суммы каждой пары нужны только два числа, а остальные мы можем не хранить в памяти. Когда вычислили их сумму, ее также не надо хранить в памяти, а можно сразу записать в выходной файл. Поэтому будем использовать такой **алгоритм**:

- 1) открыть два файла, один на чтение (с исходными данными), второй – на запись;
- 2) попытаться прочитать два числа в переменные **x** и **y**; если это не получилось (нет больше данных или неверные данные), закончить работу;
- 3) сложить **x** и **y** и записать результат в выходной файл;
- 4) перейти к шагу 2.

Для того, чтобы определить, успешно ли закончилось чтение, мы будем использовать тот факт, что функция **fscanf** (как и **scanf**) возвращает количество успешно считанных чисел. За один раз будем читать сразу два числа, **x** и **y**. Если все закончилось успешно, функция **fscanf** возвращает значение 2 (обе переменных прочитаны). Если результат этой функции меньше двух, данные закончились или неверные.

Заметим, что надо работать одновременно с двумя открытыми файлами, поэтому в памяти надо использовать два указателя на файлы, они обозначены именами **fin** и **fout**. Для сокращения записи ошибки при открытии файлов не обрабатываются.

```
#include <stdio.h>
main()
{
    int n, x, y, sum;
    FILE *fin, *fout;    // указатели на файлы
    fin = fopen( "input.dat", "r" );    // открыть файл на чтение
    fout = fopen( "output.dat", "w" ); // открыть файл на запись
    while ( 1 ) {
        n = fscanf ( fin, "%d%d", &x, &y );
        if ( n < 2 ) break; // данные ошибочны или нет больше данных
        sum = x + y;
        fprintf ( fout, "%d\n", sum );
    }
    fclose ( fout ); // закрыть файлы
    fclose ( fin );
}
```

В программе используется бесконечный цикл **while**. Программа выходит из него тогда, когда данные в файле закончились.



Работа с двоичными файлами

Двоичные файлы отличаются от текстовых тем, что в них записана информация во внутреннем машинном представлении. Двоичный файл нельзя просмотреть на экране (вернее, можно просмотреть, но очень сложно понять). Но есть и преимущества – из двоичных файлов можно читать сразу весь массив в виде единого блока. Также можно записать весь массив или его любой непрерывный кусок за одну команду.

При открытии двоичного файла вместо режимов **"r"**, **"w"** и **"a"** используют соответственно **"rb"**, **"wb"** и **"ab"**. Дополнительная буква **"b"** указывает на то, что файл двоичный (от английского слова *binary* – двоичный). Приведем решение одной задачи, которую мы уже разбирали ранее.

Задача. Ввести массив из 10 целых чисел из двоичного файла **input.dat**, умножить каждый элемент на 2 и вывести в двоичный файл **output.dat**.

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, n, A[N];
    FILE *fp;    // указатель на файл
    fp = fopen( "input.dat", "rb" ); // открыть двоичный файл на чтение
```

```
n = fread ( A, sizeof(int), N, fp ); // читаем весь массив
if ( n < N ) {                               // обработка ошибки
    printf("Не хватает данных в файле");
    break;
}
fclose ( fp );                                // закрыть файл

for ( i = 0; i < N; i ++ )
    A[i] = A[i] * 2;

fp = fopen( "output.dat", "wb" ); // открыть двоичный файл на запись
fwrite ( A, sizeof(int), N, fp ); // записать весь массив
fclose ( fp );                        // закрыть файл
}
```

Для чтения из двоичного файла используется функция **fread**, которая принимает 4 параметра:

- **адрес области в памяти**, куда записать прочитанные данные (в данном случае это адрес первого элемента массива **A**, который обозначается как **&A[0]** или просто **A**);
- **размер одного элемента** данных (лучше сделать так, чтобы машина сама определила его, например, в нашем случае – **sizeof(int)** – размер целого числа. Хотя в *Dev-C++* целое число занимает 4 байта, в других системах программирования это может быть не так; наша программа будет работать и в этом случае, то есть станет *переносимой* на другую платформу);
- **количество элементов** данных в массиве (**N**);
- **указатель на открытый файл**, откуда читать данные (**fp**).

Функция **fread** возвращает количество успешно прочитанных элементов массива – ее возвращаемое значение можно использовать для обработки ошибок. Если функция **fread** вернула значение, меньшее, чем **N**, в файле не хватает данных.

Для записи массива в двоичный файл используется функция **fwrite** с такими же параметрами; она возвращает количество успешно записанных элементов.

Преимущество этого способа состоит в том, что массив читается и записывается сразу единым блоком. Это значительно увеличивает скорость записи на диск (в сравнении с выводом в текстовый файл отдельно каждого элемента).

Простой поиск в массиве

Во многих задачах требуется последовательно перебрать все элементы массива и найти нужные нам. Мы рассмотрим четыре таких задачи:

- поиск одного заданного элемента;
- вывод всех элементов, которые удовлетворяют заданному условию;
- формирование нового массива из всех отобранных элементов;
- поиск минимального (максимального) элемента.

Все эти задачи решаются с помощью цикла, в котором перебираются все элементы массива от начального (0-ого) до конечного (**N-1**-ого) элемента. Такой поиск называется *линейным*, поскольку все элементы просматриваются последовательно один за другим.



Поиск одного элемента

Задача. Определить, есть ли в массиве элемент с заданным значением **x**, и, если он есть, найти его номер.

Если нет никакой информации о расположении элементов массива, то применяется линейный поиск, основная идея которого – последовательно просматривать массив, пока не будет обнаружено совпадение или не будет достигнут конец массива. Это реализует следующая простая программа:

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, X, A[N];

    int success = 0;          // переменная-флаг, флаг сброшен

    // здесь нужно ввести массив и X
    for ( i = 0; i < N; i ++ )

        if ( A[i] == X ) {    // если нашли, то...
            success = 1;      // установить флаг
            break;            // выйти из цикла
        }

    if ( success )
        printf ( "A[%d] = %d", i, A[i] );
    else
        printf ( "Элемент %d не найден.", X );
}
```

Чтобы определить ситуацию, когда элемент не найден, нам надо ввести специальную переменную **success**, которая устанавливается в 1, если элемент найден, и остается равной нулю, если в массиве нет нужного элемента. Такая переменная называется **флагом**, флаг может быть установлен (равен 1) или сброшен (равен нулю).

Для линейного поиска в худшем случае мы имеем **N** сравнений. Понятно, что для ускорения поиска надо сначала как-то упорядочить данные, в этом случае можно сделать поиск эффективным.



Поиск всех элементов, соответствующих условию

Задача. Определить, сколько в массиве положительных элементов и вывести их на экран.

Для решения этой задачи вводим **счетчик** – специальную переменную, значение которой будет увеличиваться на единицу, когда мы нашли очередной положительный элемент.

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, A[N], count = 0; // count – счетчик положительных элементов

    // здесь нужно ввести массив

    for ( i = 0; i < N; i ++ )    // цикл по всем элементам массива
        if ( A[i] > 0 ) {        // если нашли положительный, ...
            count ++;            // увеличиваем счетчик и ...
            printf ("%d ", A[i]); // выводим элемент (если нужно)
        }
```

```

    }
    printf ("\n В массиве %d положительных чисел", count);
}

```

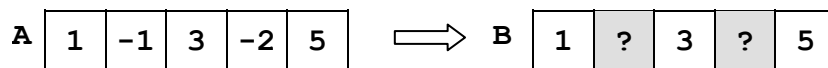
Формирование массива по заданному условию

Задача. Сформировать новый массив **В**, включив в него все положительные элементы исходного массива **А**, и вывести его на экран.

Пусть есть массив **А[N]**. Надо выбрать из него все положительные элементы и записать их в новый массив, который и будет дальше использоваться.

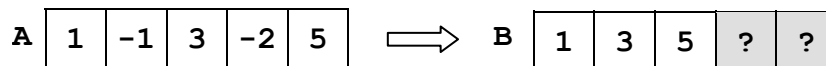
Сначала надо определить, сколько места в памяти надо выделить для массива **В**. В «худшем» случае все элементы в массиве **А** будут положительными и войдут в массив **В**, поэтому массив **В** должен иметь такой же размер, что и массив **А**.

Можно предложить такой способ: просматривать весь массив **А**, и если для очередного элемента **А[i] > 0**, его значение копируется в **В[i]**.



Однако в этом случае использовать такой массив **В** очень сложно, потому что нужные элементы стоят не подряд.

Есть более красивый способ. Объявляем временную переменную-счетчик **count**, в которой будем хранить количество найденных положительных элементов. Сначала она равна нулю. Если нашли очередной положительный элемент, то ставим его в ячейку **В[count]** и увеличиваем счетчик. Таким образом, все нужные элементы стоят в начале массива **В**.



```

#include <stdio.h>
const int N = 10;
main()
{
    int i, A[N], B[N], count = 0;

    // здесь нужно ввести массив А

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

    printf("\n Результат:\n");
    for ( i = 0; i < count; i ++ )
        printf("%d ", B[i]);
}

```

Можно сделать и так:
 if (A[i] > 0)
 B[count++] = A[i];

Обратите внимание, что переменная в последнем цикле изменяется от **0** до **count-1** (а не до **N-1**) так что на экран выводятся только реально используемые (а не все) элементы массива **В**.



Минимальный элемент

Задача. Найти и вывести на экран минимальный элемент в массиве **A**.

Для решения задачи надо выделить в памяти ячейку (переменную) для хранения найденного минимального значения. Сначала мы записываем в эту ячейку первый элемент (**A[0]**). Затем берем следующий элемент и сравниваем его с минимальным. Если он меньше минимального, записываем его значение в ячейку минимального элемента. И так далее. Когда мы рассмотрим последний элемент в массиве, в дополнительной ячейке памяти будет минимальное значение из всех элементов массива.

Заметим, что перебор в цикле начинается с элемента с номером 1 (а не 0), поскольку начальный элемент мы рассмотрели отдельно.

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, A[N], min;
    // здесь нужно ввести массив A

    min = A[0]; // предполагаем, что A[0] - минимальный
    for ( i=1; i<N; i++ ) // цикл по всем элементам с A[1] до A[N-1]
        if ( A[i] < min ) // если A[i] меньше min, ...
            min = A[i]; // запомнить A[i] как минимальный

    printf("\n Минимальный элемент %d", min);
}
```

Чтобы найти максимальный элемент, достаточно изменить условие в заголовке условного оператора на обратное (**A[i] > min**). Конечно, вспомогательную переменную в этом случае лучше (но не обязательно!) назвать **max**.

Теперь можно усложнить задачу и найти еще и номер минимального элемента.

Задача. Найти и вывести на экран минимальный элемент в массиве **A** и его номер.

Напрашивается такое решение: завести еще одну переменную, в которой хранить номер минимального элемента. Если мы нашли новый минимальный элемент, то в одну переменную записали его значение, а во вторую – его номер.

Тем не менее, можно обойтись одной дополнительной переменной. Дело в том, что по номеру элемента можно легко найти его значение в массиве. На этом основана приведенная ниже программа. Теперь мы запоминаем (в переменной **nMin**) не значение минимального элемента, а только его номер.

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, A[N], nMin; // nMin - номер минимального элемента
    // здесь нужно ввести массив A

    nMin = 0; // считаем, что A[0] - минимальный
    for ( i = 1; i < N; i++ ) // цикл по всем остальным элементам
        if ( A[i] < A[nMin] ) // если A[i] < A[nMin], то...
            nMin = i; // запомнить i

    printf("\n Минимальный элемент A[%d]=%d", nMin, A[nMin]);
}
```

}



Перестановка элементов массива



О кувшине и вазе

Представьте, что в вазе для цветов налито молоко, а в кувшине – вода с удобрениями. Как привести все в порядок? Надо использовать третью емкость такого же (или большего) объема. Сначала переливаем в нее воду из кувшина (или молоко из вазы, все равно), затем в пустой кувшин переливаем молоко (или в вазу – воду), а затем из третьей емкости переливаем воду в вазу (или, соответственно, молоко в кувшин).

Так же и в программировании: чтобы поменять местами значения двух ячеек в памяти, надо использовать временную переменную¹. Пусть даны ячейки **a** и **b**, содержащие некоторые значения. После выполнения следующих команд их значения поменяются:

```
a = 4; b = 6;
c = a;    // a = 4; b = 6; c = 4;
a = b;    // a = 6; b = 6; c = 4;
b = c;    // a = 6; b = 4; c = 4;
```

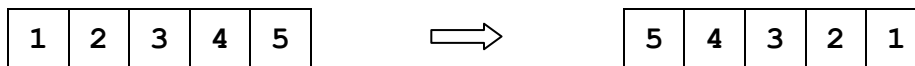
Эта цепочка операторов присваивания особая:

- она начинается и заканчивается временной переменной **c**;
- следующий оператор начинается с той переменной, на которую закончился предыдущий



Перестановка наоборот (инверсия)

Инверсия (от английского *inverse* – обратный) – это такая перестановка, когда первый элемент становится последним, второй – предпоследним и т.д.



Эта простая задача имеет один подводный камень. Пусть размер массива **N**. Тогда элемент **A[0]** надо переставить с **A[N-1]**, **A[1]** с **A[N-2]** и т.д. Заметим, что в любом случае сумма индексов переставляемых элементов равна **N-1**, поэтому хочется сделать цикл от **0** до **N-1**, в котором переставить элементы **A[i]** с **A[N-1-i]**. Однако при этом вы обнаружите, что массив не изменился.

Обратим внимание, что перестановка затрагивает одновременно и первую, и вторую половину массива. Поэтому сначала все элементы будут переставлены правильно, а затем (когда **i > N/2**) будут возвращены на свои места. Правильное решение – делать перебор, при котором переменная цикла доходит только до середины массива.

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, A[N], c;
    // здесь нужно ввести массив A
    for ( i = 0; i < N/2; i ++ ) // перебор только до середины!
    {
        c = A[i];                // цепочка для перестановки
        A[i] = A[N-1-i];         // A[i] и A[N-1-i]
```

¹ Существуют и немного более сложные способы, позволяющие обойтись всего двумя ячейками

```

    A[N-1-i] = c;
}

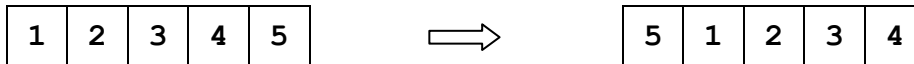
printf("\n Результат:\n");
for ( i = 0; i < N; i ++ )
    printf("%d ", A[i]);
}

```



Циклический сдвиг

При циклическом сдвиге (вправо) первый элемент переходит на место второго, второй на место третьего и т.д., а последний элемент – на место первого.



Для выполнения циклического сдвига нам будет нужна одна временная переменная – в ней мы сохраним значение последнего элемента, пока будем переставлять остальные. Обратите внимание, что мы начинаем с конца массива, иначе массив просто заполнится первым элементом. Первый элемент ставится отдельно – копированием из временной переменной.

```

#include <stdio.h>
const int N = 10;
main()
{
    int i, A[N], c;

    // здесь нужно ввести массив A

    c = A[N-1];    // запомнить последний элемент
    for ( i=N-1; i>0; i-- ) // цикл с уменьшением i
        A[i] = A[i-1];
    A[0] = c;      // первый элемент ставим отдельно

    printf("\n Результат:\n");
    for ( i=0; i<N; i++ )
        printf("%d ", A[i]);
}

```



Сортировка массивов

Сортировка – это расстановка элементов некоторого списка в заданном порядке.

Существуют разные виды сортировки (по алфавиту, по датам и т.д.), они отличаются лишь процедурой сравнения элементов. Мы рассмотрим простейший вариант сортировки – расстановку элементов массива в порядке возрастания.

Программисты придумали множество методов сортировки. Они делятся на две группы:

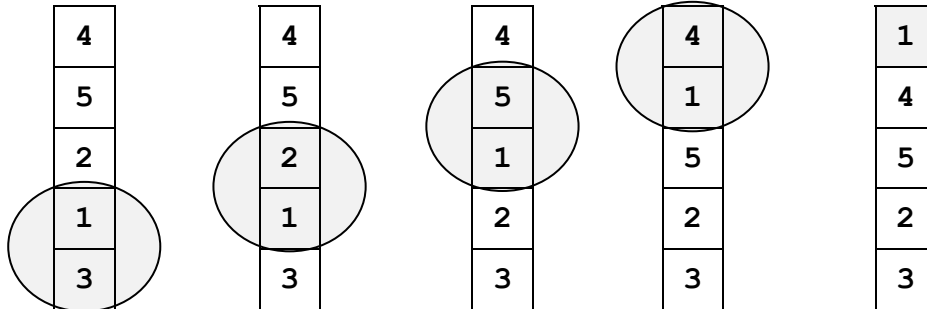
- понятные, но не эффективные
- эффективные, но непонятные (быстрая сортировка и т.п.).

Пока мы будем изучать только методы из первой группы, которых хватает для простых задач (когда размер массива не более 1000).

Метод пузырька

Название этого метода произошло от известного физического явления – пузырек воздуха в воде поднимается вверх. В этом методе сначала поднимается «наверх» (к началу массива) самый «легкий» элемент (минимальный), затем следующий и т.д.

Сначала сравниваем последний элемент с предпоследним. Если они стоят неправильно, то меняем их местами. Далее так же рассматриваем следующую пару элементов и т.д. 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



При следующем проходе наша задача – поставить на место элемент $A[1]$. Делаем это так же, но уже не рассматриваем $A[0]$, который стоит на своем месте. Сделав $N-1$ проходов, мы установим на место элементы с $A[0]$ по $A[N-2]$. Это значит, что последний элемент, $A[N-1]$, уже тоже стоит на своем месте (другого у него нет).

```
#include <stdio.h>
const int N = 10;
main()
{
    int i, j, A[N], c;

    // здесь надо ввести массив A

    for ( i = 0; i < N-1; i ++ ) // достаточно поставить N-1 элементов
        for ( j = N-2; j >= i; j -- ) // идем с конца массива в начало
            if ( A[j] > A[j+1] ) // если они стоят неправильно, ...
                {
                    c = A[j]; A[j] = A[j+1]; // переставить A[j] и A[j+1]
                    A[j+1] = c;
                }

    printf("\n Отсортированный массив:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);
}
```

Метод пузырька работает медленно, особенно на больших массивах. Можно показать, что при увеличении размера массива в 10 раз время выполнения программы увеличивается в 100 раз (метод имеет порядок N^2). К сожалению, все простые алгоритмы сортировки имеют такой (квадратичный) порядок.

Метод выбора минимального элемента

Еще один недостаток метода пузырька состоит в том, что приходится слишком часто переставлять местами соседние элементы. Этого можно избежать, если использовать метод вы-

бора минимального элемента. Он заключается в следующем. Ищем в массиве минимальный элемент и ставим его на первое место. Затем из оставшихся элементов также ищем минимальный и ставим на следующее место и т.д.

В сравнении с методом пузырька, этот метод требует значительно меньше перестановок элементов (в худшем случае $N-1$). Он дает значительный выигрыш, если перестановки сложны и занимают много времени.

```
#include <stdio.h>

const int N = 10;

main()
{
    int i, j, nMin, A[N], c;
    // здесь нужно ввести массив A

    for ( i = 0; i < N-1; i ++ )
    {
        nMin = i;    // ищем минимальный, начиная с A[i]
        for ( j = i+1; j < N; j ++ )
            if ( A[j] < A[nMin] )
                nMin = j;

        if ( nMin != i ) // если минимальный не стоит на своем месте,...
        {
            c = A[i]; A[i] = A[nMin]; // ставим его на место
            A[nMin] = c;
        }
    }

    printf("\n Отсортированный массив:\n");
    for ( i = 0; i < N; i ++ )
        printf("%d ", A[i]);
}
```



Двоичный поиск в массиве

Пусть элементы массива **A** уже расставлены по возрастанию и требуется найти элемент, равный **x**, среди элементов с номерами от **L** до **R**. Для этого использую следующую идею: выбираем средний элемент между **L** и **R**, он имеет номер $m = (L+R) / 2$, где деление выполняется нацело. Сравним его с искомым **x**. Если он равен **x**, мы нашли то, что искали. Если **x** меньше **A[m]**, надо искать дальше между **A[L]** и **A[m]**, если **x** больше **A[m]**, дальше ищем между **A[m]** и **A[R]**.

```

#include <stdio.h>
const int N = 10;
main()
{
    int L = 0, R = N-1, m, A[N],
        flag = 0;    // переменная-флаг, нашли (1) или нет (0)

    // здесь надо ввести массив A
    // и отсортировать его по возрастанию

    printf("Введите искомый элемент\n");
    scanf( "%d", &x );

    while ( L <= R ) {
        m = (L + R) / 2;    // середина интервала
        if ( A[m] == x ) {  // нашли нужный элемент
            flag = 1;      // установить флаг
            break;         // выйти из цикла
        }
        if ( x < A[m] ) R = m - 1; // сужаем границы области поиска
        else L = m + 1;
    }

    if ( flag )
        printf ( "Нашли: A[%d]=%d", m, A[m] );
    else printf ( "Такого элемента нет" );
}

```

Переменная **flag** служит для того, чтобы определить, нашли мы нужный элемент или нет. Если нашли элемент, равный **x**, надо присвоить этой переменной значение 1 и выйти из цикла. При этом в переменной **m** остается номер найденного элемента.

Если массив маленький, то скорость двоичного поиска незначительно отличается от линейного. Представим себе, что размер массива — 1000000 элементов и нужный нам элемент стоит в конце массива (это самый худший вариант для линейного поиска).

Размер массива, <i>N</i>	Число сравнений	
	Линейный поиск	Двоичный поиск
10	≤ 10	≤ 4
1 000	$\leq 1\,000$	≤ 10
1 000 000	$\leq 1\,000\,000$	≤ 20
<i>N</i>	$\leq N$	$\leq \log_2 N$

Таким образом, чем больше элементов в массиве, тем выгоднее использовать двоичный поиск, поскольку число операций возрастает как логарифм **N**, то есть медленнее, чем увеличивается размер массива. Его недостатком является то, что элементы должны быть заранее отсортированы. Двоичный поиск используется при поиске информации в больших базах данных.



Массивы в процедурах и функциях

Массивы, так же как и простые переменные, можно передавать в процедуры и функции в качестве параметров. Рассмотрим, например, функцию, вычисляющую сумму элементов массива. Желательно сделать ее так, чтобы в нее можно было передавать массивы любого размера, и

она всегда правильно вычисляла результат. Для этого функция должна знать (или определить) размер массива. В языке Си функции не могут самостоятельно определять размер массива, поэтому он должен быть обязательно одним из параметров.

```
int Sum ( int A[], int N ) // A[] - параметр-массив
{                          // N - размер массива
    int i, sum;
    sum = 0;
    for ( i = 0; i < N; i ++ )
        sum += A[i];
    return sum;
}
```

Обратите внимание, что в заголовке функции размер массива указан отдельно, нельзя объявлять массив-параметр как **A[N]**, а только как **A[]**. С другой стороны такая запись возможна только в заголовках функций, поскольку при этом не надо выделять новую память под массив. Объявлять локальный или глобальный массив, не указав явно его размер, нельзя.

Для вызова приведенной функции в параметрах надо указать название массива (без скобок) и его размер.

```
main()
{
    int A[20], B[30], s;
    // здесь надо ввести массивы A и B
    s = Sum(A,20); // вычисляем сумму элементов массива A
    printf("Сумма массива A %d, массива B %d", s,
        Sum(B,30) ); // вычисляем сумму B прямо при выводе
}
```

2. Символьные строки

Что такое символьная строка?

Понятно, что символьная строка – это последовательность символов. Мы будем рассматривать строки, в которых на каждый символ отводится 1 байт. В этом случае можно использовать $2^8=256$ различных символов. Каждый символ имеет свой код (от 0 до 255), эти коды определяются по специальной таблице.

Строка, как и другие переменные, записывается в память, причем компьютеру все равно, какие данные записаны – для него это набор байтов. Как же определить, где заканчивается строка? Есть два решения:

- 1) хранить длину строки в отдельной ячейке (как в языке Паскаль);
- 2) выбрать один особый символ, который будет обозначать конец строки, причем в середине строки этот символ не может встречаться.

В языке Си принят второй подход.

Символьная строка – это последовательность символом, которая заканчивается **символом с кодом 0**.

Символ с кодом ноль не имеет никакого изображения, в программе его записывают как `'\0'`.

Символ с кодом ноль (обозначается как `'\0'`) и цифра ноль (обозначается `'0'`, имеет код 48) – это два разных символа.

Объявление и инициализация

Строка представляет собой массив символов, поэтому и объявляется она именно как массив:

```
char s[80];
```

Однако строка отличается от массива тем, что она заканчивается символом с кодом 0 – признаком окончания строки, поэтому

Если массив символов будет использоваться как строка, надо выделять на 1 байт больше памяти.

При выделении памяти глобальные переменные заполняются нулями, а локальные содержат «мусор». Начальное значение строки можно задать при объявлении в двойных кавычках после знака равенства:

```
char s[80] = "Привет, Вася!";
```

Символы в кавычках будут записаны в начало массива `s`, а затем – признак окончания строки `'\0'`. Оставшиеся символы не меняются, и в локальных строках там будет «мусор». Можно написать и так

```
char s[] = "Привет, Вася!";
```

В этом случае компилятор подсчитает символы в кавычках, выделит памяти на 1 байт больше и занесет в эту область саму строку и завершающий ноль. Аналогично можно выделить память на указатель:

```
char *s = "Привет, Вася!";
```

Результат – тот же самый, что и в предыдущем случае, но теперь **s** – это указатель (переменная, в которой хранится адрес ячейки в памяти), и с ним можно работать так же, как с обычным указателем (присваивать, изменять и т.п.). Если строка не будет изменяться во время работы программы, то можно объявить константу (постоянную строку) так:

```
const char PRIVET[] = "Привет, Вася!";
```



Стандартный ввод и вывод

Для ввода и вывода строк с помощью функций **scanf** и **printf** используется специальный формат **%s**:

```
#include <stdio.h>
main()
{
    char Name[50];
    printf("Как тебя зовут? ");
    scanf("%s", Name);
    printf("Привет, %s!", Name);
}
```

Заметьте, что в функцию **scanf** надо передать просто имя строки (без знака **&**), ведь имя массива является одновременно адресом его начального элемента.

Однако у функции **scanf** есть одна особенность: она заканчивает ввод, встретив первый пробел. Если вы на вопрос в предыдущем примере ввели **"Вася Пупкин"**, то увидите надпись **"Привет, Вася!"** вместо ожидаемого **"Привет, Вася Пупкин!"**. Если надо ввести всю строку целиком, включая пробелы (то есть до нажатия на клавишу **Enter**), придется делать иначе, заменив вызов **scanf** на более простой:

```
gets ( s );
```

Название этой функции происходит от английских слов *get string* – получить строку.

Для вывода строки на экран можно (кроме **printf**) использовать и функцию **puts**, которая после вывода строки еще и дает команду перехода на новую строку. В примере значение строки **Name** будет напечатано на следующей строчке экрана.

```
#include <stdio.h>
main()
{
    char Name[50] = "Вася!";
    puts( "Привет," );
    puts ( Name );
}
```

Задача. Ввести символьную строку и заменить в ней все буквы **'А'** на буквы **'Б'**.

Будем рассматривать строку как массив символов. Надо перебрать все элементы массива, пока мы не встретим символ **'\0'** (признак окончания строки) и, если очередной символ – это буква **'А'**, заменить его на **'Б'**. Для этого используем цикл **while**, потому что мы заранее не знаем длину строки. Условие цикла можно сформулировать так: «пока не конец строки».

```

#include <stdio.h>
main()
{
    char a[80];
    int i;

    printf( "\n Введите строку \n" );
    gets ( s );

    i = 0; // начать с первого символа, s[0]
    while ( s[i] != '\0' ) // пока не достигли конца строки
    {
        if ( s[i] == 'A' ) // если очередной символ - 'A', ...
            s[i] = 'Б'; // меняем его на 'Б'
        i ++; // переходим к следующему символу
    }

    puts ( "Результат:\n" );
    puts ( a );
}

```

Заметьте, что

Одиночный символ записывается в апострофах, а символьная строка – в кавычках.

При выводе строк с помощью функции **printf** часто применяется форматирование. После знака % в формате указывается размер поля для вывода строки. Перед этим числом можно также поставить знак минус, что означает «прижать к левому краю поля».

Пример вывода	Результат	Комментарий
<code>printf("[%s]", "Вася");</code>	<code>[Вася]</code>	Минимальное число позиций.
<code>printf("[%6s]", "Вася");</code>	<code>[Вася]</code>	6 позиций, выравнивание вправо.
<code>printf("[% -6s]", "Вася");</code>	<code>[Вася]</code>	6 позиций, выравнивание влево.
<code>printf("[%2s]", "Вася");</code>	<code>[Вася]</code>	Строка не помещается в заданные 2 позиции, поэтому область вывода расширяется.

Работа с файлами

В реальной ситуации требуется обрабатывать очень много строк, которые чаще всего находятся в файле, причем их количество заранее неизвестно. Однако, если для обработки одной строки нам не требуется знать остальные, можно использовать способ, который мы применяли при работе с массивами данных неизвестного размера. В данном случае мы будем читать очередную строку из файла, обрабатывать ее и сразу записывать в выходной файл (если это требуется).

Работа с файлами имеет несколько особенностей. Во-первых, для чтения строки можно использовать функцию **fscanf**. Однако эта функция читает только одно слово и останавливается на первом пробеле. Поэтому

функция **fscanf** применяется тогда, когда надо читать файл по словам.

Вот пример чтения слова из открытого файла с указателем **fp**:

```
#include <stdio.h>
main()
{
    char s[80];
    FILE *fp;
    fp = fopen ( "input.dat", "r" );
    fscanf ( fp, "%s", s );
    printf ( "Первое слово файла - %s", s );
    fclose ( fp );
}
```

Если надо читать всю строку с пробелами, используют функцию **fgets**. Она принимает три параметра:

- **имя символьной строки**, в которую записать данные;
- **максимальную длину** этой строки; функция не допускает выхода за границы строки; если строка в файле длиннее, чем можно записать в память, читается только начальная часть, а остальное – при следующих вызовах **fgets**;
- **указатель на файл**.

Если функция **fgets** не может прочитать строку из файла (например, если нет больше строк), то она возвращает в качестве результата специальное значение **NULL**. Это свойство можно использовать для обработки ошибок и прекращения ввода данных.

```
#include <stdio.h>
main()
{
    char s[80];
    FILE *fp;
    fp = fopen ( "input.dat", "r" );
    if ( NULL == fgets ( s, 80, fp ) )
        printf ( "Не удалось прочитать строку" );
    else
        printf ( "Первая строка файла - %s", s );
    fclose ( fp );
}
```

Функция **fgets** читает строку из файла, пока не случится одно из двух событий:

- встретится символ перехода на новую строку '**\n**';
- прочитано столько символов, что они заняли всю строку (с учетом последнего нуля), например, в нашем случае она остановится, если прочтет 79 символов.

В конце строки будет поставлен символ '**\0**'. Кроме того, если был найден символ перехода на новую строку '**\n**', он сохраняется и в строке **s**.

Задача. В каждой строке файла **input.dat** заменить все буквы '**А**' на '**Б**' и вывести измененный текст в файл **output.dat**.

Обратите внимание, что в этой задаче файл может быть любой длины. Но мы можем обрабатывать строки последовательно одну за другой, так как для обработки одной строки не нужны предыдущие и следующие.

Для чтения из файла используем цикл **while**. Он заканчивает работу, если функция **fgets** вернет значение **NULL**, то есть все строки обработаны.


```
#include <stdio.h>
main()
{
    char s[80];
    int i;
    FILE *fin, *fout;

    fin = fopen ( "input.dat", "r" );
    fout = fopen ( "output.dat", "w" );

    while ( NULL != fgets ( s, 80, fin ) ) // читаем строку
    {
        i = 0; // начинаем с s[0]
        while ( s[i] != '\0' ) { // пока не конец строки
            if ( s[i] == 'A' ) s[i] = 'B'; // меняем символ
            i ++; // переходим к следующему символу
        }

        fprintf ( fout, "%s", s ); // выводим строку в файл
    }

    fclose ( fin );
    fclose ( fout );
}
```

Обратите внимание, что мы не поставили символ перехода на новую строку при вызове функции **fprintf**. Это связано с тем, что при чтении функция **fgets** сохраняет символ **'\n'** в конце каждой строки (кроме последней), поэтому строки будут выведены в выходной файл так же, как они были записаны в исходном файле.



Функции для работы со строками

В языке Си есть достаточно много специальных функций, которые работают со строками – последовательностями символов с нулевым символом на конце. Для использования этих функций надо включить в программу заголовочный файл

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

Многие из этих функций достаточно опасны при неправильном использовании, ведь они не проверяют, достаточно ли выделено памяти для копирования, перемещения или другой операции, единственным признаком окончания строки для них является символ **'\0'**.



Длина строки – strlen

Это самая простая функция, которая определяет, сколько символов в переданной ей строке (не считая **'\0'**). Ее имя происходит от английских слов *string length* (длина строки).

```
#include <stdio.h>
#include <string.h>
main()
{
    int len;
    char s[] = "Prodigy";
    len = strlen(s);

    printf ( "Длина строки %s равна %d", s, len );
}
```

В этом примере функция определит, что длина строки равна 7. Теперь рассмотрим более сложную задачу.

Задача. В текстовом файле `input.dat` записаны строки текста. Вывести в файл `output.dat` в столбик длины этих строк.

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

main()
{
    char s[80];
    FILE *fin, *fout;

    fin = fopen ( "input.dat", "r" );
    fout = fopen ( "output.dat", "w" );

    while ( NULL != fgets(s, 80, fin) ) // читаем строку s
    {
        fprintf(fout, "%d\n", strlen(s)); // выводим ее длину в файл
    }

    fclose ( fin );
    fclose ( fout );
}
```

Несмотря на то, что с первого взгляда программа написана верно, числа в файле будут на единицу больше, чем длины строк (кроме последней строки). Вспомнив предыдущий материал, объясните это. Далее будет показано, как получить точный результат.



Сравнение строк – `strcmp`

Для сравнения двух строк используют функцию `strcmp` (от английских слов *string comparison* – сравнение строк). Функция возвращает ноль, если строки равны (то есть «разность» между ними равна нулю) и ненулевое значение, если строки различны. Сравнение происходит по кодам символов, поэтому функция различает строчные и заглавные буквы – они имеют разные коды.

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

main()
{
    char s1[] = "Вася",
        s2[] = "Петя";

    if ( 0 == strcmp(s1,s2) )
        printf("Строки %s и %s одинаковы", s1, s2);
    else printf("Строки %s и %s разные", s1, s2);
}
```

Если строки не равны, функция возвращает «разность» между первой и второй строкой, то есть *разность кодов первых различных символов*. Эти числа можно использовать для сортировки строк – если «разность» отрицательна, значит первая строка «меньше» второй, то есть стоит за ней в алфавитном порядке. В таблице показано несколько примеров (код буквы 'А' равен 65, код буквы 'В' – 66, код буквы 'С' – 67).

s1	s2	результат <code>strcmp(s1, s2)</code>
AA	AA	0
AB	AAB	'B' - 'A' = 66 - 65 = 1
AB	CAA	'A' - 'C' = 65 - 67 = -2
AA	AAA	'\0' - 'A' = -65

Задача. Ввести две строки и вывести их в алфавитном порядке.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[80], s2[80];
    printf ("Введите первую строку");
    gets(s1);
    printf ("Введите вторую строку");
    gets(s2);

    if ( strcmp(s1,s2) <= 0 )
        printf("%s\n%s", s1, s2);
    else printf("%s\n%s", s2, s1);
}
```

Иногда надо сравнить не всю строку, а только первые несколько символов. Для этого служит функция **strncmp** (с буквой **n** в середине). Третий параметр этой функции – количество сравниваемых символов. Принцип работы такой же – она возвращает нуль, если заданное количество первых символов обеих строк одинаково.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[80], s2[80];
    printf ("Введите первую строку");
    gets(s1);
    printf ("Введите вторую строку");
    gets(s2);

    if ( 0 == strncmp(s1, s2, 2) )
        printf("Первые два символа %s и %s одинаковы", s1, s2);
    else
        printf("Первые два символа %s и %s разные", s1, s2);
}
```

Один из примеров использования функции **strcmp** – проверка пароля. Составим программу, которая спрашивает пароль и, если пароль введен неверно, заканчивает работу, а если верно – выполняет какую-нибудь задачу.

Задача. Составить программу, которая определяет, сколько цифр в символьной строке. Программа должна работать только при вводе пароля «куку».

```

#include <stdio.h>
#include <string.h>
main()
{
    char pass[] = "куку", // правильный пароль
          s[80];          // вспомогательная строка
    int i, count = 0;

    printf ("Введите пароль ");
    gets(s);
    if ( strcmp ( pass, s ) != 0 )
    {
        printf ( "Неверный пароль" );
        return 1;          // выход по ошибке, код ошибки 1
    }

    printf ("Введите строку");
    gets(s);

    i = 0;
    while ( s[i] != '\0' ) {
        if ( s[i] >= '0' && s[i] <= '9' )
            count ++;
    }

    printf("\nНашли %d цифр", count);
}

```

В этой программе использован тот факт, что коды цифр расположены в таблице символов последовательно от '0' до '9'. Поэтому можно использовать двойное неравенство, а не сравнивать текущий символ `s[i]` с каждой из цифр. Обратите внимание на разницу между символами '\0' (символ с кодом 0, признак конца строки) и '0' (символ с кодом 48, цифра 0). Переменная `count` работает как счетчик.



Копирование строк

Часто надо записать новое значение в строку или скопировать информацию из одной строки в другую. Функции копирования принадлежат к числу «опасных» – они могут вызвать серьезную ошибку, если произойдет **выход за границы массива**. Это бывает в том случае, если строка, в которую копируется информация, имеет недостаточный **размер** (под нее выделено мало места в памяти).

В копировании участвуют две строки, они называются «источник» (строка, откуда копируется информация) и «приемник» (куда она записывается или добавляется).

При копировании строк надо проверить, чтобы для строки-приемника было выделено достаточно места в памяти

Простое копирование выполняет функция `strcpy`. Она принимает два аргумента: сначала строка-приемник, потом – источник (порядок важен!).

```

char s1[50], s2[10];
gets(s1);
strcpy ( s2, s1); // s2 (приемник) <- s1 (источник)
puts ( s2 );

```

Этот фрагмент программы является «опасным» с точки зрения выхода за границы строки. В строку `s1` можно безопасно записать не более 49 символов (плюс завершающий ноль). Поэтому

если с клавиатуры будет введена строка длиннее 49 символов, при записи ее в память произойдет выход за границы строки **s1**. Строка **s2** может принять не более 9 символов, поэтому при большем размере **s1** произойдет выход за границы строки **s2**.

Поскольку реально функции передается адрес начала строки, можно заставить функцию начать работу любого символа, а не только с начала строки. Например, следующая строка копирует строку **s2** в область памяти строки **s1**, которая начинается с ее 6-ого символа, оставив без изменения первые пять:

```
strcpy ( s1+5, s2 );
```

При этом надо следить, чтобы не выйти за границу массива. Кроме того, если до выполнения этой операции в строке **s1** меньше 5 символов, фокус не удастся.

Еще одна функция позволяет скопировать только заданное количество символов, она называется **strncpy** и принимает в третьем параметре количество символов, которые надо скопировать. Важно помнить, что эта функция **НЕ записывает завершающий ноль**, а только копирует символы (в отличие от нее **strcpy** всегда копирует завершающий ноль). Функция **strncpy** особенно полезна тогда, когда надо по частям собрать строку из кусочков.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[] = "Ку-ку", s2[10];

    strncpy ( s2, s1, 2 ); // скопировать 2 символа из s1 в s2
    puts ( s2 );          // ошибка! нет последнего '\0'
    s2[2] = '\0';         // добавляем символ окончания строки
    puts (s2);            // вывод
}
```



Проблемы при копировании строк

При копировании стандартные функции **strcpy** и **strncpy** поступают так: определяют количество символов, которые надо скопировать и затем переписывают их, начиная с первого символа до последнего. Если области источника и приемника не перекрываются, то все проходит нормально. Но попробуем «раздвинуть» строку, например для того, чтобы вставить что-то в ее середину. Пусть в строке **s1** записано имя и фамилия человека, а в строке **s2** – его отчество. Надо получить в строке **s1** полностью имя, фамилию и отчество.

```
#include <stdio.h>
#include <string.h>
main()
{
    int n;
    char s1[80] = "Иван Рождественский",
          s2[] = "Петрович ";
    n = strlen(s2); // длина второй строки

    strcpy (s1+5, s1+5+n); // пытаемся раздвинуть на n символов
    strncpy(s1+5, s2, n);  // вставляем отчество в середину

    puts ( s1 );
}
```

При первом вызове **strcpy** мы хотели скопировать конец строки, начиная с символа с номером 5 (он шестой с начала, так как нумерация идет с нуля) вправо на **n** символов (объявленная

длина массива символов – 80 – позволяет это сделать). Однако из-за того, что копирование выполнялось с начала блока данных, скопировав на новое место первый символ фамилии ('Р') функция стерла букву 'н' (справа на 9 символов) и т.д. В результате получили

```
s1 = "Иван РождествеРождествеРождес"
```

В следующей строчке мы скопировали в середину отчество (без завершающего нуля) и получили

```
s1 = "Иван Петрович РождествеРождес"
```

Таким образом, вся задумка не удалась из-за того, что функция копирования работает в данном случае неверно. Выход из этой ситуации такой – написать свою функцию копирования, которая копирует не с начала блока, а с конца (однако она будет неверно работать в обратной ситуации – при сжатии строки). Например, так:

```
void strcpy1 ( char s1[], char s2[] )
{
    int n = strlen(s2);
    while ( n >= 0 )
    {
        s1[n] = s2[n];
        n --;
    }
}
```

Заметьте, что завершающий нуль строки **s2** также копируется. Если использовать в нашем примере эту функцию вместо **strcpy**, то получим желаемый результат.

Возникает вопрос: можно ли сделать функцию, которая всегда правильно копирует? Конечно, можно, хотя это и не просто. Для этого в самой функции надо использовать вспомогательную строку (ее называют *буфером*) и в ней формировать результат так, чтобы в процессе копирования не портилась исходная строка. Когда результат в буфере готов, его останется скопировать в то место, где была исходная строка и удалить память, выделенную под буфер. Попробуйте написать такую функцию, если известно, что ее будут использовать для копирования строк длиной не более 80 символов.



Объединение строк

Еще одна функция – **strcat** (от *string concatenation* – сцепка строк) позволяет добавить строку источник в конец строки-приемника (завершающий нуль записывается автоматически). Надо только помнить, что приемник должен иметь достаточный размер, чтобы вместить обе исходных строки. Функция **strcat** автоматически добавляет в конец строки-результата завершающий символ '\0'.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[80] = "Могу, ",
        s2[] = "хочу, ", s3[] = "надо!";
    strcat ( s1, s2 ); // дописать s2 в конец s1
    puts ( s1 );      // "Могу, хочу, "
    strcat ( s1, s3 ); // дописать s3 в конец s1
    puts ( s1 );      // "Могу, хочу, надо!"
}
```

Заметьте, что если бы строка **s1** была объявлена как **s1[]** (или с длиной меньше 18), произошел бы выход за границы массива с печальными последствиями.

Задача. Ввести с клавиатуры имя файла. Изменить его расширение на ".exe".

Алгоритм решения:

1. Найти в имени файла точку ' .' или признак конца строки '\0'.
2. Если нашли точку, скопировать начиная с этого места новое расширение ".exe" (используем функцию **strcpy**).
3. Если нашли конец строки (точки нет), добавить в конец расширение ".exe" (используем функцию **strcat**).

```
#include <stdio.h>
#include <string.h>
main()
{
    char s[80];
    int  n;           // номер символа '.'
    printf("Введите имя файла ");
    gets ( s );

    n = 0;
    while ( ( s[n] != '.' )           // ищем первую точку
            && ( s[n] != '\0' ) )     // или конец строки
        n ++;

    if ( s[n] == '.' )               // если нашли точку, то...
        strcpy ( s+n, ".exe" );     // меняем расширение,
    else strcat ( s, ".exe" );       // иначе добавляем

    puts ( a );
}
```

Задача. Ввести с клавиатуры фамилию и имя в одну строку (допустим, "Иванов Вася"). Построить в памяти строку «Привет, Вася Иванов!».

Алгоритм решения этой задачи можно записать так:

1. Ввести строку с клавиатуры (назовем ее **s**).

[illegible]

2. Найти длину первого слова (фамилии), обозначим ее через n . Тогда символ с номером n – это пробел между именем и фамилией.

[illegible]

3. Запишем в новую строку первую часть фразы – «Привет, », используя функцию `strcpy`.

[illegible]

4. Добавим в конец этой строки второе слово (имя), используя функцию **strcat**. Для этого надо скопировать в конец строки **a** все символы строки **s**, начиная с **n+1**-ого:

[illegible]

5. Добавим в конец строки пробел, используя функцию **strcat**.

П	р	и	в	е	т	,			В	а	с	я	\0							
---	---	---	---	---	---	---	--	--	---	---	---	---	----	--	--	--	--	--	--	--

6. Определим длину полученной строки и обозначим ее через **len**. Символ с номером **len** – это символ с кодом `'\0'`.

len

a П р и в е т , В а с я \0

7. Добавим в конец строки первое слово (фамилию), используя функцию **strncpy**. Для этого скопируем в конец строки **a** первые **n** символов строки **s**. Обратите внимание, что в конце строки теперь нет завершающего символа `'\0'`.

len+n

a П р и в е т , В а с я И в а н о в ? ?

8. Осталось добавить в конец строки символ `'!'` и `'\0'`. Для этого используется функция **strcpy**. Для копирования нужно использовать адрес **s+len+n**, так как к строке длиной **len** мы добавили фамилию длиной **n**.

П р и в е т , В а с я И в а н о в ! \0

В программе используется важное свойство массивов в языке Си (в том числе и символьных строк, которые являются массивами символов):

Если массив называется **s**, то запись **s+i** обозначает адрес элемента **s[i]** (так же, как и **&s[i]**).

Так как функциям **strcpy** и **strcat** надо передать адреса в памяти, куда и откуда переместить данные, мы можем использовать запись **a+len** вместо **&a[len]** и т.д.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s[80], a[80] = "Привет, ";
    int n, len;
    printf("Введите фамилию и имя ");
    gets ( s );

    n = 0;
    while ( (s[n] != ' ') // ищем первый пробел
           && (s[n] != '\0') ) // или конец строки
        n ++;
    if ( s[n] != ' ' ) { // если нет пробела, ...
        printf( "Неверная строка" );
        return 1; // выход по ошибке, код ошибки 1
    }
    strcat ( a, s+n+1 ); // добавить имя
    strcat ( a, " " ); // добавить пробел
    len = strlen ( a ); // найти длину строки
    strncpy ( a + len, s, n ); // добавить фамилию
    strcpy ( a + len + n, "!" ); // добавить "!"

    puts ( a );
}
```



Поиск в строках

Когда говорят о поиске в строках, обычно рассматривают две задачи: найти первый заданный символ с начала (или с конца), или также найти заданную подстроку (если она есть).

Первую задачу выполняют функции **strchr** (поиск с начала строки) и **strrchr** (поиск с конца строки), а вторую – функция **strstr**.

Все эти функции возвращают указатель на найденный символ (или на первый символ найденной подстроки). Это значит, что переменная, в которую записывается это значение, должна быть объявлена как *указатель* на символьную переменную. Мы уже встречались с указателями, когда работали с файлами. Указатель – это ячейка в памяти, в которую можно записывать *адрес* другой переменной.

Структура вызова функций такая: на первом месте – где искать (строка), на втором – что искать (один символ для функций **strchr** и **strrchr** или строка для **strstr**). Чтобы получить номер символа с начала строки, надо вычесть из полученного указателя адрес начала массива. Если поиск завершился неудачно, функции возвращают **NULL**.

```
#include <stdio.h>
#include <string.h>
main()
{
    char s1[] = "Мама мыла раму",
          s2[] = "Война и мир", *p;

    p = strchr(s1, 'a');
    if ( p != NULL ) {
        printf("Первая буква а: номер %d", p - s1);
        p = strrchr(s1, 'a');
        printf("\nПоследняя буква а: номер %d", p - s1);
    }

    p = strstr( s2, "мир");
    if ( p != NULL )
        printf("\nНашли мир в %s", s2);
    else printf("\nНет слова мир в %s", s2);
}
```

Вспомните, что при чтении строк из файла с помощью функции **fgets** на конце иногда остается символ перехода на новую строку **'\n'**. Чаще всего он совсем не нужен и надо его удалить – поставить на его месте ноль (признак конца строки). Делается это так:

```
char s[80], *p;
...
p = strrchr (s, '\n'); // ищем символ '\n'
if ( p != NULL )      // если нашли, ...
    *p = '\0';        // записываем в это место '\0'
```

Рассмотрим теперь более сложную задачу.

Задача. С клавиатуры вводится предложение и слово. Надо определить, сколько раз встречается это слово в предложении.

Функция **strstr** может определить только первое вхождение слова в строку, поэтому в одну строчку эту задачу не решить.

Попробуем использовать такую идею: если мы нашли адрес первого данного слова в строке и записали его в указатель **p**, то искать следующее слово нужно не с начала строки, а с адреса **p+длина_слова**. Повторяем эту операцию в цикле, пока функция **strstr** может найти слово в оставшейся части строки. Поскольку начало области поиска постоянно смещается с каждым новым найденным словом, адрес оставшейся части надо хранить в отдельной переменной типа «указатель на символ». Реализация может выглядеть так:

```

#include <stdio.h>
#include <string.h>
main()
{   int len, count;
    char s[80], word[20],
        *p,                // указатель на найденное слово
        *start;            // указатель на начало зоны поиска
    puts ( "Введите предложение" );
    gets ( s );
    puts ( "Введите слово для поиска" );
    gets ( word );
    len = strlen ( word ); // находим длину слова

    count = 0; // счетчик найденных слов
    start = s; // в первый раз ищем с начала строки
    while ( 1 ) {
        p = strstr (start, word); // есть ли еще слова?
        if ( p == NULL ) break;    // если нет, то выход
        count ++;                 // увеличить счетчик
        start = p + len;          // сместили начало поиска
    }

    printf ( "В этом предложении %d слов %s", count, word );
}

```

В конце работы цикла в переменной **count**, будет записано количество данных слов в предложении. Заметьте, что вместо переменной **start** можно везде использовать **p**, результат от этого не изменится.



Форматирование строк

В программах часто требуется перед выводом информации сформировать всю строку для вывода целиком, включив в нее все необходимые данные. Например, сообщение об ошибке выводится стандартной функцией, и в это сообщение надо включить числовые данные. Другой пример – вывод текста в графическом режиме, для которого нет аналога функции **printf**.

В этих случаях необходимо использовать функцию **sprintf**, которая поддерживает те же форматы данных, что и **printf**, но записывает результат не на экран и не в файл, а в *символьную строку* (под нее надо заранее выделить память). Вот как выглядит вывод на экран значения переменных **x** и **y** в графическом режиме:

```

#include <stdio.h>
#include <conio.h>
#include <graphics.h>
main()
{
    char s[80]; // вспомогательная строка
    int x, y;
    // здесь нужно открыть окно для графики
    x = 1;
    y = 5;

    sprintf (s, "X=%d, Y=%d", x, y); // вывод в строку s
    outtextxy ( 100, 100, s );      // вывод строки s на экран
    getch();
    closegraph();
}

```

Не забудьте, что для использования функции **outtextxy** надо открыть окно для работы с графикой (с помощью функции **initwindow**).



Чтение из строки

Иногда, особенно при чтении данных из файлов, возникает обратная задача: есть символьная строка, в которой записаны данные. Необходимо ввести их в соответствующие ячейки памяти.

В этом случае используется функция **sscanf**, которая читает данные по указанному формату не с клавиатуры (как **scanf**) и не из файла (как **fscanf**), а из символьной строки. В приведенном ниже примере мы ищем в файле строчку, которая начинается с символа **#** и считываем из нее значения **x** и **y**.

Сложность задачи заключается в том, что мы не знаем точно, какая по счету эта строчка в файле. Если не использовать функцию **sscanf**, то пришлось бы сначала найти номер нужной строки в файле, затем начать просмотр с начала, отсчитать нужное количество строк и использовать **fscanf**.

```
#include <stdio.h>
main()
{
    char s[80];    // вспомогательная строка
    int x, y;
    FILE *fp;
    fp = fopen ( "input.dat", "r" );
    while ( fgets ( s, 80, fp ) )
        if ( s[0] == '#' ) { // если строка начинается с #, ...
            sscanf ( s+1, "%d%d", &x, &y); // читаем данные
            break; // выход из цикла
        }
    fclose ( fp );
    printf ( "x = %d, y = %d", x, y );
}
```



Строки в функциях и процедурах

Стандартные и ваши собственные функции могут принимать строки в качестве параметров. Здесь, однако, есть некоторые тонкости.

Вы знаете, что если параметр функции объявлен как **int a**, то изменение переменной **a** в функции никак не влияет на ее значение в вызывающей программе – функция создает копию переменной и работает с ней (для того, чтобы функция могла это сделать, надо объявить параметр как **int &a**). Это относится ко всем простым типам.

Когда вы передаете в функцию или процедуру строку, вы в самом деле передаете *адрес начала строки*, никаких копий строки не создается. Поэтому всегда надо помнить, что

При изменении строки-параметра в функции или процедуре меняется и соответствующая строка в основной программе.

Как же объявить строку-параметр в своей функции? В простейшем варианте – так же, как и массив символов: **char s[]**. А можно также и как указатель (все равно в функцию передается адрес строки **char *s**). Между этими двумя способами есть некоторая разница. В первом случае **s** – это имя массива символов, поэтому нельзя его изменять, а во втором случае – указатель на символ, который можно сдвигать, как хочется.

Все стандартные функции языка Си объявляют символьные строки как указатели – это дает большую свободу действий. Сравните, например, две реализации функции, которая копирует строки (аналогично **strcpy**). Первая выполнена с помощью параметра-массива, а вторая – с помощью указателя.

```
void copy1 ( char s1[],
             char s2[] )
{
    int i = 0;
    while ( s2[i] ) {
        s1[i] = s2[i];
        i ++;
    }
    s1[i] = '\0';
}
```

```
void copy2 ( char *s1, char *s2 )
{
    while ( *s1++ = *s2++ );
}
```

- 1) скопировать ***s2** по адресу **s1**
- 2) увеличить **s1** и **s2**
- 3) делать так пока ***s2** не ноль

Как видите, вторая функция получилась более компактной. Применение указателей позволило не вводить дополнительную переменную, хотя и сделала программу менее ясной. Итак, в условии цикла **while** стоит оператор присваивания. Если не обращать внимания на плюсы, он означает «взять символ по адресу **s2** и записать его по адресу **s1**». Двойные плюсы ПОСЛЕ **s1** и **s2** означают, что ПОСЛЕ выполнения присваивания оба **указателя** надо увеличить на единицу, то есть перейти к следующему символу.

Что же является условием цикла? Оказывается условие – это величина ***s1**, то есть код символа по адресу **s1**. Когда же происходит проверка? Это зависит от расположения знаков **++**. В данном случае они стоят ПОСЛЕ имен переменных, поэтому операция инкремента выполняется ПОСЛЕ проверки условия. Проверка выполняется так: скопировали очередной символ, посмотрели на него, и если он – ноль (признак конца строки), то вышли из цикла. После этого увеличили указатели **s1** и **s2**. Обратите внимание, что после выхода из цикла увеличение указателей также происходит, и они будут указывать не на нули, завершающие строки, а на следующие байты в памяти.

3. Матрицы (двухмерные массивы)

Что такое матрица?

Вспомните, что из себя представляет адрес любого человека (или фирмы). Вы можете сказать: "Я живу на Невском проспекте в доме 20 в квартире 45", но никому в голову не придет сказать: "Я живу в 13678 квартире от начала Невского проспекта". Почему? Потому что неудобно. Первая часть адреса – улица, вторая – дом, третья – квартира.

Часто обычный массив неудобен для хранения данных из-за своей линейной структуры. Например, пусть нам надо обрабатывать информацию о количестве выпавших осадков за несколько лет, причем известны осадки по месяцам. Если обрабатывать данные вручную, то удобнее всего использовать таблицу – по горизонтали откладывать года, а по вертикали – месяцы (или наоборот). В принципе в программе можно использовать и одномерный массив, но тогда требуется пересчитывать индексы – по известному году и месяцу получать смещение ячейки от начала линейного массива:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
янв	фев	мар	апр	май	июн	июл	авг	сен	окт	ноя	дек	янв	фев	мар	апр	май	июн	июл
2008												2009						

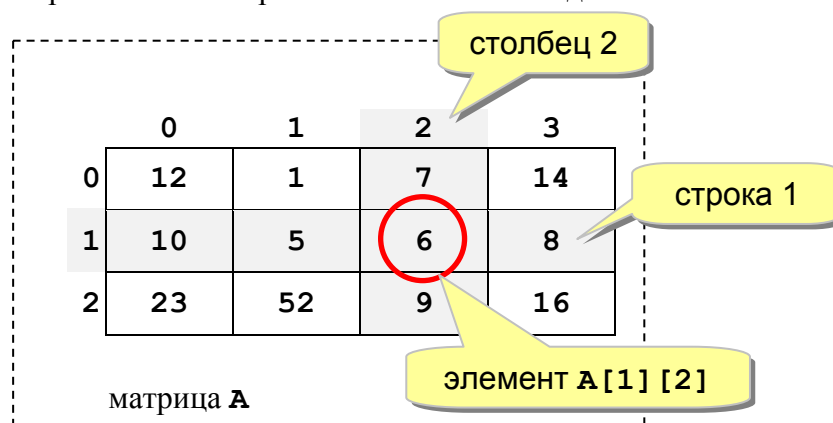
Для такого пересчета можно было бы использовать формулу

$$i = (\text{год} - 2008) * 12 + \text{месяц} - 1;$$

где i – индекс нужного элемента массива. Тем не менее, так не делают (почти никогда), потому что для работы с табличными данными во всех современных языках программирования существуют *двухмерные массивы* или *матрицы*.

Матрица – это прямоугольная таблица элементов (например, чисел или символов). В информатике матрица представляется в виде двухмерного массива, то есть массива, все элементы которого имеют два индекса.

Матрица, как и таблица, состоит из строк и столбцов. Два индекса элемента – это и есть номера строки и столбца, на пересечении которых этот элемент находится.



В языке Си каждый индекс записывается отдельно в квадратных скобках. Каждую строку и каждый столбец матрицы можно рассматривать как обычный одномерный массив. Поэтому можно сказать, что матрица – это *массив из массивов*. Существует только два ограничения:

- 1) все элементы матрицы должны быть **одинакового типа**;
- 2) все строки матрицы должны быть одинаковой длины.

Первый индекс элемента матрицы – это *строка*, второй – *столбец*. Поэтому когда говорят о «матрице 4 на 5», это означает, что матрица имеет 4 строки и 5 столбца.

Матрицы, у которых число строк равно числу столбцов, называют *квадратными*. В квадратных матрицах можно выделить *главную диагональ* – это все элементы, у которых номер строки равен номеру столбца, то есть

$A[0][0], A[1][1], \dots, A[N-1][N-1]$

для матрицы размером **N** на **N**.

Объявление матриц

Матрицы объявляются также, как и простые массивы, но у них не один индекс, а два. При объявлении в отдельных квадратных скобках указывается количество строк и количество столбцов. Например, оператор

```
int Bmv[20][10];
```

выделит место в памяти под матрицу целых чисел, имеющую 20 строк и 10 столбцов (всего 200 элементов). Если матрица глобальная (объявляется выше всех процедур и функций), то она в самом начале заполняется нулями. Локальные матрицы (объявленные внутри процедуры или функции) содержат «мусор» – неизвестные значения.

Начальные значения элементов

При объявлении можно сразу задать все или часть ее элементов, например так

```
float X[2][3] = {{1., 2., 3.},{4., 5., 6.}};
```

Как видно из примера, элементы каждой строки заключаются в отдельные фигурные скобки. Если задать не все элементы, то остальные заполнятся нулями:

```
float X[2][3] = {{1., 3.},{6.}};
```

Здесь элементы $X[1][2]$, $X[2][1]$ и $X[2][2]$ будут нулевыми.

Расположение матриц в памяти

Иногда бывает полезно знать, как матрицы располагаются в памяти ЭВМ. Оказывается во всех современных языках программирования (кроме *Фортрана*) элементы матрицы располагаются *по строкам*, то есть сначала изменяется последний индекс. Объявленная выше матрица **X** расположена так:

$X[0][0]$	$X[0][1]$	$X[0][2]$	$X[1][0]$	$X[1][1]$	$X[1][2]$
-----------	-----------	-----------	-----------	-----------	-----------

Стандартный ввод и вывод

Как и для одномерных массивов, матрицы могут быть введены с клавиатуры, из файла, и заполнены с помощью случайных чисел. Общий принцип – для каждого элемента функция чтения вызывается отдельно. Представим себе матрицу как массив строк равной длины. Для ввода одной строки требуется цикл, и таких строк несколько. Поэтому для работы с матрицами требуется *двойной* или *вложенный* цикл, то есть цикл в цикле.

Ввод с клавиатуры

Единственная проблема состоит в том, чтобы не перепутать переменные в двух циклах и пределы их изменения.

```
#include <stdio.h>
const int M = 5; // число строк
const int N = 4; // число столбцов
main()
{
    int i, j, A[M][N];
    for ( i = 0; i < M; i ++ )           // цикл по строкам
        for ( j = 0; j < N; j ++ )       // цикл по столбцам строки
        {
            printf ("A[%d][%d]=", i, j); // подсказка для ввода
            scanf ("%d", & A[i][j]);     // ввод A[i][j]
        }
    // работа с матрицей
}
```

Заметьте, что при изменении порядка циклов (если поменять местами два оператора `for`) изменится и порядок ввода элементов в память.

Заполнение случайными числами

Выполняется также в двойном цикле аналогично одномерным массивам. В примере показано заполнение целой матрицы случайными числами в интервале $[a, b]$ (для вещественных чисел формула изменится – см. одномерные массивы). Функция

```
int random ( int N ) { return rand() % N; }
```

возвращающая случайное целое число в интервале $[0, N-1]$, была рассмотрена выше, когда мы говорили о массивах (ее нужно добавить в программу).

В этой и следующей программах мы будем считать, что объявлена целая матрица **M** на **N**, где **M** и **N** — целые константы (объявленные через **const**), а также целые переменные **i** и **j**.

```
for ( i = 0; i < M; i ++ )
    for ( j = 0; j < N; j ++ )
        A[i][j] = random(b-a+1) + a;
```

Вывод на экран

При выводе матрицы ее элементы желательно расположить в привычном виде – по строкам. Напрашивается такой прием: вывели одну строку матрицы, перешли на новую строку экрана, и т.д. Надо учитывать, что для красивого вывода на каждый элемент матрицы надо отвести равное количество символов (иначе столбцы будут неровные). Делается это с помощью форматирования – цифра после знака процента задает количество символов, отводимое на данное число.

```
printf("Матрица A\n");
for ( i = 0; i < M; i ++ ) {           // цикл по строкам
    for ( j = 0; j < N; j ++ )         // вывод одной строки (в цикле)
        printf ( "%4d", A[i][j] );    // 4 символа на число
    printf("\n");                      // переход на другую строку
}
```

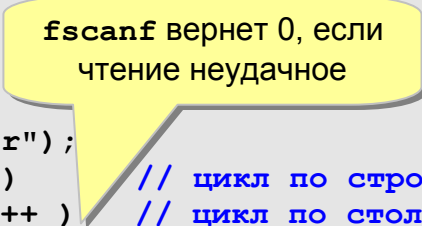

Работа с файлами

Текстовые файлы

При вводе из текстового файла надо читать последовательно все элементы, обрабатывая (так же, как и для линейных массивов) ошибки отсутствия или недостатка данных в файле.

```
#include <stdio.h>
const int M = 5; // число строк
const int N = 4; // число столбцов
main()
{
    int i, j, A[M][N];
    FILE *fp;
    fp = fopen("input.dat", "r");
    for ( i = 0; i < M; i ++ ) // цикл по строкам
        for ( j = 0; j < N; j ++ ) // цикл по столбцам
            if ( 0 == fscanf(fp, "%d", &A[i][j]) ) // ввод A[i][j]
            {
                puts("Не хватает данных");
                fclose ( fp ); // закрыть файл по ошибке
                return 1; // выход по ошибке, код ошибки 1
            }

    fclose ( fp ); // закрыть файл нормально
    // работа с матрицей
}
```



Вывести матрицу в текстовый файл можно так же, как и на экран, только надо сначала открыть текстовый файл на запись, затем в двойном цикле использовать функцию **fprintf** вместо **printf**, и в конце закрыть файл (см. вывод одномерных массивов в файл).

Двоичные файлы

С двоичным файлом удобно работать тогда, когда данные записала (или будет читать) другая программа и их не надо просматривать вручную. Основное преимущество этого способа — скорость чтения и записи, поскольку весь массив читается (или записывается) сразу единым блоком. При этом функциям **fread** и **fwrite** надо указать размер одного элемента массива и количество таких элементов, то есть **M*N**.

В программе, которая приведена ниже, матрица читается из двоичного файла, затем с ней выполняются некоторые действия (они обозначены многоточием) и эта же матрица записывается в выходной файл.

```
#include <stdio.h>
const int M = 5; // число строк
const int N = 4; // число столбцов
main()
{
    int total, A[M][N];
    FILE *fp;

    fp = fopen("input.dat", "rb");
    total = fread(A, sizeof(int), M*N, fp); // чтение матрицы
    fclose ( fp );
}
```



```

    if ( total != M*N ) // обработка ошибки
    {
        printf("Не хватает данных");
        return 1;      // выход по ошибке, код ошибки 1
    }
    // работа с матрицей

    fp = fopen("output.dat", "wb"); // запись матрицы в файл
    if ( M*N != fwrite(A, sizeof(int), M*N, fp) )
        printf("Ошибка записи в файл");
    fclose ( fp );
}

```

Для обработки ошибок используется тот факт, что функции `fread` и `fwrite` возвращают количество реально прочитанных (записанных) элементов, и если оно не равно заданному ($M \cdot N$), то произошла ошибка.



Алгоритмы для работы с матрицами



Перебор элементов матрицы

В отличие от одномерных массивов, для перебора всех элементов матрицы надо использовать двойной цикл. Ниже показано, как найти минимальный элемент в массиве и его индексы. Сначала считаем, что минимальным является элемент `A[0][0]` (хотя можно начать и с любого другого), а затем проходим все элементы, проверяя, нет ли где еще меньшего. Так же, как и для одномерного массива, запоминаются только индексы, а значение минимального элемента «вытаскивается» прямо из массива.

```

float A[M][N], i, j, row, col;
...

row = col = 0; // сначала считаем, что A[0][0] - минимальный
for ( i = 0; i < M; i ++ ) // просмотр всех строк
    for ( j = 0; j < N; j ++ ) // просмотр всех столбцов
        if ( A[i][j] < A[row][col] ) {
            row = i; // запомнили новые индексы
            col = j;
        }

printf ("Минимальный элемент A[%d][%d]=%d",
        row, col, A[row][col]);

```

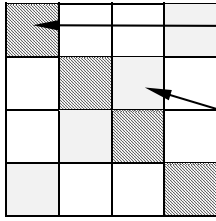


Работа с отдельными элементами

Рассмотрим квадратную матрицу N на N . Выведем на экран обе ее диагонали (главную диагональ и перпендикулярную ей). С главной диагональю все просто – в цикле выводим все элементы, у которых номера строки и столбца равны, то есть `A[i][i]` для всех i от 0 до $N-1$. Вторую диагональ формируют такие элементы:

`A[0][N-1], A[1][N-2], A[2][N-3], ..., A[N-1][0]`

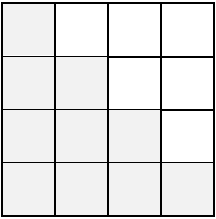
Обратим внимание, что каждый следующий элемент имеет номер строки на 1 больше, а номер столбца – на 1 меньше. Таким образом, **сумма номеров строки и столбца постоянна** и равна $N-1$. Тогда, зная номер строки i можно сразу сказать, что на второй диагонали стоит ее элемент `A[i][N-1-i]`.



```
printf("Главная диагональ:\n");
for ( i = 0; i < N; i ++ )
    printf ("%d ", A[i][i]);

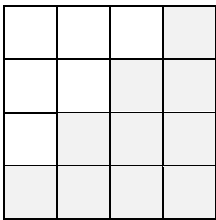
printf("\nВторая диагональ:\n");
for ( i = 0; i < N; i ++ )
    printf ("%d ", A[i][N-1-i]);
```

Теперь можно рассмотреть более сложный случай. Обнулим все элементы, кроме тех, которые стоят выше главной диагонали. В первой строке это единственный элемент $A[0][0]$, во второй – два элемента: $A[1][0]$ и $A[1][1]$, в третьей – три: $A[2][0]$, $A[2][1]$ и $A[2][2]$. Заметим, что в строке i обнуляются все элементы, у которых номера столбцов от 0 до i .



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

Еще более сложно обработать все элементы, стоящие не выше второй диагонали. В первой строке это единственный элемент $A[0][N-1]$, во второй – два элемента: $A[1][N-2]$ и $A[1][N-1]$, в третьей – три: $A[2][N-3]$, $A[2][N-2]$ и $A[2][N-1]$. Заметим, что в строке i обнуляются все элементы, у которых номера столбцов от $N-1-i$ до $N-1$.



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



Перестановка строк и столбцов

Пусть надо переставить две строки с индексами $i1$ и $i2$. Это значит, что для каждого столбца j надо поменять местами элементы $A[i1][j]$ и $A[i2][j]$ через временную переменную (она называется **temp**).

```
for ( j = 0; j < N; j ++ ) {
    temp = A[i1][j];
    A[i1][j] = A[i2][j];
    A[i2][j] = temp;
}
```



Преобразование в одномерный массив

Иногда надо скопировать матрицу A размером M на N в одномерный массив B размером $M*N$. Очевидно, что при копировании по строкам (сначала первая строка, затем вторая и т.д.) элемент первой строки $A[0][j]$ надо скопировать в $B[j]$, элементы второй строки $A[1][j]$ – в $B[N+j]$ и т.д. Отсюда следует, что для любой строки i элемент $A[i][j]$ копируется в $B[i*N+j]$. теперь осталось только в двойном цикле перебрать все элементы матрицы.

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

Заметим, что если надо провести какую-то операцию со всеми или некоторыми (стоящими подряд) элементами матрицы и для одномерного массива уже есть соответствующая процедура, ее можно использовать, учитывая, что имя строки массива (например, **A[0]**) является указателем на начальный элемент этой строки. При этом надо учитывать, что в памяти элементы матрицы расположены по строкам. Например, функция вычисления суммы элементов (см. массивы) может применяться для матрицы так:

```
s0 = Sum(A[0], N);      // сумма строки 0
s14 = Sum(A[1], 4*N);   // сумма строк 1-4
sAll = Sum(A[0], M*N);  // сумма всех строк
```

4. Массивы символьных строк

Если строка – это массив символов, то массив строк – это массив из массивов или *двухмерный массив символов* (матрица). Как и в матрицах, длина всех строк (выделенная память) должна быть одинакова. Отличие массива строк от матрицы целых или вещественных заключается в том, что строка рассматривается не просто как набор символов, а как единое целое – символьная строка.



Объявление и инициализация

Ниже показано два способа объявления массива из 4 строк, в каждой из которых может быть до 80 символов (считая завершающие нули). Первый способ – это стандартное объявление двухмерного массива символов.

```
char s[4][80];
```

Во втором сначала определяется новый тип данных **str80** – строка длиной 80 символов, а потом объявляется массив из этих строк.

```
typedef str80[80]; // новый тип данных – строка из 80 символов
...
str80 s[4];        // объявление массива строк
```

Напомним, что директива **typedef** должна стоять до первого использования нового типа. Принято ставить ее вне всех процедур и функций, там же, где объявляются глобальные переменные. Дальше каждую строку можно использовать отдельно. В том числе можно «добраться» до любого символа, вспомнив, что мы имеем дело с двухмерным массивом и **s[2][15]** – это 15-ый символ 2-ой строки (не забудьте, что нумерация начинается с нуля!).

Всем или нескольким первым строкам массива можно задать начальные значения после знака равенства в фигурных скобках через запятую. Если начальных значений меньше, чем строк в массиве, оставшиеся последние строки заполняются нулями.

```
char s[4][80] = { "Вася", "Петя" };
```



Ввод и вывод

Для каждой строки ввод и вывод делается отдельно, то есть для ввода и вывода массива строк надо использовать цикл. Следующий фрагмент позволяет ввести не более 20 строк (пока не введена пустая строка). Переменная **count** будет хранить количество введенных строк.

```
char s[20][80];
int i, count = 0;
printf("Введите текст (Enter - закончить)\n");
for ( i = 0; i < 20; i++ ) { // пытаемся прочитать 20 строк
    gets (s[count]);        // прочитать строку с клавиатуры
    if (s[count][0] == '\0') // если введена пустая строка,
        break;              // то выйти из цикла
    count++;
}
```

Вывод массива строк на экран, а также работа с текстовым файлом (ввод и вывод) выполняется аналогично в цикле.



Сортировка

Мы уже научились сортировать массивы целых и вещественных чисел, а теперь надо сортировать строки. Можно придумать разные способы, но наиболее часто встречается алфавитная сортировка. При сортировке строк возникают две проблемы.

1. Определить, какая из двух строк «меньше», то есть какая из двух должна стоять выше.
2. Сортировка массивов чисел предусматривает перестановку элементов массива. В случае строк это связано с копированием больших объемов данных, что крайне невыгодно.

Начнем с первой. Мы говорили, что есть функция сравнения строк **strcmp**, которая возвращает нуль, если строки равны, и не нуль, если они разные. Оказывается, эта функция возвращает «разность» этих строк, то есть *разность кодов* их первых отличающихся символов. Если функция вернула отрицательное число, то первая строка «меньше» второй и стоит по алфавиту раньше, если положительное – наоборот. Здесь надо учитывать, что сравнение идет по таблице кодов, и коды заглавных букв меньше, чем коды строчных (и для русских, и для английских).

Вторая проблема более серьезная и решается с помощью *указателей*. Сначала выделим в памяти массив указателей на строки и сделаем так, чтобы *i*-ый указатель указывал на *i*-ую строку массива. Теперь достаточно правильно расставить указатели и сортировка будет выполнена – к строкам можно будет обращаться в алфавитном (или каком-либо другом) порядке с помощью этих указателей. Процедура сортировки методом пузырька выглядит так:

```
void SortStrings ( char *s[], int n ) // *s[] – массив указателей
{                                     // n – число строк
    char *p;
    int i, j;
    for ( i = 0; i < n-1; i ++ )
        for ( j = n-1; j > i; j -- )
            if ( strcmp(s[j-1],s[j]) > 0 ) {
                p = s[j]; s[j] = s[j-1]; // перестановка УКАЗАТЕЛЕЙ
                s[j-1] = p;
            }
}
```

Эту функцию использует приведенная ниже программа, которая сортирует строки и выводит их на экран уже в алфавитном порядке.

```
main()
{
    char s[20][80]; // массив из 20 строк
    char *ps[20];  // массив из 20 указателей на строки
    int i, count;
    // здесь нужно ввести строки и записать
    // в переменную count их количество

    for ( i = 0; i < count; i ++ ) // расставить указатели
        ps[i] = s[i];

    SortStrings ( ps, count );      // сортировка указателей

    for ( i = 0; i < count; i ++ )
        puts ( ps[i] );            // вывод через указатели
}
```

5. Управление памятью



Указатели

Рассмотрим такую задачу: в файле записаны целые числа. Надо отсортировать их и записать в другой файл. Проблема заключается в том, что заранее неизвестно, сколько в файле таких чисел. В такой ситуации есть два варианта:

1. Выделить память с запасом, если известно, например, что этих чисел гарантированно не более 1000.
2. Использовать динамическое выделение памяти – сначала определить, сколько чисел в массиве, а потом выделить ровно столько памяти, сколько нужно.

Наиболее грамотным решением является второй вариант (использование динамических массивов). Для этого надо сначала поближе познакомиться с указателями.

Указатель — это переменная, в которой хранится адрес другой переменной или участка памяти.

Указатели являются одним из основных понятий языка Си. В такие переменные можно записывать адреса любых участков памяти, на чаще всего – адрес начального элемента динамического массива. Что же можно делать с указателями?

Объявить	<code>char *pC; int *pI; float *pF;</code>	Указатели объявляются в списке переменных, но перед их именем ставится знак *. Указатель всегда указывает на переменную того типа, для которого он был объявлен. Например, pC может указывать на любой символ, pI – на любое целое число, а pF – на любое вещественное число.
Присвоить адрес	<code>int i, *pI; ... pI = &i;</code>	Такая запись означает: «записать в указатель pI адрес переменной i ». Запись &i обозначает адрес переменной i .
Получить значение по этому адресу	<code>float f, *pF; ... f = *pF;</code>	Такая запись означает: «записать в переменную f то вещественное число, на которое указывает указатель pF ». Запись *pF обозначает содержимое ячейки, на которую указывает pF .
Сдвинуть	<code>int *pI; ... pI++; pI += 4; -- pI; pI -= 4;</code>	В результате этих операций значение указателя меняется особым способом: pI++ сдвигает указатель на РАЗМЕР ЦЕЛОГО ЧИСЛА, то есть на 4 байта, а не на 1 как можно подумать. А запись pI+=4 или pI=pI+4 сдвигает указатель на 4 целых числа дальше, то есть на 16 байт.
Обнулить	<code>char *pC; ... pC = NULL;</code>	Если указатель равен нулевому адресу NULL , это обычно означает, что указатель недействительный. По нему нельзя ничего записывать (это вызывает сбой программы компьютера).
Вывести на экран	<code>int i, *pI; ... pI = &i; printf("Адр.i =%p", pI);</code>	Для вывода указателей используется формат %p .

Итак, что надо знать про указатели:

- указатель – это переменная, в которой записан адрес другой переменной;
- при объявлении указателя надо указать тип переменных, на которых он будет указывать, а перед именем поставить знак *;
- знак & перед именем переменной обозначает ее адрес;
- знак * перед указателем в рабочей части программы (не в объявлении) обозначает значение ячейки, на которую указывает указатель;
- нельзя записывать по указателю, который указывает непонятно куда – это вызывает сбой программы, поскольку что-то стирается в памяти;
- для обозначения недействительного указателя используется константа **NULL**;
- при изменении значения указателя на **n** он в самом деле сдвигается к **n**-ому следующему числу данного типа, то есть для указателей на целые числа на **n*sizeof(int)** байт;
- указатель печатаются по формату **%p**.

Теперь вам должно быть понятно, что многие функции ввода типа **scanf** и **fscanf** в самом деле принимают в параметрах адреса переменных, например

```
scanf ( "%d", &i);
```



Динамическое выделение памяти

Динамическими называются массивы, размер которых неизвестен на этапе написания программы. Прием, о котором мы будем говорить, относится уже не к стандартному языку Си, а к его расширению Си ++. Существуют и стандартные способы выделения памяти в языке Си (с помощью функций **malloc** и **calloc**), но они не очень удобны.

Следующая простейшая программа, которая использует динамический массив, вводит с клавиатуры размер массива, все его элементы, а затем сортирует их и выводит на экран.

```
#include <stdio.h>
main()
{
    int N;          // размер массива (заранее неизвестен)
    int *A;         // указатель для выделения памяти
    printf ("Размер массива > "); // ввод размера массива
    scanf ("%d", &N);
    A = new int [N]; // выделение памяти
    if ( A == NULL ) { // если не удалось выделить
        printf("Не удалось выделить память");
        return 1;      // выход по ошибке, код ошибки 1
    }
    for (i = 0; i < N; i ++ ) { // дальше так же, как для массива
        printf ("\nA[%d] > ", i+1);
        scanf ("%d", &A[i]);
    }
    // здесь сортировка и вывод на экран
    delete A; // освободить память
}
```

Итак, мы хотим выделить в памяти место под новый массив целых чисел во время работы программы. Мы уже знаем его размер, пусть он хранится в переменной **N** (это число обязательно должно быть больше нуля). Оператор выделения памяти **new** вернет нам адрес нового выделенного блока и для работы с массивом нам надо где-то его запомнить. Вы уже знаете, что есть специальный класс переменных для записи в них адресов памяти – указатели.



Что новенького?

- динамические массивы используются тогда, когда на момент написания программы размер массива неизвестен
- для того, чтобы работать с динамическим массивом, надо объявить указатель соответствующего типа (в нем будет храниться адрес первого элемента массива);

```
int *A;
```

- когда требуемый размер массива стал известен, надо использовать оператор **new** языка Си++, указав в скобках размер массива (в программе для краткости нет проверки на положительность **N**) ;

```
A = new int[N];
```

- нельзя** использовать оператор **new** при отрицательном или нулевом **N**;
- после выделения памяти надо проверить, успешно ли оно прошло; если память выделить не удалось, то значение указателя будет равно **NULL**, использование такого массива приведет к сбой программы;
- работа с динамическим массивом, на который указывает указатель **A**, идет также, как и с обычным массивом размера **N** (помните, что язык Си не следит за выходом за границы массива);
- после использования массива надо освободить выделенную память, вызвав оператор

```
delete A;
```

- после освобождения памяти значение указателя не изменяется, но использовать его уже нельзя, потому что память освобождена;
- учитывая, что при добавлении числа к указателю он сдвигается на заданное число ячеек **ЗАДАННОГО ТИПА**, то следующие записи равносильны и вычисляют адрес **i**-ого элемента массива:

```
&A[i]            и            A+i
```

Отсюда следует, что **A** совпадает с **&A[0]**, и поэтому имя массива можно использовать как адрес его начального элемента;



Ошибки, связанные с выделением памяти

Самые тяжелые и трудно вылавливаемые ошибки в программах на языке Си связаны именно с неверным использованием динамических массивов. В таблице перечислены наиболее тяжелые случаи и способы борьбы с ними.

Ошибка	Причина и способ лечения
Запись в чужую область памяти	Память была выделена неудачно, а массив используется. Вывод: надо всегда делать проверку указателя на NULL .
Повторное удаление	Массив уже удален и теперь удаляется снова.

указателя	Вывод: если массив удален из памяти, обнулите указатель – ошибка быстрее выявится.
Выход за границы массива	Запись в массив в элемент с отрицательным индексом или индексом, выходящим за границу массива
Утечка памяти	Неиспользуемая память не освобождается. Если это происходит в функции, которая вызывается много раз, то ресурсы памяти скоро будут израсходованы. Вывод: убирайте за собой «мусор» (освобождайте память).



Выделение памяти для матрицы

Для выделения памяти под одномерный массив целых чисел нам потребовался указатель на целые числа. Для матрицы надо выделить указатель на массив целых чисел, который объявляется как

```
int **A;
```

Но лучше всего сразу объявить новый тип данных - указатель на целое число. Новые типы объявляются директивой **typedef** вне всех процедур и функций (там же, где и глобальные переменные).

```
typedef int *pInt;
```

Этой строкой мы сказали компилятору, что любая переменная нового типа **pInt** представляет собой указатель на целое число или адрес массива целых чисел. К сожалению, место для матрицы не удастся так же просто выделить в памяти, как мы делали это для одномерного массива. Если написать просто

```
int M = 5, N = 7;
pInt *A;
A = new int[M][N]; // ошибочная строка
```

компилятор выдает множество ошибок. Связано это с тем, что ему требуется заранее знать длину одной строки, чтобы правильно расшифровать запись типа **A[i][j]**. Ниже рассмотрены три способа решения этой проблемы.



Известный размер строки

Если размер строки матрицы известен, а неизвестно только количество строк, можно поступить так: ввести новый тип данных – строка матрицы. Когда количество строк станет известно, с помощью оператора **new** выделяем массив таких данных.

typedef int row10[10]; // новый тип: массив из 10 элементов
main()
{
int N;
row10 *A; // указатель на массив (матрица)
printf ("Введите число строк ");
scanf ("%d", &N);
A = new row10[N]; // выделить память на N строк

<code>A[0][1] = 25;</code>	<code>// используем матрицу, как обычно</code>
<code>printf("%d", A[2][3]);</code>	
<code>delete A;</code>	<code>// освобождаем память</code>
<code>}</code>	

Неизвестный размер строки

Пусть размеры матрицы **M** и **N** заранее неизвестны и определяются в ходе работы программы. Тогда можно предложить следующий способ выделения памяти под новую матрицу. Поскольку матрицу можно рассматривать как массив из строк-массивов, объявим **M** указателей и выделим на каждый из них область памяти для одномерного массива размером **N** (то есть, на одну строку). Сами эти указатели тоже надо представить в виде динамического массива. Определив требуемые размеры матрицы, мы выделяем сначала динамический массив указателей, а потом на каждый указатель – место для одной строки.

<code>typedef int *pInt; // новый тип данных: указатель на целое</code>
<code>main()</code>
<code>{</code>
<code>int M, N, i;</code>
<code>pInt *A; // указатель на указатель</code>
<code>// ввод M и N</code>
<code>A = new pInt[M]; // выделить память под массив указателей</code>
<code>for (i = 0; i < M; i ++) // цикл по всем указателям</code>
<code> A[i] = new int[N]; // выделяем память на строку i</code>
<code>// работаем с матрицей A, как обычно</code>
<code>for (i = 0; i < M; i ++) // освобождаем память для всех строк</code>
<code> delete A[i];</code>
<code>delete A; // освобождаем массив указателей</code>
<code>}</code>

В рассмотренном выше случае на каждую строку выделяется свой участок памяти. Можно поступить иначе: сначала выделим область памяти сразу на всю матрицы и запишем ее адрес в **A[0]**. Затем расставим указатели так, чтобы **A[1]** указывал на **N+1**-ый элемент с начала блока (начало строки 1), **A[2]** – на **2N+1**-ый (начало строки 2) и т.д. Таким образом, в памяти выделяется всего два блока – массив указателей и сама матрица.

<code>typedef int *pInt;</code>
<code>main()</code>
<code>{</code>
<code>int M, N, i;</code>
<code>pInt *A; // указатель на указатель</code>
<code>// ввод M и N</code>
<code>A = new pInt[M]; // память на массив указателей</code>
<code>A[0] = new int [M*N]; // память для матрицы</code>
<code>for (i = 1; i < M; i ++) // расставляем указатели</code>
<code> A[i] = A[i-1] + N;</code>
<code>// работаем с матрицей</code>
<code>delete A[0]; // освобождаем матрицу</code>
<code>delete A; // освобождаем указатели</code>
<code>}</code>

6. Рекурсия

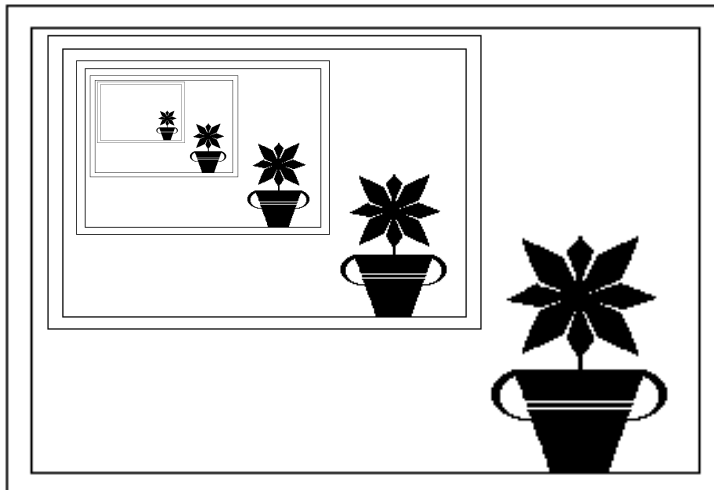
Что такое рекурсия?

Рекурсивные объекты

Всем известна сказка, которая начинается словами «У попа была собака...». Это бесконечное предложение будем называть сказкой о попе и собаке. Как же нам определить его строго математически? Оказывается, это можно сделать примерно так:

У попа была собака, он ее любил.
Она съела кусок мяса, он ее убил.
В ямку закопал, надпись написал:
Сказка о попе и собаке

Как видите, частью этой сказки является сама сказка. Так мы пришли к понятию рекурсии. Еще один пример – картинка, содержащая свое изображение.



Рекурсия – это определение объекта через самого себя.

С помощью рекурсии в математике определяются многие бесконечные множества, например множество натуральных чисел. Действительно, можно задать их так:

Натуральное число.

- 1) 1 - натуральное число.
- 2) Число, следующее за натуральным – натуральное.

Понятие факториала $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$ также можно задать рекурсивно:

Факториал.

- 1) $0! = 1$ (так условно принято).
- 2) $n! = n \cdot (n-1)!$



Рекурсивные процедуры и функции

Причем же здесь программирование? Оказывается, процедуры и функции в современных языках программирования также могут вызывать сами себя.

Рекурсивными называются процедуры и функции, которые вызывают сами себя.

Например, функцию вычисления факториала можно записать так:

```
int Factorial ( int n )
{
    if ( n <= 0 ) return 1;           // вернуть 1
    else          return n*Factorial(n-1); // рекурсивный вызов
}
```

Обратите внимание, что функция `Factorial` вызывает сама себя, если $n > 0$. Для решения этой задачи можно использовать и рекурсивную процедуру (а не функцию). Вспомним, как рекурсивная процедура может вернуть значение-результат? Через параметр, переданный по ссылке (в объявлении процедуры у его имени стоит знак ссылки `&`). При рекурсивных вызовах процедура меняет это значение.

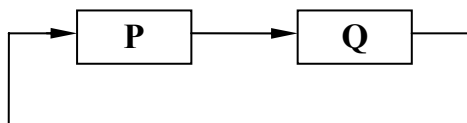
```
void Factorial ( int n, int &fact )
{
    if ( n == 0 ) fact = 1; // рекурсия закончилась
    else {
        Factorial(n-1, fact); // рекурсивный вызов, считаем (n-1)!
        fact *= n;           // n! = n*(n-1)!
    }
}
```

В отличие от функции, процедура может (если надо) возвращать несколько значений-результатов с помощью параметров, передаваемых по ссылке.



Косвенная рекурсия

Реже используется более сложная конструкция, когда процедура вызывает сама себя не напрямую, а через другую процедуру (или функцию). Это описывается следующей схемой:



Такой прием называется *косвенная рекурсия*.



Не допустим бесконечную рекурсию!

При использовании рекурсивных процедур и функций велика опасность, что вложенные вызовы будут продолжаться бесконечно (это похоже на закливание цикла `while`). Поэтому в таких функциях необходимо предусмотреть условие, которое проверяется на каждом шаге и заканчивает вложенные вызовы, когда перестает выполняться.

Для функции вычисления факториала таким условием является $n \leq 0$. Докажем, что рекурсия в примере, рассмотренном выше, закончится.

1. Рекурсивные вызовы заканчиваются, когда n становится равно нулю.
2. При каждом новом рекурсивном вызове значение n уменьшается на 1 (это следует из того, что вызывается `Factorial(n-1, ...)`).

3. Поэтому, если в начале $n > 0$, то, постепенно уменьшаясь, n достигает значения 0 и рекурсия заканчивается.

Рекурсивная процедура или функция должна содержать условие, при котором рекурсия заканчивается (не происходит нового вложенного вызова).



Когда рекурсия не нужна

При новом рекурсивном вызове компьютер делает так:

1. Запоминается состояние вычислений на данном этапе.
2. В стеке (особой области памяти) создается **новый** набор локальных переменных (чтобы не испортить переменные текущего вызова).

Поскольку при каждом вызове затрачивается новая память и расходуется время на вызов процедуры и возврат из нее, при использовании рекурсии необходимо помнить, что

глубина рекурсии (количество вложенных вызовов) должна была достаточно мала.

Программа, использующая рекурсию с большой глубиной, будет выполняться долго и может вызвать **переполнение стека** (нехватку стековой памяти). Поэтому

если задача может быть легко решена без использования рекурсии, рекурсию использовать нежелательно.

Например, задача вычисления факториала очень просто решается с помощью обычного цикла **for** (такое решение с помощью циклов называют *итеративным*, циклическим):

```
int Factorial ( int n )
{
    int i, fact = 1;
    for ( i = 2; i <= n; i ++ )
        fact *= i;
    return fact;
}
```

Эта функция работает намного быстрее, чем рекурсивная.

Доказано, что любая рекурсивная программа может быть написана без использования рекурсии, хотя такая реализация может оказаться очень сложной.

Задача. Составить функцию для вычисления чисел Фибоначчи f_i , которые задаются так:

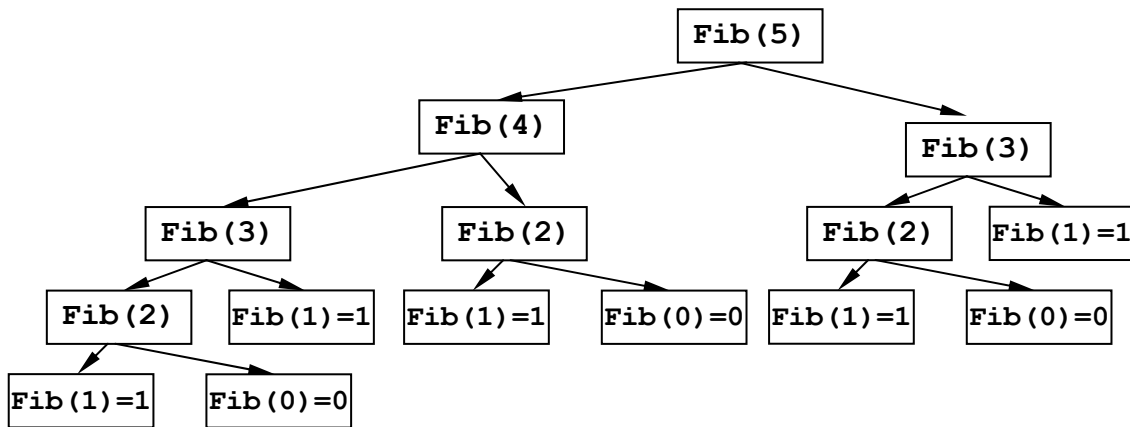
1. $f_0 = 0$, $f_1 = 1$.
2. $f_i = f_{i-1} + f_{i-2}$ для $i > 1$.

Использование рекурсии «в лоб» дает функцию

```
int Fib ( int n )
{
    if ( n == 0 ) return 0;
    if ( n == 1 ) return 1;
    return Fib(n-1) + Fib(n-2);
}
```

Заметим, что каждый рекурсивный вызов при $n > 1$ порождает еще 2 вызова функции, многие выражения (числа Фибоначчи для малых n) вычисляются много раз. Поэтому практического

значения этот алгоритм не имеет, особенно для больших n . Схема вычисления **Fib(5)** показана в виде дерева ниже:



Заметим, что очередное число Фибоначчи зависит только от двух предыдущих, которые будем хранить в переменных **f1** и **f2**. Сначала примем **f1=1** и **f2=0**, затем вычисляем следующее число Фибоначчи и записываем его в переменную **x**. Теперь значение **f2** уже не нужно и мы скопируем **f1** в **f2** и **x** в **f1**.

```

int Fib2(int n)
{
    int i, f1 = 1, f2 = 0, x;
    for (i = 2; i <= n; i++) {
        x = f1 + f2;      // следующее число
        f2 = f1; f1 = x;  // сдвиг значений
    }
    return x;
}
  
```

Такая процедура для больших n (>20) работает в сотни тысяч (!!!) раз быстрее, чем рекурсивная. Вывод:

там, где можно легко обойтись без рекурсии, надо без нее обходиться.



Рекурсивный поиск

Вспомним задачу, которая рассматривалась при обсуждении работы со строками: определить, сколько раз встречается заданное слово в предложении. Рекурсивную процедуру можно описать так:

- 1) ищем первое заданное слово с помощью функции **strstr**; если не нашли, то стоп;
- 2) количество слов = 1 + количество слов в оставшейся части строки.

```

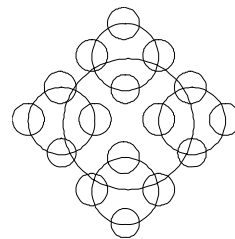
int HowMany( char *s, char *word )
{
    char *p = strstr(s, word); // ищем первое слово
    if ( ! p ) return 0;        // не нашли - 0 слов
    return 1 +                  // одно уже нашли, ...
        HowMany(p+strlen(word), word); // ищем дальше
}
  
```

Функция получилась короткая, понятная, но по скорости работы не самая эффективная.



Рекурсивные фигуры

Многие рисунки и кривые задаются рекурсивно. Рассмотрим сначала простейший пример. В центре рисунка – большая окружность. На ней находятся центры четырех окружностей меньшего диаметра, на каждой из которых – центры еще четырех окружностей еще меньшего диаметра и т.д. Всего – **N** разных диаметров (**N** уровней рекурсии).



```
void RecCircle ( float x, float y, float R, int n )
{
    float k = 0.5;
    circle ( x, y, R );           // рисуем окружность
    if ( n == 1 ) return;         // все нарисовали, выход
    RecCircle( x+R, y, k*R, n-1); // четыре рекурсивных вызова
    RecCircle( x-R, y, k*R, n-1);
    RecCircle( x, y+R, k*R, n-1);
    RecCircle( x, y-R, k*R, n-1);
}
```

Процедура для рисования такой фигуры принимает 4 параметра – координаты центра базовой окружности (**x, y**), ее радиус **R** и количество уровней **n**, которые надо нарисовать. На следующем уровне радиус изменяется в **k** раз (в примере – **k=0.5**), количество оставшихся уровней уменьшается на 1, и пересчитываются координаты для 4-х окружностей следующего уровня.

Важно доказать, что рекурсия закончится. В самом деле, как видно из процедуры, при **n = 1** рекурсия заканчивается. При каждом новом рекурсивном вызове количество уровней **n** уменьшается на 1, поэтому если в самом начале **n > 0**, то оно достигнет значения 1 и рекурсия завершится. Для большей «защиты от дурака» лучше условие окончания рекурсии записать так:

```
if ( n <= 1 ) return;
```

Если этого не сделать, рекурсия никогда не завершится (теоретически), если в начале задали **n < 1**.

Основная программа получается очень короткой – в ней надо открыть окно для графики, и один раз вызвать процедуру **RecCircle**.

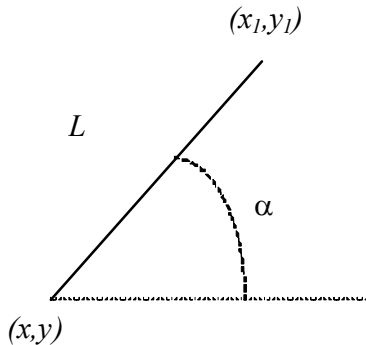
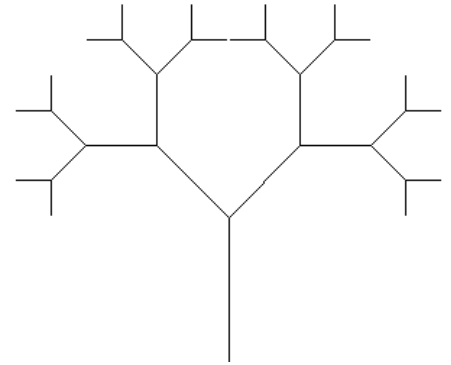
```
#include <conio.h>
#include <graphics.h>

// сюда надо вставить процедуру

main()
{
    initwindow(600, 500);
    RecCircle ( 300, 250, 100, 3 ); // 3 уровня
    getch();
    closegraph();
}
```

Пифагорово дерево

Пифагорово дерево задается так: «ствол» дерева – вертикальный отрезок длиной 1. От него вверх симметрично отходят две ветки так, чтобы угол между ними был 90 градусов, их длина равна $k*1$, где $k < 1$. Так повторяется для заданного количества уровней. На рисунке показано дерево Пифагора при $k=0.7$, имеющее 5 уровней.



то формулы для определения координат конца ветки (x_1, y_1) принимают вид:

$$x_1 = x + L * \cos(\alpha)$$

$$y_1 = y - L * \sin(\alpha)$$

Знак минус при вычислении y_1 появился из-за того, что ось y для экрана направлена вниз. Ветки следующего уровня, выходящие из вершины с координатами (x_1, y_1) , имеют углы наклона $\alpha + \pi/4$ (правая ветка) и $\alpha - \pi/4$ (левая ветка).

Не забудем также, что для использования функций **sin** и **cos** надо подключить заголовочный файл **math.h** и передавать им значения углов в радианах. Параметрами процедуры являются координаты начала базовой ветки (x, y) , ее длина **L**, угол наклона в радианах **angle** и количество уровней **n**, которые надо нарисовать.

```
void Pifagor( float x, float y, float L,
              float angle, int n)
{
    float k = 0.7, x1, y1;
    x1 = x + L*cos(angle);    // координаты второго конца
    y1 = y - L*sin(angle);
    line (x, y, x1, y1);      // рисуем ствол
    if ( n <= 1) return;      // все нарисовали, выход
    Pifagor(x1, y1, k*L, angle+M_PI_4, n-1); // рекурсивные
    Pifagor(x1, y1, k*L, angle-M_PI_4, n-1); // вызовы
}
```

Основная программа выглядит примерно так же, как и в предыдущем примере.

Перебор вариантов

Одна из главных практически важных областей, где применение рекурсии значительно упрощает решение – задачи на перебор вариантов.

Сочетания

Необходимо получить все возможные сочетания чисел от 1 до **K**, которые имеют длину **N** (в последовательности могут быть одинаковые элементы). Для решения задачи выделим в памяти массив **A[N]**. Представим себе, что все его первые **q** элементов (с номерами от 0 до **q-1**) уже определены, и надо найти все сочетания, при которых эти элементы не меняются.

Сначала ставим на место элемент с номером q . По условию на этом месте может стоять любое число от 1 до K , поэтому сначала ставим 1 и находим все варианты, при которых $q+1$ элементов уже поставлены (здесь будет рекурсивный вызов), затем ставим на это место 2 и т.д. Если $q=N$, то все элементы уже поставлены на место, и надо печатать массив – одна из нужных комбинаций готова. Ниже приведена рекурсивная процедура, которая использует массив $A[N]$ и печатает все комбинации на экране.

```
void Combinations ( int A[], int N, int K, int q )
{
    if ( q == N ) // одна комбинация получена
        PrintData ( A, N );
    else
        for (int i=1; i<=K; i++ ) {
            A[q] = i;
            Combinations(A, N, K, q+1); // рекурсивный вызов
        }
}
```

Для вывода массива на экран используется такая процедура:

```
void PrintData ( int Data[], int N )
{
    for (int i = 0; i < N; i++ )
        printf("%2d ", Data[i]);
    printf("\n");
}
```

В основной программе надо вызвать процедуру с параметром $q = 0$, поскольку ни один элемент еще не установлен:

```
#include <stdio.h>
// здесь надо разместить процедуры
main()
{
    int A[5], N = 5, K = 10;
    Combinations ( A, N, K, 0 );
}
```



Перестановки

Существует еще одна весьма непростая задача, которая хорошо решается с помощью рекурсии. Представьте, что к вам пришли N гостей. Сколько существует различных способов рассадить их за столом?

Сформулируем условие задачи математически. В массиве $A[N]$ записаны целые числа. Надо найти все возможные перестановки этих чисел (в перестановку должны входить все элементы массива по 1 разу).

Как и в предыдущем примере, предположим, что q элементов массива (с номерами от 0 до $q-1$) уже стоят на своих местах. Тогда в позиции q может стоять любой из неиспользованных элементов – их надо перебрать в цикле и вызвать рекурсивно эту же процедуру, увеличив на 1 количество установленных элементов. Чтобы не потерять никакой элемент, в цикле для всех i от q до $N-1$ будем переставлять элементы $A[i]$ и $A[q]$, генерировать все возможные комбинации, а затем менять их местами, восстанавливая исходный порядок.

Для обмена местами двух элементов массива будем использовать вспомогательную переменную **temp**.

```
void Permutations ( int A[], int N, int q )
{
    int temp;
    if ( q == N ) // перестановка получена
        PrintData ( A, N ); // вывод на экран
    else
        for (int i = q; i <= N; i ++ ) {
            temp = A[q]; A[q] = A[i]; A[i] = temp; // A[q]↔A[i]
            Permutations(A, N, q+1); // рекурсивный вызов
            temp = A[q]; A[q] = A[i]; A[i] = temp; // A[q]↔A[i]
        }
}
```



Разложение числа на сумму слагаемых

Задача. Дано целое число **S**. 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).

Самое сложное в этой задаче – сразу исключить из рассмотрения повторяющиеся последовательности. Для этого будем рассматривать только *неубывающие* последовательности слагаемых (каждое следующее слагаемое *не меньше* предыдущего).

Все слагаемые будем записывать в массив. Для этого нам понадобится массив (назовем его **A**) из **S** элементов, поскольку самое длинное представление – это сумма **S** единиц. Чтобы не расходовать память зря, лучше выделять его динамически (см. раздел про массивы).

Пусть **q** элементов массива (с номерами от 0 до **q-1**) уже стоят на своих местах, причем **A[q-1]=m**. Каким может быть следующее слагаемое – элемент **A[q]**? Поскольку последовательность неубывающая, он не может быть меньше **m**. С другой стороны, он должен быть не больше, чем **R=S-S₀**, где **S₀** – сумма предыдущих элементов.

Эти размышления приводят к следующей процедуре, которая принимает в параметрах сумму оставшейся части, массив для записи результата и количество уже определенных элементов последовательности.

```
void Split(int R, int A[], int q)
{
    int i, start = 1;
    if ( R <= 0 ) {
        PrintData ( A, q );
        return;
    }
    if ( q > 0 ) start = A[q-1]; // новый элемент не меньше A[q-1]
    for (i = start; i <= R; i ++ ) { // перебрать все до R
        A[q] = i;
        Split(R-i, A, q+1); // рекурсивный вызов
    }
}
```

Основная программа вызывает эту процедуру так (используется динамический массив):

```
#include <stdio.h>
// здесь надо поместить процедуры Split и PrintData
main()
{
    int *A, S;

    printf ( "Введите натуральное число " );
    scanf ( "%d", &S );

    A = new int[S]; // выделить память
    Split (S, A, 0); // получить все разложения
    delete A;      // освободить память
}
```



Быстрая сортировка (*QuickSort*)

Один из лучших известных методов сортировки массивов – **быстрая сортировка** Ч. Хоара (*Quicksort*) основана как раз на применении рекурсии.

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 надо – сначала поменять местами первый и последний, а затем последовательно двигаться с двух сторон. Хотя это справедливо только тогда, когда порядок элементов в самом деле обратный, подобная идея заложена в основу алгоритма *Quicksort*.

Пусть дан массив **A** из n элементов. Выберем сначала наугад любой элемент массива (назовем его **x**). Обычно выбирают средний элемент массива, хотя это не обязательно. На первом этапе мы расставим элементы так, что слева от некоторой границы находятся все числа, меньшие или равные **x**, а справа – большие или равные **x**. Заметим, что элементы, равные **x**, могут находиться в обеих частях.

$A[i] \leq x$	$A[i] \geq x$
---------------	---------------

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.

Размер обеих частей чаще всего не совпадает. Лучше всего выбирать **x** так, чтобы в обеих частях было равное количество элементов. Такое значение **x** называется *медианой* массива. Однако для того, чтобы найти медиану, надо сначала отсортировать массив, то есть заранее решить ту самую задачу, которую мы собираемся решить этим способом. Поэтому обычно в качестве **x** выбирают средний элемент массива.

Будем просматривать массив слева до тех пор, пока не обнаружим элемент, который больше **x** (и, следовательно, должен стоять справа от **x**), потом справа 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**. 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, то есть **67**. Найдем первый слева элемент массива, который больше или равен **x** и должен стоять во второй части. Это число **78**. Обозначим номер этого элемента через **i**. Теперь находим первый справа элемент, который меньше **x** и должен стоять в первой части. Это число **34**. Обозначим его номер через **j**.

78	6	82	67	55	44	34
----	---	----	----	----	----	----

i **j**

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

34	6	82	67	55	44	78
----	---	----	----	----	----	----

i **j**

Следующая пара элементов для перестановки – числа **67** и **55**.

34	6	44	67	55	82	78
----	---	----	----	----	----	----

i **j**

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

34	6	44	55	67	82	78
----	---	----	----	----	----	----

Теперь остается применить тот же способ рекурсивно к первой и второй частям массива. Рекурсия заканчивается, когда в какой-то части остается один элемент. Этот алгоритм реализован в процедуре, приведенной ниже.

```

void QuickSort ( int A[], int from, int to )
{
    int x, i, j, temp;
    if ( from >= to ) return;    // условие окончания рекурсии
    i = from;    // рассматриваем элементы с A[from] до A[to]
    j = to;
    x = A[(from+to)/2];    // выбрали средний элемент
    while ( i <= j ) {
        while ( A[i] < x ) i ++;    // ищем пару для перестановки
        while ( A[j] > x ) j --;
        if ( i <= j ) {
            temp = A[i]; A[i] = A[j]; A[j] = temp;    // перестановка
            i ++;    // двигаемся дальше
            j --;
        }
    }
    QuickSort ( A, from, j );    // сортируем левую часть
    QuickSort ( A, i, to );    // сортируем правую часть
}

```

Насколько эффективна такая сортировка? Все описанные ранее способы сортировки массивов имели порядок $O(n^2)$, то есть при увеличении в 10 раз длины массива время сортировки

увеличивается в 100 раз. Можно показать, что *Quicksort* имеет в среднем порядок $O(n \cdot \log n)$, что значительно лучше.

Чтобы почувствовать, что такое $O(n \cdot \log n)$ и $O(n^2)$, посмотрите на таблицу, где приведены числа, иллюстрирующие скорость роста для нескольких разных функций.

n	$\log n$	$n \cdot \log n$	n^2
1	0	0	1
16	4	64	256
256	8	2 048	65 536
4 096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Если считать, что числа в таблице соответствуют микросекундам, то для задачи с 1048476 элементами алгоритму со временем работы $O(\log n)$ потребуется 20 микросекунд, а алгоритму с временем работы $O(n^2)$ – более 12 дней.

Однако, этому алгоритму присущи и недостатки. Все зависит от того, насколько удачным будет на каждом шаге выбор x . В идеальном случае мы должны всегда выбирать в качестве x медиану – такой элемент, что в данном диапазоне есть равное количество меньших и больших элементов. При этом каждый раз зона сортировки сокращается в два раза и требуется всего $\log n$ проходов, общее число сравнений равно $n \cdot \log n$. В самом худшем случае (если мы каждый раз наибольшее значение из данного диапазона) он имеет скорость $O(n^2)$.

Кроме того, этот алгоритм, как и все усовершенствованные методы сортировки, неэффективен при малых n . Иногда для сортировки коротких отрезков в него включают прямые методы сортировки.

Ниже приведены результаты теста – время сортировки в секундах для разных наборов данных тремя способами (для процессора *Pentium 120*).

n	тип данных	метод пузырька	метод выбора мин. элемента	быстрая сортировка
1000	возрастающие	0.199	0.104	0.003
	убывающие	0.226	0.123	0.003
	случайные	0.235	0.120	0.004
5000	возрастающие	4.698	2.898	0.023
	убывающие	5.027	2.788	0.024
	случайные	5.275	2.953	0.024
15000	возрастающие	41.319	27.143	0.080
	убывающие	44.945	34.176	0.084
	случайные	44.780	33.571	0.068

Таким образом, для массива размером 15000 элементов метод *Quicksort* более чем в 500 раз эффективнее, чем прямые методы.

7. Структуры



Что такое структуры?

Очень часто при обработке информации приходится работать с блоками, в которых присутствуют разные типы данных. Например, информация о книге в каталоге библиотеки включает в себя автора (символьная строка), название книги (тоже строка), год издания (целое число), количество страниц (целое число) и т.п.

Для хранения этой информации в памяти не подходит обычный массив, так как в массиве все элементы должны быть одного типа. Конечно, можно использовать несколько массивов разных типов, но это не совсем удобно (желательно, чтобы все данные о книге находились в памяти в одном месте).

В современных языках программирования существует особый тип данных, который может включать в себя несколько элементов более простых (причем разных!) типов.

Структура — это тип данных, который может включать в себя несколько **полей** - элементов разных типов (в том числе и другие структуры).

Одно из применений структур — организация различных баз данных, списков и т.п.



Объявление и инициализация

Поскольку структура - это новый тип данных, его надо предварительно объявить в начале программы, например так

```
struct Book {  
    char author[40];    // автор, символьная строка  
    char title[80];     // название, символьная строка  
    int year;           // год издания, целое число  
    int pages;          // количество страниц, целое число  
};
```

При таком объявлении типа никакая память не выделяется, реальных структур в памяти еще нет. Этот фрагмент просто скажет компилятору, что в программе *могут быть* элементы такого типа. Теперь, когда надо выделить память, мы напишем

```
Book b;
```

Это оператор выделяет место в памяти под одну структуру типа **Book** и дает ей имя **b** (таким образом, **Book** — это названия типа данных (как **int** или **float**), а **b** — это имя конкретного экземпляра). При этом можно сразу записать в память нужное начальное значение всех или нескольких первых элементов структуры в фигурных скобках через запятую. Поля заполняются последовательно в порядке их объявления. Память, отведенная на поля, оставшиеся незаполненными, обнуляется.

```
Book b = {  
    "А. С. Пушкин",  
    "Полтава",  
    1998 };
```

Работа с полями структуры

Обращение по имени

Для обращения ко всей структуре используется ее имя, а для обращения к отдельному полю имя этого поля ставится через **точку**. Элементы структуры вводятся последовательно по одному. Заполнять их можно в любом порядке. С полем структуры можно работать так же, как и с переменной соответствующего типа: числовые переменные могут участвовать в арифметических выражениях, со строками можно выполнять все стандартные операции.

```
Book b;  
strcpy ( b.author, " А.С. Пушкин " );  
b.year = 1998;
```

Обращение по адресу

Пусть известен адрес структуры в памяти. Как известно, адрес может быть записан в указатель – специальную переменную для хранения адреса. Для обращения к полю структуры по ее адресу используют специальный оператор **->**.

```
Book b;      // структура в памяти  
Book *p;     // указатель на структуру  
p = &b;      // записать адрес структуры в указатель  
strcpy ( p->author, "А.С. Пушкин" ); // обращение по адресу  
p->year = 1998;
```

Ввод и вывод

Поэлементный ввод и вывод

При вводе с клавиатуры и выводе на экран или в текстовый файл с каждым полем структуры надо работать отдельно, как с обычной переменной. В приведенном примере данные вводятся в структуру **b** типа **Book** с клавиатуры и записываются в конец текстового файла **books.txt**.

```
Book b;  
FILE *fp;  
printf("Автор ");  
gets(b.author);  
printf("Название книги ");  
gets(b.title);  
printf("Год издания, кол-во страниц ");  
scanf("%d%d", &b.years, &b.pages );  
fp = fopen("books.txt", "a"); // дописать в конец файла  
fprintf("%s\n%s\n%d %d\n",  
        b.author, b.title, b.years, b.pages );  
fclose ( fp );
```




Работа с двоичным файлом

Структуры очень удобно записывать в двоичные файлы, поскольку можно за 1 раз прочитать или записать сразу одну или даже несколько структур. Вспомним, что при чтении из двоичного файла функции **fread** надо передать адрес нужной области памяти (куда записать прочитанные данные), размер одного блока и количество таких блоков. Для того, чтобы не вычислять размер структуры вручную (можно легко ошибиться), применяют оператор **sizeof**.

```
Book b;
int n;
FILE *fp;
fp = fopen("books.dat", "rb");

n = fread (&b, sizeof(Book), 1, fp);
if ( n == 0 ) {
    printf ( "Ошибка при чтении из файла" );
}

fclose ( fp );
```

Можно было также вместо **sizeof(Book)** написать **sizeof(b)**, поскольку запись **b** – это как раз один экземпляр структуры **Book**. Функция **fread** возвращает число успешно прочитанных элементов (в нашем случае – структур). Поэтому если в примере переменная **n** равно нулю, чтение закончилось неудачно и надо вывести сообщение об ошибке.

Для записи структуры в двоичный файл используют функцию **fwrite**. Ее параметры – те же, что и у **fread**. Пример ниже показывает добавление структуры в конец двоичного файла **books.dat**.

```
Book b;
FILE *fp;
// здесь надо заполнить структуру
fp = fopen("books.dat ", "ab");
fwrite(&b, sizeof(Book), 1, fp);
fclose ( fp );
```



Копирование

Задача. Пусть в памяти выделено две структуры одного типа и в одну из них записаны какие-то данные. Требуется скопировать все данные из первой структуры во вторую.

Пусть структуры имеют тип **Book** и называются **b1** и **b2**. Существуют три способа решения этой задачи. Самый сложный – копирование каждого поля отдельно:

```
Book b1, b2;
// здесь заполняем структуру b1
strcpy ( b2.author, b1.author );
strcpy ( b2.title, b1.title );
b2.year = b1.year;
b2.pages = b1.pages;
```

Можно использовать специальную функцию **memcpy**, которая умеет копировать блоки памяти. Для ее использования надо подключить к программе заголовочный файл **mem.h**.

```
#include <mem.h>
Book b1, b2;
// здесь нужно заполнить структуру b1
```



```
memcpy(&b2, &b1, sizeof(Book)); // куда, откуда, сколько байт
```

Самый простой способ – третий. Достаточно просто написать

```
b2 = b1;
```

При этом программа скопирует одну структуру в другую «бит в бит». Зачем же рассказывать про остальные два способа? Только для того, чтобы понять, как это все работает, поскольку непонимание ведет к трудноуловимым ошибкам.



Массивы структур

Структуры служат для обработки большого объема информации, поэтому чаще всего в программе используются массивы структур. Они объявляются так же, как обычно, но предварительно (выше) надо объявить саму структуру как новый тип данных.

Для обращения к полю структуры также используют точку, но теперь надо указать в квадратных скобках еще номер нужной структуры, например

```
Book A[20];  
...  
A[12].pages = 50;  
for ( i = 0; i < 20; i ++ ) // цикл по всем структурам в массива  
    puts(A[i].title);      // вывести название книги
```

Если вы работаете с двоичными файлами, то чтение и запись всего массива структур выполняется в одну строчку. Покажем приемы работы с двоичным файлом на примере задачи.

Задача. В файле **books.dat** записаны структуры типа **Book**. Известно, что их не больше 100. Требуется прочитать их в память, у всех книг установить 2009 год издания и записать обратно в тот же файл.

Поскольку по условию известно, что структур не больше 100, заранее выделяем в памяти массив на 100 структур.

```
Book b[100];
```

При чтении из файла пытаемся читать все 100 структур:

```
n = fread ( &b[0], sizeof(Book), 100, fp );
```

Чтобы определить, сколько структур было в действительности прочитано, используем значение **n**, которое функция **fread** возвращает в качестве результата. Вот полная программа:

```
#include <stdio.h>  
  
struct Book {                // объявление нового типа данных  
    char author[40];  
    char title[80];  
    int year;  
    int pages;  
};  
  
main()  
{  
    Book b[100];             // выделение памяти под массив структур  
    int i, n;  
    FILE *fp;
```

```

    fp = fopen("books.dat", "rb"); // читаем 100 структур
    n = fread(&b[0], sizeof(Book), 100, fp); // прочитали n шт.
    fclose ( fp );

    for ( i=0; i<n; i++ ) // обрабатываем все, что прочитали
        b[i].year = 2009;

    fp = fopen("books.dat", "wb"); // записываем n шт.
    fwrite ( b, sizeof(Book), n, fp );
    fclose ( fp );
}

```



Динамическое выделение памяти

Предположим, что надо создать массив структур, размер которого выясняется только во время работы программы. Для этого надо

- 1) объявить переменную типа указатель на нужный тип данных;
- 2) выделить память с помощью оператора **new** и запомнить адрес выделенного блока;
- 3) использовать новую область как обычный массив.

```

Book *B;
int n;
printf("Сколько у вас книг? ");
scanf ( "%d", &n ); // вводим размер массива
B = new Book[n];    // выделяем память
// здесь работаем с массивом B, как обычно
delete B;           // освобождаем память

```

Иногда требуется выделить в памяти одну структуру. При этом мы получаем ее адрес, который записываем в переменную-указатель. Как получить доступ к полю структуры?

Один из вариантов – «разыменовать» указатель, то есть обратиться к той структуре, на которую он указывает. Если **p** – указатель на структуру типа **Book**, то обратиться к ее полю **author** можно как **(*p).author**. Эта запись означает «мне нужно поле **author** той структуры, на которую указывает указатель **p**».

В языке Си существует и другой способ обратиться к полю структуры: можно написать и **p->author**, что значит то же самое, что и **(*p).author**, но более понятно. В следующем примере динамически выделяется память на 1 структуру, ее поля считываются с клавиатуры, и затем структура записывается в конец текстового файла **books.txt**.

```

Book *p;
FILE *fp;

p = new Book; // выделить память на 1 структуру
printf("Автор "); // ввод полей через указатель
gets ( p->author );
printf("Название книги ");
gets ( p->title );
printf("Год издания, кол-во страниц ");
scanf("%d%d", &p->year, &p->pages);

fp = fopen("books.txt", "a"); // дописать в конец файла
fprintf("%s\n%s\n%d %d\n", // обращение через указатель
        p->author, p->title, p->year, p->pages);

fclose ( fp );
delete p; // освободить память

```



Структуры как параметры процедур

Структуры, так же, как и любые другие типы, могут быть параметрами функций и процедур. В этом разделе будут показаны три способа передачи структур в процедуры и функции и описаны их отличия. Рассмотрим процедуру, которая записывает в поле **year** (год издания книги) значение 2009.



Передача по значению

Если параметр процедуры объявлен как:

```
void Year2009( Book b )
{
    b.year = 2009;
}

main()
{
    Book b;
    Year2009 ( b );
}
```

то при работе процедуры создается КОПИЯ этой структуры в стеке (специальной области памяти, где хранятся параметры и локальные переменные процедур и функций) и процедура работает с этой копией. Такой подход имеет два недостатка:

- 1) во-первых, процедура не изменяет поля структуры в вызывающей программе; это значит, что в нашем случае задача решается неверно;
- 2) во-вторых, самое главное – структуры могут быть достаточно больших размеров, и создание новой копии может привести к нехватке места в **стеке**, где создаются локальные переменные.



Передача по ссылке

Поэтому чаще всего параметры-структуры передаются в процедуры и функции *по ссылке* (при объявлении за именем типа структуры ставят знак **&**).

```
void Year2009( Book &b )
{
    b.year = 2009;
}
```

В этом случае фактически в процедуру передается *адрес* структуры и процедура работает с тем же экземпляром, что и вызывающая программа. При этом все изменения, сделанные в процедуре, остаются в силе. Работа со структурой-параметром внутри процедуры и вызов этой процедуры из основной программы никак не отличаются от предыдущего варианта.



Передача по адресу

Передача по ссылке возможна только при использовании языка Си++ (она не входит в стандарт языка Си). Тем не менее, в классическом Си можно передать в качестве параметра *адрес* структуры. При этом обращаться к полям структуры внутри процедуры надо через оператор **->**.

```

void Year2009( Book *b ) // параметр – адрес структуры
{
    b->year = 2009; // обращение по адресу (через указатель)
}
main()
{
    Book b;
    Year2009 ( &b ); // передается адрес структуры
}

```



Сортировка по ключу

Поскольку в структуры входит много полей, возникает вопрос: а как их сортировать? Это зависит от конкретной задачи, но существуют общие принципы, о которых мы и будем говорить.

Для сортировки выбирается одно из полей структуры, оно называется *ключевым полем* или просто *ключом*. Структуры расставляются в определенном порядке *по значению ключевого поля*, содержимое всех остальных полей игнорируется.

И еще одна проблема – при сортировке элементы массива меняются местами. Структуры могут быть достаточно больших размеров, и копирование этих структур будет занимать очень много времени. Поэтому

Сортировку массива структур обычно выполняют *по указателям*.

Мы уже использовали этот прием для сортировки массива символьных строк. В памяти формируется массив указателей **p**, и сначала они указывают на структуры в порядке расположения их в массиве, то есть указатель **p[i]** указывает на структуру с номером **i**. Затем остается только расставить указатели так, чтобы ключевые поля соответствующих структур были отсортированы в заданном порядке. Для решения задачи удобно объявить новый тип данных **PBook** – указатель на структуру **Book**.

```
typedef Book *PBook;
```

Пример ниже показывает сортировку массива структур по году выпуска книг. Основной алгоритм сортировки *массива указателей* заложен в процедуру **SortYear**, которая имеет два параметра – массив указателей и число структур. Здесь используется «метод пузырька».

```

void SortYear ( PBook p[], int n )
{
    int i, j;
    PBook temp; // вспомогательный указатель

    for ( i = 0; i < n-1; i ++ ) // сортировка
        for ( j = n-2; j >= i; j -- )
            if ( p[j+1]->year < p[j]->year ) // если указатели стоят
                { // неправильно, то...
                    temp = p[j]; // переставить их
                    p[j] = p[j+1];
                    p[j+1] = temp;
                }
}

```

Основная программа, которая выводит на экран информацию о книгах в порядке их выхода, выглядит примерно так:

```
#include <stdio.h>
const int N = 20;

struct Book {           // новый тип данных - структура
    char author[40];
    char title[80];
    int year;
    int pages;
};
typedef Book *PBook;    // новый тип данных - указатель на структуру

// здесь надо расположить процедуру SortYear

main()
{
    Book B[N];          // массив структур
    PBook p[N];         // массив указателей

    // здесь нужно заполнить структуры

    for ( i = 0; i < N; i ++ ) // начальная расстановка указателей
        p[i] = &B[i];

    SortYear ( p, N );   // сортировка по указателям

    for ( i = 0; i < N; i ++ ) // цикл по всем структурам
        printf("%d: %s\n",    // вывод через указатели
            p[i]->year, p[i]->title);
}
```