[12. Массивы.](#_Toc515770161)

[12.1. Понятие массив.](#_Toc515770162)

[12.2. Массивы в С. Объявление массивов.](#_Toc515770163)

[12.3. Доступ к элементам массива. Размещение массивов в памяти.](#_Toc515770164)

[12.4. Проблема ввода.](#_Toc515770165)

[12.5. Инициализация массивов.](#_Toc515770166)

[12.5.1. Инициализация по умолчанию.](#_Toc515770167)

[12.5.2. Явная инициализация.](#_Toc515770168)

[12.5.3. Инициализация безразмерных массивов.](#_Toc515770169)

[12.6. Инициализация массивов и классы памяти.](#_Toc515770170)

[12.7. Вычисление размера массива (sizeof()).](#_Toc515770171)

[12.8. Выход индекса за пределы массива.](#_Toc515770172)

[12.9. Многомерные массивы.](#_Toc515770173)

[12.9.1. Инициализация двумерного массива.](#_Toc515770174)

[12.10. Массивы в качестве аргументов функций.](#_Toc515770175)

[12.10.1. Передача массивов функциям С.](#_Toc515770176)

[12.10.2. Передача массивов функциям C++.](#_Toc515770177)

[12.11. Ввод и вывод строк.](#_Toc515770178)

[12.12. Строковые функции и символьные массивы.](#_Toc515770179)

[12.12.1. Функции gets(), puts(), fgets(), fputs() и sprintf().](#_Toc515770180)

[12.12.2. Функции strcpy(), strcat(), strncmp() и strlen().](#_Toc515770181)

[12.13. Динамическое выделение памяти.](#_Toc515770182)

[12.13.1. Функции malloc, calloc, free и операторы new и delete](#_Toc515770183)

[12.13.2. Функция memset().](#_Toc515770184)

[12.14. Динамические массивы](#_Toc515770185)

[12.14.1. Создание динамической переменной](#_Toc515770186)

[12.14.2. Создание динамического массива](#_Toc515770187)

[12.14.3. Создание одномерного динамического массива, заполненного случайными числами](#_Toc515770188)

[12.14.4. Объявление и удаление двумерного динамического массива](#_Toc515770189)

1. Массивы.

В С понятия массивов, указателей и строк взаимосвязаны. Здесь вы узнаете, как описывать и использовать массивы. Во многих книгах по С массивы и указатели обсуждаются совместно. Это не совсем правильно, поскольку в С существует множество способов использования массивов, для которых совсем не обязательно глубокое понимание указателей. Кроме того, поскольку имеется достаточно много информации о массивах как таковых, лучше не смешивать ее с обсуждением указателей. С другой стороны, указатели позволяют полностью понять, как обрабатываются массивы.

Массивы являются важнейшим средством языка, применяемым во многих программах. Их использование позволяет удобным способом размещать в памяти большое количество необходимой информации.

## Понятие массив.

Массив — это набор переменных, имеющих одно и то же базовое имя и отличающихся одна от другой числовым признаком. Например, с помощью описания

float debts [20];

объявляется, что debts — массив, состоящий из двадцати членов, или «элементов». Первый элемент массива называется debts[0], второй — debts[1], и т. д. вплоть до debts[19]. Заметим, что перечисление элементов массива начинается с 0, а не с 1. Поскольку мы объявили, что массив имеет тип float, каждому его элементу можно присвоить величину типа float. К примеру, можно писать так:

debts[5] = 32.54;

debts[6] = 1.2е+21;

Массивы могут быть образованы из данных любого типа:

int nannies[22]; /\* массив, содержащий 22 целых числа \*/

char alpha [26]; /\* массив, содержащий 26 символов \*/

long big[500]; /\*массив, содержащий 500 целых чисел типа long \*/

Раньше, например, мы говорили о строках, являющихся частным случаем массива типа char. (В общем массив типа char — это массив, элементами которого являются величины типа char. Строка — массив типа char, в котором нуль-символ ' \0' используется для того, чтобы отмечать конец строки.)

09_05

Числа, используемые для идентификации элементов массива, называются «подстрочными индексами» или просто «индексами». Индексами должны быть целые числа, и, как уже упоминалось, индексирование начинается с 0. Элементы массива размещаются в памяти последовательно, друг за другом, как показано на рисунке.

09_06

Существует огромное количество возможностей применения массивов. Ниже приводится сравнительно простой пример. Предположим, вы хотите написать программу, осуществляющую ввод 10 различных результатов спортивной игры, которые будут обрабатываться позже. Используя понятие массива, вы сможете избежать придумывания 10 различных имен переменных — по одной для каждого счета. Кроме того, для чтения данных вы можете воспользоваться циклом for:

/\* ввод счета \*/

main()

{

int i, score[10];

for (i = 0; i <= 9; i ++ )

scanf(" %d" , &a[i]); /\* ввод десяти результатов \*/

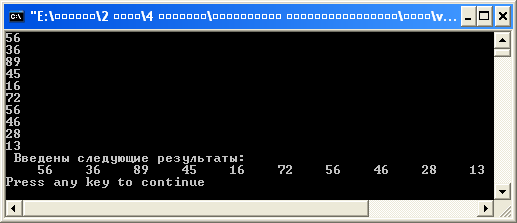
printf(" Введены следующие результаты: \n");

for (i = 0; i <= 9; i ++ )

printf(" %5d" , a[i]); /\* проверка правильности ввода \*/

printf("\n");

}



В понятие хорошего стиля программирования входит эхо-печать только что введенных величин. Она дает уверенность, что в программе будут обрабатываться те данные, для которых она предназначена.

Применяемый здесь способ гораздо более удобен, чем использование 10 различных операторов scanf() и 10 отдельных операторов printf() для ввода и проверки десяти результатов, определяющих число набранных очков. Цикл for обеспечивает очень простой и удобный способ использования индексов массивов.

Какого типа операции могли бы мы выполнить над этими данными? Мы могли бы найти их среднее, величину стандартного отклонения (мы знаем даже, как это сделать) и максимальное значение счета, а также произвести их сортировку в некотором порядке. Займемся двумя простейшими задачами: нахождением среднего и максимального результатов.

Чтобы вычислить среднее, мы можем добавить к нашей программе приведенный ниже фрагмент:

int sum, average;

for (i = 0, sum = 0; i < = 9; i ++ ) /\* две инициализации \*/

sum += a[i]; /\* суммирование элементов массива \*/

average = sum/10; /\* классический метод усреднения \*/

printf(" Средний результат равен %d.\n", average);

Для нахождения максимального результата к программе можно добавить следующий фрагмент:

int highest;

for (highest = a[0], i = 1, i < = 9; i++ )

if(a[i] > highest)

highest = a[i];

printf(" Максимальный результат равен %d.\n" , highest);

Здееь мы начинаем с того, что полагаем переменную highest равной величине а[0]. Затем производится сравнение значения highest с каждым элементом массива. Когда обнаруживается, что некоторая величина больше текущего значения переменной highest, ей присваивается эта новая большая величина.

Теперь объединим все части программы. На псевдокоде алгоритм ее работы можно записать следующим образом:

ввод результатов.

эхо-печать результатов.

вычисление и печать среднего значения.

вычисление и печать максимального значения.

Кроме того, мы несколько обобщим нашу программу:

/\* результаты \*/

#define NUM 10

main()

{

int i, sum, average, highest, score [NUM];

printf(" Укажите 10 результатов.\n");

for (i = 0; i < NUM; i++ )

scanf(" %d", &score[i]); /\* ввод десяти результатов \*/

printf(" Введены следующие результаты:\n");

for (i = 0; i < NUM; i++)

printf("%5d", score [i]); /\* проверка правильности ввода \*/

printf("\n");

for (i = 0, sum = 0; i < NUM; i++)

sum + = score [i]; /\* суммирование элементов массива \*/

average = sum/NUM; /\* классический метод усреднения \*/

printf("Средний результат равен %d.\n", average);

for (highest = score[0], i = 1; i < NUM; i++ )

if (score[i] > highest) /\* какая из величин больше \*/

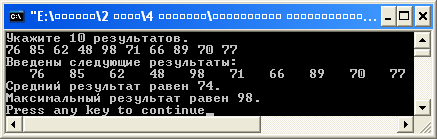
highest = score[i];

printf(" Максимальный результат равен %d.\n", highest);

}

Мы заменили число 10 символической константой и воспользовались тем, что выражения i <= (NUM — 1) и i < NUM эквивалентны.

Давайте сначала посмотрим, как это программа работает, а затем сделаем несколько замечаний.



Первый момент, который необходимо отметить, состоит в том, что мы использовали четыре различных цикла for. Вас может заинтересовать вопрос: является ли это на самом деле необходимым или мы можем некоторые из данных операций объединить в одном цикле? Такая возможность существует, и она позволила бы сделать программу более компактной. Однако мы побоялись следовать такому подходу (видите, какие мы впечатлительные люди!), поскольку это противоречит принципу модульности. Смысл, заключенный в данной фразе, состоит в том, что программа должна быть разбита на отдельные единицы, или «модули», причем каждый из них должен выполнять одну задачу. (Наша запись на псевдокоде отражает деление программы на четыре модуля.) Такое разбиение облегчает чтение текста программы. Возможно, еще более важным является то, что если отдельные части программы не перемешаны, ее коррекция или модификация упрощаются. Для этого необходимо только исключить из программы требуемый модуль, заменить его новым, а оставшуюся часть программы не изменять.

Между массивами и указателями существует очень тесная связь, поэтому обычно их рассматривают вместе. Но, прежде чем исследовать эту связь, давайте проверим наши знания о массивах и пополним их, а уж после этого перейдем к изучению связи между массивами и указателями.

Второй момент, на который необходимо обратить внимание, состоит в том, что не очень удобно иметь программу, которая обрабатывает ровно 10 чисел. Что произойдет, если кто-то выйдет из игры и будет получено только 9 результатов? Используя символическую константу для обозначения числа 10, мы упростили внесение изменений в программу, но все равно должны произвести ее компиляцию заново. Существуют ли для этого другие возможности? Мы рассмотрим их ниже.

Можно рассматривать массивы как переменные, содержащие несколько элементов одного типа. Доступ к каждому отдельному элементу данных осуществляется при помощи индекса этой переменной. В языке С массивы не являются стандартным типом данных; они представляют собой составной тип, созданный на основе других типов данных. В С возможно создавать массивы из любых типов переменных: символов, целых, чисел двойной длины, массивов, указателей, структур и так далее. В общих чертах концепции массивов и способы их использования в С и C++ совпадают.

Вы уже знаете, что массив представляет собой группу элементов одного типа. Когда нам требуется для работы массив, мы сообщаем об этом компилятору при помощи операторов описания. Для создания массива компилятору необходимо знать тип данных и требуемый класс памяти, т. е. то же самое, что и для простой переменной (называемой «скалярной»). Кроме того, должно быть известно, сколько элементов имеет массив. Массивы могут иметь те же типы данных и классы памяти, что и простые переменные, и к ним применим тот же принцип умолчания. Рассмотрим примеры, различных описаний массивов:

/\* несколько описаний массивов \*/

int temp [365]; /\* внешний массив из 365 целых чисел \*/

main ()

{

float rain [365]; /\* автоматический массив из 365 чисел типа float \*/

static char code [12]; /\* статический массив из 12 символов \*/

extern temp[]; /\* внешний массив; размер указан выше \*/

}

Как уже упоминалось, квадратные скобки ([]) говорят о том, что temp и все остальные идентификаторы являются именами массивов, а число, заключенное в скобки, указывает количество элементов массива. Отдельный элемент массива определяется при помощи его номера, называемого также индексом. Нумерация элементов начинается с нуля, поэтому temp[0] является первым, а temp[364] последним 365-элементом массива temp.

Но все это вам уже должно быть известно, поэтому изучим что-нибудь новое.

## Массивы в С. Объявление массивов.

Массивы имеют четыре основных характеристики:

* Отдельные объекты данных в массиве называются элементами.
* Все элементы массива должны иметь одинаковый тип данных.
* Все элементы располагаются в памяти компьютера последовательно, и индекс первого элемента равен нулю.
* Имя массива является постоянным значением, представляющим собой адрес первого элемента массива.

Так как предполагается, что все элементы массива имеют одинаковый размер, нельзя описывать массивы со смешанными типами данных. Без этого ограничения было бы очень сложно определять местоположение конкретного элемента в массиве. Поскольку все элементы имеют одинаковый размер, и эта особенность используется для нахождения заданного элемента, из этого следует то, что все элементы хранятся в памяти компьютера последовательно (при этом младший адрес соответствует первому элементу, а старший адрес занимает последний элемент). Это означает, что между элементами нет промежутков, и они физически соседствуют в памяти компьютера.

Можно создавать массивы внутри массивов, то есть многомерные массивы. На практике, если элемент массива является структурой, то смешанные типы данных могут существовать в массиве, располагаясь внутри элементов структуры.

И, наконец, имя массива представляет собой постоянное значение, которое нельзя изменить во время выполнения программы. По этой причине массивы нельзя использовать как lvalue. Выражения lvalue представляют собой адреса памяти, содержание которых программа может менять; они часто используются в левой части операторов присваивания. Если бы имена массивов были допустимыми lvalue-выражениями, программа могла бы менять их значение. В результате изменился бы начальный адрес самого массива. Это может показаться мелочью, однако некоторые виды выражений, кажущиеся с виду допустимыми, запрещены в С. Все программисты на С вероятно знакомы с этими тонкостями, однако будет полезным, если вы будете понимать, откуда они берутся.

Ниже приведены примеры объявлений массивов:

int iarray[12]; /\* массив из 12 целых чисел \*/

char carray[20]; /\* массив из 20 символов \*/

Так же, как и во всех объявлениях данных в С, объявление массива начинается с указания его типа, затем следуют его имя и пара квадратных скобок, в которые заключено выражение типа константа, определяющее размер массива. В квадратных скобках допускается использование имени переменной. В этом случае нужно обязательно указать размер массива до начала выполнения программы. Выражение должно преобразовываться в значение константы для того, чтобы компилятор точно знал объем памяти, занимаемый массивом.

Для указания размера массива лучше всего использовать константы, определенные посредством #define:

#define iARRAY\_MAX 20

#define fARRAY MAX 15

int iarray[iARRAY\_MAX];

float farray[fARRAY MAX];

Такой подход гарантирует, что при последующих обращениях к массиву не будет превышен его указанный размер. Например, для обращения к элементам массива очень часто используется цикл for:

#include <stdio.h>

#define iARRAY\_MAX 20

int iarray[iARRAY\_MAX];

main ()

{

int i;

for(i = 0; i < iARRAY\_MAX; i++) {

...

...

...

}

return(0);

}

## Доступ к элементам массива. Размещение массивов в памяти.

Обычно при объявлении некоторой переменной резервируется одна или несколько ячеек памяти, и посредством таблицы ссылок с этой ячейкой связывается некоторое имя, которое можно использовать для доступа к ячейкам памяти. К примеру, в следующем описании в оперативной памяти резервируется только одна ячейка для целого числа и с ней связывается имя ivideo\_tapes (верхняя часть рис. 9.1):

int ivideo\_tapes;

С другой стороны, при следующем описании резервируются семь последовательных ячеек памяти, с которыми связывается имя ivideo\_library (нижняя часть рис. 9.1):

int ivideo\_library[7] ;

09_01

Рис. 9.1. Хранение переменных и массивов в памяти

Поскольку все элементы массива должны иметь один и тот же тип данных, каждая из семи ячеек массива ivideo\_library может хранить одно целое число.

Посмотрим на отличия между доступом к отдельной ячейке, связанной с переменной ivideo\_tapes, и доступом к семи ячейкам с массивом ivideo\_library. Для доступа к ячейке, связанной с переменной, достаточно указать имя переменной — ivideo\_tapes. При работе с массивом ivideo\_library для точного определения одной из семи ячеек необходимо при обращении указывать некоторый индекс. Следующие операторы адресуют элементы массива — с первого до последнего:

ivideo\_library[0];

ivideo\_library[1];

ivideo\_library[2];

ivideo\_library[3];

...

…

…

ivideo\_library[6];

При обращении к некоторому элементу массива целое число, заключенное в квадратные скобки, является индексом, указывающим смещение или разность между адресуемой и первой ячейками.

При работе с массивами квадратные скобки используются в двух совершенно различных случаях. При описании массива в квадратных скобках указывается число элементов:

int ivideo\_library[7];

Однако при обращении к некоторому элементу массива используются его имя и индекс, заключенный в квадратные скобки:

ivideo\_library[3] ;

Если отталкиваться от приведенного выше объявления массива ivideo\_library, то следующий оператор будет логически неверным:

ivideo\_library[7] = 53219;

Это не будет допустимой ссылкой к элементу массива ivideo\_library. Оператор пытается адресовать ячейку со смещением 7 относительно первого элемента, то есть восьмую ячейку. Происходит ошибка, так как массив состоит только из семи элементов. Программист обязан следить, чтобы индексное выражение оставалось в пределах границ массива.

Рассмотрим объявления:

#define iDAYS\_OF\_WEEK 7

int ivideo\_library[iDAYS\_OF\_WEEK];

int iweekend = 1;

int iweekday =2;

Посмотрим, что произойдет в следующих выполняемых операторах:

ivideo\_library[2] ;

ivideo\_library[iweekday];

ivideo library[iweekend + iweekday];

ivideo\_library[iweekday - iweekend];

ivideo\_library[iweekend - iweekday];

Оба первых оператора обращаются к третьему элементу массива. В первом операторе для этого используется константа, во втором — переменная. Последующие три оператора иллюстрируют использование в качестве индекса выражений, при условии, что они дают допустимое целочисленное значение индекса. Третий оператор имеет значение индекса, равное 3, и обращается к четвертому элементу массива. В четвертом операторе индексное значение равно 1, и он обращается ко второму элементу массива. Последний оператор — неправильный, поскольку значение индекса, —1, недопустимо.

К любому элементу массива можно обращаться, не зная истинного размера каждого элемента. Например, вы хотите адресовать третий элемент целочисленного массива ivideo\_library. Вспомним, что в различных системах под данные одного и того же типа может выделяться различное число ячеек памяти. На одном компьютере целое число может занимать 2 байта памяти, а на другом — 4. На любом компьютере для обращения к третьему элементу массива можно использовать выражение ivideo\_library[2]. Индексное значение указывает смещение в виде числа элементов, вне зависимости от количества отведенных для хранения разрядов памяти.

Адресация со смещением верна и для массивов других типов. В одной системе для целых чисел требуется памяти в два раза больше, чем для символов, а в другой — в четыре раза больше. В любом случае, для доступа к четвертому элементу либо целочисленного массива, либо символьного массива, нужно использовать значение индекса, равное 3.

В повседневной жизни постоянно приходится сталкиваться с однотипными объектами. Как и многие другие языки высокого уровня, C++ предоставляет программисту возможность работы с наборами однотипных данных - массивами. Отдельная единица таких данных, входящих в массив, называется элементом массива. В качестве элементов массива могут выступать данные любого типа (один тип данных для каждого массива), а также указатели на однотипные данные. Массивы бывают одномерными и многомерными.

Поскольку все элементы массива имеют один тип, они также обладают одинаковым размером. Использованию массива в программе предшествует его объявление, резервирующее под массив определенное количество памяти. При этом указывается тип элементов массива, имя массива и его размер. Размер сообщает компилятору, какое количество элементов будет размещено в массиве. Например:

int Array[20];

зарезервирует в памяти место для размещения двадцати целочисленных элементов.

Элементы массива в памяти располагаются непосредственно один за другим. На рис. 0601 показано расположение одномерного массива двухбайтных элементов (типа short) в памяти.

Обращение к элементам массива может осуществляться одним из двух способов:

* по номеру элемента в массиве (через его индекс);
* по указателю.

При обращении через индекс за именем массива в квадратных скобках указывается номер элемента, к которому требуется выполнить доступ. Следует помнить, что в C++ элементы массива нумеруются начиная с 0. Первый элемент массива имеет индекс 0. второй - индекс 1 и т.д. Таким образом, запись типа:

х = Array[13];

y = Array[19];

выполнит присвоение переменной х значения 14-го элемента, а переменной у-значение 20-го элемента массива.



Рис. 0701 Одномерный массив в памяти

Доступ к элементам массива через указатели заключается в следующем. Имя объявляемого массива ассоциируется компилятором с адресом его самого первого элемента (с индексом 0). Таким образом, можно присвоить указателю адрес нулевого элемента, используя имя массива:

char ArrayOfChar[] = {'W', 'O', 'R','L','D'};

char\* pArr = ArrayOfChar;

Разыменовывая указатель pArr. можно получить доступ к содержимому ArrayOfChar[0]:

char Letter = \*pArr;

Поскольку в C++ указатели и массивы тесно взаимосвязаны, увеличивая или уменьшая значение указателя на массив, программист получает возможность доступа ко всем элементам массива путем соответствующей модификации указателя:

// pArr указывает на ArrayOfChar[0] ('W')

pArr += 3;

// pArr указывает на ArrayOfChar[3] ('L')

pArr ++;

// pArr указывает на ArrayOfChar[4] ('D')

char Letter = \*pArr; // Letter = 'D';

Таким образом, после проведенных арифметических операций указатель pArr будет ссылаться на элемент массива с индексом 4. К этому же элементу можно обратиться иным образом:

Lettеr = \*(ArrayOfCher +4);

// Эквивалент Letter = ArrayOf Char [4];

Присвоение значений одного массива значениям другого массива вида Array[] = Another [] или Array = Another недопустимо, так как компилятор не может самостоятельно скопировать все значения одного массива в значения другого. Для этого программисту необходимо предпринимать определенные действия (при доступе к элементам по индексу чаще всего используется циклическое присвоение). Объявление вида

char (\*Array)[10];

определяет указатель Array на массив из 10 символов (char). Если же опустить скобки, компилятор поймет запись как объявление массива из 10 указателей на тип char. Рассмотрим пример использования массива.

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

// Объявление целочисленного массива,

// содержащего 5 элементов

short Number[5];

char endl = '\n';

// Заполнение всех элементов маccива

// в цикле

for(int i= 0; i< 5; i++)

Number[i] = i;

//

//…

//

// Вывод содержимого а 3-го

// по 5-й элемента

for(int i=2; i<5; i++)

cout << Number[i] << endl;

getchar(); getchar();

return 0;

}



## Проблема ввода.

Существует несколько способов последовательного ввода набора данных, скажем чисел. Мы обсудим здесь некоторые из них, переходя от менее удобных к более удобным.

Вообще говоря, наименее удобный способ — это тот, который мы только что использовали; написание программы, допускающей ввод фиксированного числа элементов данных. (Такой способ, однако, прекрасно подходит для тех ситуаций, когда число входных данных никогда не изменяется.) Если число входных элементов данных изменяется, необходимо осуществить повторную компиляцию программы.

Следующий шаг состоит в том, чтобы спросить у пользователя, сколько элементов данных будет введено. Так как размер массива в программе фиксирован, она должна проверить, не превышает ли величина, содержащаяся в ответе пользователя, размер массива. Затем пользователь может начать ввод данных. Тогда начало нашей программы можно переписать следующим образом:

printf(" Сколько элементов данных вы будете вводить? \n");

scanf(" %d" , &nbr);

while(nbr > NUM)

{

printf("Я смогу обрабатывать не больше %d элементов; пожалуйста, укажите");

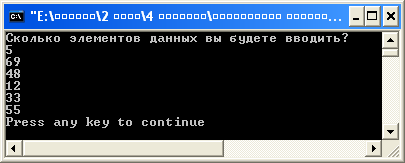
printf("меньшую величину.\n" , NUM);

scanf("%d", &nbr);

} /\* гарантирует, что nbr < = NUM — максимального размера массива \*/

for(i = 0; i < nbr; i++)

scanf("%d", &score[i]);



Мы можем продолжить движение в этом направлении, заменяя в каждом случае символическую константу NUM в программе (исключая наличие ее в директиве #define и в описании массива) переменной nbr. При таком способе различные операции будут выполняться только над теми элементами массива, в которые введены данные.

Недостатком указанного подхода является лежащее в его основе предположение, что пользователь не ошибается при подсчете элементов; если же при программировании полагаться на то, что пользователь всегда все делает правильно, программы оказываются ненадежными.

Это подводит нас к следующему методу, при котором в программе осуществляется подсчет количества вводимых чисел. После всего сказанного выше очевидно, что у компьютеров имеются для этого все возможности. Основная проблема здесь состоит в том, как сообщить компьютеру о завершении ввода чисел. Один из методов — дать пользователю возможность вводить специальный признак, указывающий на конец ввода. Признак должен принадлежать к данным того же типа, что и остальные вводимые данные, так как он должен быть прочитан тем же оператором программы. Но при этом он должен отличаться от обычных данных. К примеру, если бы мы вводили результаты игры, чтобы узнать, кто набрал от 0 до 100 очков, мы не могли бы выбрать число 74 в качестве такого признака, потому что оно может соответствовать некоторому возможному результату. С другой стороны, например, число 999 или — 3 вполне могло бы подойти в качестве такого признака, поскольку оно не соответствует требуемому результату. Ниже приводится программа, являющаяся реализацией этого метода:

#define STOP 999 /\* признак завершения ввода \*/

#define NUM 50

main()

{

int i, count, temp, score [NUM];

printf(" Начните ввод результатов. Введите 999 для указания\n");

printf(" конца ввода. Максимальное число результатов, которое вы\n");

printf(" можете ввести — это %d.\n", NUM);

count = 0;

scanf(" %d", &temp); /\* ввод величины \*/

while(temp != STOP && count <= NUM /\* проверка наличия признака STOP \*/

{

/\* и проверка, не произошло ли превышения

размера массива \*/ score[count++ ] — temp; /\* запись величины в память

и коррекция счетчика \*/ if (count < NUM + 1)

scanf("%d", &temp); /\* ввод очередного результата \*/ else

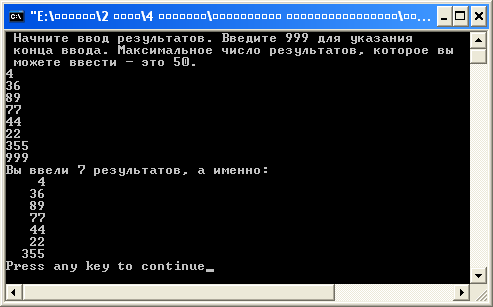
printf("fl не могу принять больше данных.\n");

}

printf("Bы ввели %d результатов, а именно:\n", count); for (i = 0; i < count; i++ )

printf("%5d\n", score[i]);

}



Мы вводим данные во временную переменную temp и присваиваем ее значение соответствующему элементу массива только в том случае, если оно не является признаком конца ввода. Совершенно не обязательно реализовывать все именно так; мы просто считаем, что указанный способ делает процесс проверки несколько более наглядным.

Обратите внимание на то, что проверяется выполнение двух условий: прочитан ли признак конца ввода и есть ли место в массиве для следующего числа. Если мы заполнили массив данными до того, как указали признак конца ввода, программа вежливо сообщает нам об этом и прекращает ввод данных.

Заметьте также, что мы воспользовались постфиксной формой операции увеличения. Поэтому, когда значение count равно 0, элементу массива score [0] присваивается величина переменной temp, а затем count возрастает на 1. После каждой итерации цикла while величина счетчика count становится на единицу больше последнего использованного индекса массива. Это как раз то, что нам нужно, поскольку score [0] — первый элемент, score [20] — 21-й элемент и т. д. Когда работа цикла в программе завершается, значение count оказывается равным полному числу прочитанных элементов данных. Затем величина count используется в качестве верхней границы числа итераций для последующих циклов.

Этот алгоритм хорошо работает, пока у нас имеется запас таких чисел, которые никогда не будут вводиться как данные. Но что делать, если мы захотим иметь программу, допускающую ввод в качестве данных любых чисел, относящихся к некоторому определенному типу? В таком случае мы не сможем использовать ни одно из чисел как признак конца ввода.

Мы столкнулись с аналогичной проблемой, когда искали подходящий символ для признака End-of-File. Тогда было принято решение использовать для ввода символов специальную функцию (getchar()), которая при обращении к ней фактически возвращала величину типа int. Это позволяло функции читать «символ» EOF, который на самом деле не был обычным символом. В рассматриваемом нами примере полезной оказалась бы функция, которая осуществляла бы ввод целых чисел, могла бы, кроме того, читать данные не только целого типа, но и использовать их в качестве признака конца ввода.

Мы можем одновременно и обрадовать и огорчить вас: такое решение оказывается возможным, но вы должны узнать несколько больше о работе функций; поэтому обсуждение данной идеи откладывается.

**Резюме.**

Основной темой было обсуждение возможностей управления ходом выполнения программы. Язык Си предоставляет много средств для структурирования программ. С помощью операторов while и for реализуются циклы с предусловием. Второй оператор особенно подходит для циклов, включающих в себя инициализацию и коррекцию переменной. Использование операции «запятая» в цикле for позволяет инициализировать и корректировать более одной переменной. Для тех редких случаев, когда требуется использовать цикл с постусловием, язык Си предоставляет оператор do while. Операторы break, continue и goto обеспечивают дополнительные возможности управления ходом выполнения программы.

Мы обсудили здесь также и понятие массива. Массивы в программе описываются так же, как обычные переменные, но при этом в квадратных скобках указывается число элементов. Первому элементу массива присваивается номер 0, второму — номер 1 и т. д. Индексы, используемые для нумерации элементов массива, могут обрабатываться обычным образом при помощи циклов.

## Инициализация массивов.

Инициализацию массивов, содержащих элементы базовых типов, можно производить при их объявлении. При этом непосредственно после объявления необходимо за знаком равенства (=) перечислить значения элементов в фигурных скобках через запятую (,) по порядку их следования в массиве. Например, выражение:

int Temp[12] = {2, 4, 7, 11, 12, 12, 13, 12, 10, 8, 5, 1};

проинициализирует массив температур Temp соответствующими значениями. Так, элемент Temp[0] получит значение 2, элемент Temp [1] - значение 4 и т.д. до элемента Temp [11] с присвоением ему значения 1.

Если в списке инициализации значений указано меньше, чем объявлено в размере массива, имеет место частичная инициализация. В этом случае иногда после последнего значения в инициализирующем выражении для наглядности ставят запятую:

int Temp[12] = {2, 4, 7, };

Таким образом, элементы Temp[0], Теmр[1] и Теmр[2] будут проинициализированы, в то время как оставшиеся элементы инициализации не получат.

Имеется три способа инициализации массивов:

* По умолчанию во время их создания. Применимо только к глобальным и статическим (static) массивам.
* Явно во время создания при помощи констант инициализации.
* Во время выполнения программы при присваивании или копировании данных в массив.

Для инициализации массива во время создания можно использовать только константы. Если элементы массива должны получать значения из переменных, то в программном коде должны быть явные операторы инициализации массива.

### Инициализация по умолчанию.

В стандарте ANSI С оговорено, что массивы бывают либо глобальные (описанные вне main() и любых других функций), либо автоматические static (статические, но описанные после какой-либо открывающей скобки), и при отсутствии инициализирующей информации они всегда получают значения двоичных нулей. В С числовые массивы инициализируются значением ноль. (Массивы указателей получают начальные значения null). Можно запустить следующую программу, чтобы убедиться в том, что любой компилятор С удовлетворяет этому стандарту:

/\*09INITAR.C

Программа на С, проверяющая инициализацию массива\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iGLOBAL\_ARRAY\_SIZE 10

#define iSTATIC\_ARRAY\_SIZE 10

int iglobal\_array[iGLOBAL\_ARRAY\_SIZE]; /\*глобальный массив\*/

int main ()

{

int i;

static int static\_iarray[iSTATIC\_ARRAY\_SIZE]; /\*массив static\*/

int iarray[3]={2,1,0};

char carray[3]={'a','s','d'};

char sarray1[8]="Please\n";

char sarray2[]="Thenks\n";

for(i=0;i<=9;i++)

printf("iglobal\_array[%2d]=%d static\_iarray[%2d]=%d\n",

i,iglobal\_array[i],i,static\_iarray[i]);

for(i=0;i<=2;i++)

printf("iarray[%2d]=%d carray[%2d]=%c\n",i,iarray[i],i,carray[i]);

for(i=0;i<=6;i++)

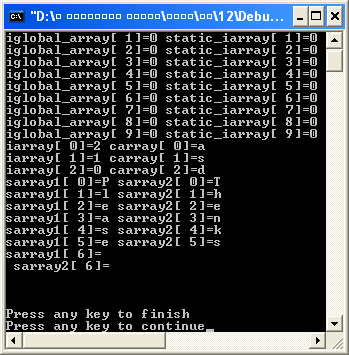
printf("sarray1[%2d]=%c sarray2[%2d]=%c\n",i,sarray1[i],i,sarray2[i]);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



При запуске программы вы должны увидеть на экране нули — это свидетельствует о том, что массивы обоих типов были автоматически инициализированы. Также в программе отображен еще один важный момент: первый индекс всех массивов в С равен нулю. В отличие от других языков, в С нельзя сделать первый индекс равным 1. Если вам интересно почему, вспомните о тесной связи языка С с ассемблером. В ассемблере первый элемент в таблице всегда имеет нулевое смещение.

### Явная инициализация.

Аналогично описаниям и инициализации переменных типов int, char, float, double и других вы можете инициализировать массивы. Стандарт ANSI С позволяет задавать и инициализирующие значения любого массива, глобального или иного, описанного в любой части программы. Следующий фрагмент кода иллюстрирует описание и инициализацию четырех массивов:

int iarray[3] = {-1, 0, 1};

static float fpercent[4] = {1.141579, 0.75,55E0, -.33E1} ;

static int idecimal[3] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

char cvowels[] = {'A', 'a', 'E', 'e', 'I', 'i', 'O', 'o', 'U', 'u'};

Первый оператор объявляет массив iarray из трех целых чисел и в фигурных скобках, через запятые, задает значения его элементов. Как обычно, оператор заканчивается точкой с запятой. Результат этих действий следующий: когда откомпилированная программа загружается в память компьютера, ячейки памяти, отведенные под массив iarray, уже содержат начальные значения, поэтому их не нужно инициализировать во время выполнения программы. Важно понять то, что это не просто удобство, а разнесение действий во времени. Если программа все же меняет значения массива iarray, они так и остаются измененными. Многие компиляторы позволяют инициализировать только глобальные или статические (static) массивы, как во втором операторе, который дает начальные значения массиву fpercent при загрузке всей программы.

В третьем операторе при объявлении массива задано ошибочное число элементов. Многие компиляторы рассматривают это как ошибку, другие — резервируют нужное место, учитывая большее значение: либо размер массива, либо количество начальных значений. Этот оператор вызовет сообщение об ошибке компилятора Visual C/C++, указывающее на слишком большое число начальных значений. Если же размер массива будет превышать количество начальных значений, то они будут присвоены первым элементам массива, а остальные станут нулевыми. Все это означает, что нет необходимости считать значения, если вы определяете их все без исключения. Если размер массива не указан, как в четвертом операторе, количество начальных значений и определяет размер массива.

### Инициализация безразмерных массивов.

Для большинства компиляторов безразлично — указываете ли вы размер массива, или задаете список фактических значений; важно указать либо то, либо другое. Например: в программах часто описывается набор собственных сообщений об ошибках. Это можно сделать двумя способами. Вот первый из них:

char szInput\_Error[37] = "Please enter a value between 0 — 9:\n";

/\*Пожалуйста, введите число от 0 до 9\*/

char szDevice\_Error[16] = "Disk not ready\n";

/\*Диск не готов\*/

char szMomtor\_Error[32] = "Program needs a color monitor.\n";

/\*Программе требуется цветной монитор\*/

char szWarning[44]="This operation will erase the active file!\n";

/\*Данная операция уничтожит активный файл\*/

При таком подходе необходимо считать количество символов в строке, не забывая добавить 1, чтобы учесть невидимый null-символ окончания строки \0. Это, в лучшем случае, очень утомительный для глаз метод, который чреват многими ошибками. При втором способе, показанном ниже, используются безразмерные массивы, и язык С автоматически определяет их размер:

char szInput\_Error[] = "Please enter a value between 0 — 9:\n";

char szDevice\_Error[] = "Disk not ready\n";

char szMomtor\_Error[] = "Program needs a color monitor.\n";

char szWarning[] = "This operation will erase the active file!\n";

Всякий раз, когда встречается оператор инициализации массива и размер массива не указан, компилятор автоматически выделяет достаточно места для всех указанных данных.

При инициализации массивов неопытный программист может столкнуться с несколькими "подводными камнями". Например: массив, размер которого не указан, а список значений отсутствует, имеет нулевую длину. Если вслед за массивом следуют объявления других данных, то оказывается, что имя массива нулевой длины ссылается на адрес этих данных, следовательно, дальнейшая передача значений в нулевой массив приведет к их записи по адресам, выделенным другим переменным.

Кроме этого, инициализировать следует не только одномерные безразмерные массивы. В случае многомерных массивов в С для их правильной инициализации нужно указывать все их размерности, кроме самой левой (то есть допускается запись типа int iarray[][3]—{{1,2,3},...,{100,200,300} } - n-ное число массивов из трех целых чисел; обратное написание недопустимо, так как длина массива не может быть неопределенной). В этом случае можно строить таблицы переменной длины, при этом компилятор автоматически выделит нужный объем памяти.

## Инициализация массивов и классы памяти.

Для хранения данных, необходимых программе, часто используют массивы. Например, в массиве из 12 элементов можно хранить информацию о количестве дней каждого месяца. В подобных случаях желательно иметь удобный способ инициализации массива перед началом работы программы. Такая возможность, вообще говоря, существует, но только для статической и внешней памяти. Давайте посмотрим, как она используется.

Мы знаем, что скалярные переменные можно инициализировать в описании типа при помощи таких выражений, как, например:

int fix = 1;

float flax = PI\*2;

при этом предполагается, что PI — ранее введенное макроопределение. Можем ли мы делать что-либо подобное с массивом? Ответ не однозначен: и да, и нет.

Внешние и статические массивы можно инициализировать. Автоматические и регистровые массивы инициализировать нельзя.

В многомодульном проекте инициализация массива при объявлении производится лишь в одном из модулей. В других модулях получить доступ к элементам массива можно с помощью ключевого слова extern:

// Модуль first.ерр

…

char Hello [] = {'Н', 'е', 'l', 'l', 'о'};

…

// Модуль second.срр

extern Hello[];

…

// Модуль n. cрр

extern Hello[5];

…

Попытка повторной инициализации вызовет сообщение компилятора об ошибке.

Прежде чем попытаться инициализировать массив, давайте посмотрим, что там находится, если мы в него ничего не записали.

/\* проверка содержимого массива \*/

main ()

{

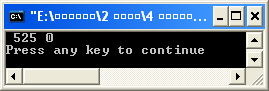
int fuzzy[2]; /\* автоматический массив \*/

static int wuzzy[2]; /\* статический массив \*/

printf(" %d %d\n" , fuzzy[1], wuzzy[1]);

}

Программа напечатает



Полученный результат иллюстрирует следующее правило:

Если ничего не засылать в массив перед началом работы с ним, то внешние и статические массивы инициализируются нулем, а автоматические и статические массивы содержат какой-то «мусор», оставшийся в этой части памяти.

Прекрасно! Теперь мы знаем, что нужно предпринять для обнуления статического или внешнего массива — просто ничего не делать. Но как быть, если нам нужны некоторые значения, отличные от нуля, например количество дней в каждом месяце. В этом случае мы можем делать так:

/\* дни месяца \*/

int days[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

main ()

{

int index;

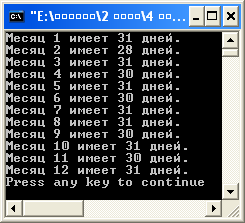
extern int days[]; /\* необязательное описание \*/

for (index = 0; index < 12; index++)

printf(" Месяц %d имеет %d дней.\n", index + 1, days [index]);

}

Результат:



Программа не совсем корректна, поскольку она выдает неправильный результат для второго месяца каждого четвертого года.

Определив массив days[] вне тела функции, мы тем самым сделали его внешним. Мы инициировали его списком, заключенным в скобки, используя при этом запятые для разделения элементов списка.

Количество элементов в списке должно соответствовать размеру массива. А что будет, если мы ошиблись в подсчете? Попробуйте переписать последний пример, используя список, который короче, чем нужно (на два элемента):

/\* дни месяца \*/

int days[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31};

main ()

{

int index;

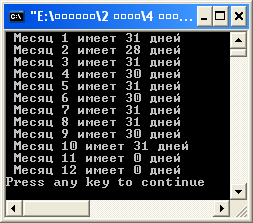
extern int days[]; /\* необязательное описание \*/

for (index = 0; index < 12; index++ )

printf(" Месяц %d имеет %d дней\n", index + 1, days [index]);

}

В этом случае результат оказывается иным:



Можно видеть, что у компилятора не возникло никаких проблем: просто, когда он исчерпал весь список с исходными данными, то стал присваивать всем остальным элементам массива нулевые значения.

Однако в случае излишне большого списка компилятор будет также не столь «великодушен» к вам, поскольку посчитает выявленную избыточность ошибкой. Поэтому нет никакой необходимости заранее подвергать себя «насмешкам» компилятора. Надо просто выделить массив, размер которого будет достаточен для размещения списка:

/\* дни месяца \*/

int days[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31};

main ()

{

int index;

extern int days[]; /\* необязательное описание \*/

for (index = 0; index < sizeof days/(sizeof (int)); index++ )

printf(" Месяц %d имеет %d дней\n" , index + 1, days [index]);

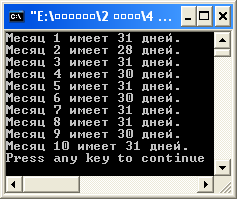
}

К этой программе следует сделать два существенных замечания.

Первое: если вы используете пустые скобки для инициализации массива, то компилятор сам определит количество элементов в списке и выделит для него массив нужного размера.

Второе: оно касается добавления, сделанного в управляющем операторе for. He полагаясь (вполне обоснованно) на свои вычислительные способности, мы возложили задачу подсчета размера массива на компилятор. Оператор sizeof определяет размер в байтах объекта или типа, следующего за ним. В нашей вычислительной системе размер каждого элемента типа int равен двум байтам, поэтому для получения количества элементов массива мы делим общее число байтов, занимаемое массивом, на 2. Однако в других системах элемент типа int может иметь иной размер. Поэтому в общем случае выполняется деление на значение переменной sizeof (для элемента типа int).

Ниже приведены результаты работы нашей программы:



Ну вот, теперь мы получаем точно 10 значений. Наш метод, позволяющий программе самой находить размер массива, не позволил нам напечатать конец массива.

Существует и более короткий способ инициализации массивов, но поскольку он применим только к символьным строкам, мы рассмотрим его в следующей главе.

В заключение мы покажем, что можно присваивать значения элементам массива, относящегося к любому классу памяти. Например, в приведенном ниже фрагменте программы присваиваются четные числа элементам автоматического массива:

/\* присваивание значений массиву \*/

main ()

{

int counter, evens [50];

for(counter = 0; counter < 50; counter++)

evens [counter] = 2 \* counter;

...

}

## Вычисление размера массива (sizeof()).

Как вы уже знаете, операция sizeof() возвращает физический размер в байтах того объекта, к которому она применяется. Ее можно использовать с объектами любых типов, за исключением битовых полей. Часто операция sizeof() применяется для определения физического размера переменной в тех случаях, когда размер переменных этого типа может меняться от одного компьютера к другому. Вы уже видели, что в зависимости от используемой системы целые числа могут иметь длину 2 или 4 байта. Если у операционной системы запрашивается дополнительная память для размещения семи целых чисел, то необходим способ для определения количества памяти: либо 14 байт (7x2 байта на число), либо 28 байт (7x4 байта на число).

При объявлении одномерного массива с одновременной его инициализацией разрешается опускать значение размера, обычно указываемое в квадратных скобках. При этом компилятор самостоятельно подсчитает количество элементов в списке инициализации и выделит под них необходимую область памяти:

// Выделение в памяти места для хранения

// шести объектов типа int

// (24 байта для 32-разрядной системы)

int Even [] = {0, 2, 4, 5, 8, 10};

Если далее в программе потребуется определить, сколько элементов имеется в массиве, можно воспользоваться следующим выражением:

Int Size = sizeof(Even)/sizeof(Even[0]);

Здесь выражение sizeof(Even) определяет общий размер, занимаемый массивом Even в памяти (в байтах), а выражение sizeof(Even[0]) возвращает размер (тоже в байтах) одного элемента массива.

В следующей программе вопрос вычисления размера массива решается автоматически (и печатается значение 14 для тех систем, в которых для целого числа отводится 2 байта):

/\*09SIZEOF.C

Программа на С, использующая операцию sizeof

для определения размера массива\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iDAYS\_OF\_WEEK 7

main()

{

int ivideo\_library[iDAYS\_OF\_WEEK]={1,2,3,4,5,6,7};

printf("There are %d number of bytes in the array"

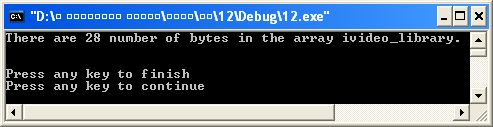
" ivideo\_library.\n",(int) sizeof(ivideo\_library));

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Данный вопрос поднимается в случаях, если программа должна быть переносимой и не зависеть от конкретных аппаратных средств. Вас может удивить использование преобразования в целое (int) результата операции sizeof(). Дело в том, что в стандарте ANSI С операция sizeof() возвращает не тип int, а тип size\_t, значения которого достаточно велики для хранения возвращаемого значения. В стандарте ANSI С эта поправка сделана потому, что на некоторых компьютерах значение целого числа недостаточно для представления объектов всех типов. В приведенном примере возвращаемое значение преобразуется в целое для того, чтобы оно соответствовало формату %d функции printf(). В противном случае, если возвращаемое значение будет больше целого, то функция printf() выполнится неправильно.

В следующем примере, изменяя тип данных массива iarray, можно определить, как хранятся данные различных типов:

/\*09ARRAY.C

Программа на С, иллюстрирующая непрерывность хранения массива\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iDAYS 7

main()

{

int index, iarray[iDAYS];

printf("sizeof(int) is %d\n\n",(int) sizeof(int));

for(index=0;index < iDAYS; index++)

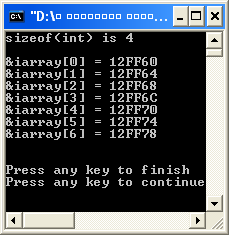
printf("&iarray[%d] = %X\n",index,&iarray[index]);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Если эта программа будет выполняться на компьютере с длиной слова, равной 2 байта, результат работы программы будет следующим:

sizeof(int) is 2

&iarray[0] = 2F32

&iarray[1] = 2F34

&iarray[2] = 2F36

&iarray[3] = 2F38

&iarray[4] = 2F3A

&iarray[5] = 2F3C

&iarray[6] = 2F3E

Обратите внимание на то, что операция определения адреса (&) может использоваться с любыми переменными, включая элементы массива. Элемент массива можно использовать так же, как и любую другую переменную: он может входить в выражения, может принимать значения или передаваться в качестве аргумента (или параметра) в функцию. Из приведенного примера видно, что адреса соседних элементов массива отличаются друг от друга точно на 2 байта. Вы поймете важность такого непрерывного хранения, когда будете использовать массивы вместе с переменными-указателями.

Следующий листинг на C++ представляет собой программу, эквивалентную только что рассмотренной:

// 09ARRAY.CPP

// Программа на C++, иллюстрирующая непрерывность хранения массива

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iMAX 7

main()

{

int index, iarray[iMAX];

cout << "sizeof(int) is " << (int) sizeof(int) << "\n\n";

for(index=0;index < iMAX; index++)

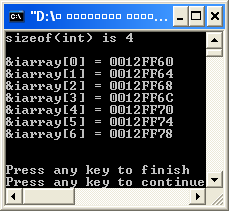
cout << "&iarray[" << index << "] = " << &iarray[index] <<endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



## Выход индекса за пределы массива.

Вам, наверное, знакомо выражение: "За все нужно платить". Это справедливо и для массивов в С. "Все" — это быстрое выполнение кода программы, а "плата" — отсутствие проверки границ массива. Напомним: поскольку язык С был разработан для замены ассемблера, в целях обеспечения компактности кода, ошибки такого рода компилятор не проверяет. Поскольку компилятор никак не определяет ошибочные ситуации, вы должны быть очень осторожны при работе с граничными значениями индекса массива. Например: компилятор транслирует следующую программу без каких-либо сообщений, однако, при записи в память за границами массива она может изменить значения других переменных или даже вызвать крах программы:

/\*09NORUN.C

**НЕ ЗАПУСКАЙТЕ эту программу на С**\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iMAX 10

#define iOUT\_OF\_RANGE 50

Int main()

{

int inot\_enough\_room[iMAX], index;

for(index=0; index < iOUT\_OF\_RANGE; index++)

inot\_enough\_room[index]=index;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

## Многомерные массивы.

Многомерный массив размерности N можно представить как одномерный массив из массивов размерности (N-1). Таким образом, например, трехмерный массив - это массив, каждый элемент которою представляет двумерную матрицу.

Примеры объявления многомерных массивов:

// Двумерный массив 6x9 элементов:

char Matrix2D[6][9];

// Трехмерный:

unsigned long Arr3D[4] [2] [8];

// Массив 7-й степени мерности:

my\_\_type Heaven[22][16][7][47][345][91][3];

Выражение Array [idx][idy], представляющее двумерный массив, переводится компилятором в эквивалентное выражение:

\*(\*(Array+idx)+idy)



**Рис.0702** Двумерный массив в памяти

Многомерные массивы инициализируются в порядке наискорейшего изменения самого правого индекса (задом наперед): сначала происходит присвоение начальных значений всем элементам последнего индекса, затем предыдущего и т.д. до самого начала:

int Mass[3][2][4] = {1, 2 3 4 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 22, 23, 24};



**Рис. 0703** Трёхмерный массив в памяти

В этом случае элементы массива будут получать следующие значения:

Mass[0][0][0] = 1;

Mass[0][0][1] = 2;

Mass[0][0][2] = 3;

…

Mass[0][1][0] = 5;

Mass[0][1][1] = 6;

…

Mass[1][0][0] = 9;

Mass[1][0][1] = 10;

…

Mass[l][1][0] = 13;

Mass[l][1][1] = 14;

…

Mass[2][0][0] = 17;

…

Mass[2][1][0] = 21;

…

Чтобы не запутаться, для наглядности можно группировать данные с помощью промежуточных фигурных скобок:

int Mass[3] [2] [4] = {{1,2, 3, 4},{5,6,7,8}}, {{9,10,11,12},{13,14,15,16}}, {{17,18,19,20},{21,22,23,24}};

Для многомерных массивов при инициализации разрешается опускать только величину первой размерности:

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL, "Rus");

char x[][3] = {{9, 8, 7}, {6, 5, 4}, {3, 2, 1}};

for(int i = 0; i<3; i++){

for(int j = 0; j < 3; j++)

cout << (int)x[i][j];

cout << '\n';

}

getchar(); getchar();

return 0;

}

В начале работы программы в памяти резервируется место для девяти однобайтных элементов (тип char) с заполнением массива в порядке следования байт инициализации:

x00=9; x01=8; x02=7; x10=6; x11=5; x12=4; x20=3; x21=2; x22=1.

Далее с помощью вложенного цикла осуществляется вывод значений массива: внешний цикл for перебирает строки исходной матрицы, в то время как внутренний цикл выводит значения массива по столбцам. В результате на экране будет отображено:

987

654

321



Доступ к элементам многомерного массива через указатели осуществляется немного сложнее. Поскольку, например, двумерный массив Matrix[x][у] может быть представлен как одномерный (Matrix[x]), каждый элемент которого также одномерным массивом (Matrix[у]), указатель на двумерный массив pMtrx, ссылаясь на элемент массива Matrix[x][y] по сути, указывает на массив Matrix [у] в массиве Matrix[x].

Таким образом, для доступа к содержимому ячейки указатель pMtrx придется разыменовывать дважды.

#include <iostream>

using namespace std;

int main ()

{

setlocale(LC\_ALL, "Rus");

char ArrayOfChar[3][2] ={'W', 'О', 'R', 'L', 'D', '!'};

char\* pArr = (char\*)ArrayOfChar;

pArr += 3;

char Letter = \*pArr;

cout << Letter;

getchar(); getchar();

return 0;

}

В приведенном примере объявляется массив символов размерностью 3 х 2 и указатель рАrг на него (фактически - указатель на ArrayOfChar [0] [0]). В строке

char\* pArr = (char\*)ArrayOfChar;

идентификатор ArrayOfChar уже является указателем на элемент с индексом 0, однако, поскольку массив двумерный, требуется его повторное разыменование.

Увеличение pArr на 3 приводит к тому, что он указывает на элемент массива, значение которого символ ' L' (элемент ArrayOfChar [1] [1]). Далее осуществляется вывод содержимого ячейки массива, на которую указывает pArr.



Понятие "размерность" относится к количеству индексов, используемых для обращения к отдельному элементу массива. Все используемые до сих пор массивы были одномерными, и для доступа к элементам требовался только один индекс. Глядя на объявление массива, можно определить его размерность. Если в объявлении только одна пара квадратных скобок([]), то массив — одномерный; если две пары ([][]), то двухмерный, и так далее. Массивы, имеющие более одного измерения, называются многомерными. Для моделирования пространства максимальное число измерений обычно равно 3.

В следующем объявлении описан двухмерный массив, который инициализируется при выполнении программы:

/\*092DARAY.C

Программа на С, иллюстрирующая использование двухмерного массива\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iROWS 4

#define iCOLUMNS 5

main()

{

int irow, icolumn, iadd, imultiple;

int istatus[iROWS][iCOLUMNS];

for(irow=0; irow < iROWS; irow++)

for(icolumn=0; icolumn < iCOLUMNS; icolumn++)

{

iadd=iCOLUMNS-icolumn;

imultiple=irow;

istatus[irow][icolumn]=(irow+1)\*icolumn+iadd\*imultiple;

}

for(irow=0; irow < iROWS; irow++)

{

printf("CURRENT ROW: %d\n",irow); /\* текущая строка \*/

printf("RELATIVE DISTANCE FROM BASE:\n"); /\*смещение относительно базы \*/

for(icolumn=0; icolumn < iCOLUMNS; icolumn++)

printf(" %d ",istatus[irow][icolumn]);

printf("\n\n");

}

printf ("\n\nPress any key to finish\n");

\_getch();

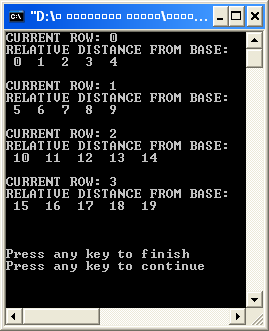
return(0);

}

В программе для вычисления смещения относительно первого элемента и инициализации каждого элемента массива используются два цикла for. Созданный массив имеет 4 строки (iROWS) и в каждой строке 5 колонок (iCOLUMNS), что в сумме дает 20 элементов. Многомерные массивы хранятся в памяти компьютера последовательно. Элементы многомерных массивов группируются в порядке индексов: от правого к левому. В предыдущем примере элемент в строке 1 и колонке 1 будет шестым элементом массива в памяти. Хотя вычисление смещения может показаться несколько запутанным, посмотрите, как легко адресуется каждый элемент массива:

istatus[irow][icolumn] = . . .

Результат выполнения программы выглядит следующим образом:



Многомерные массивы можно инициализировать теми же способами, что и одномерные. Например, в следующей программе описывается двухмерный массив dpowers, который инициализируется при описании. Функция pow() возвращает значение х, возведенное в степень у.

/\*092DADBL.C

Программа на С, использующая двухмерный массив типа double \*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iBASES 6

#define iEXPONENTS 3

#define iBASE 0

#define iRAISED\_TO 1

#define iRESULT 2

main()

{

double dpowers[iBASES][iEXPONENTS]={

1.1, 1, 0,

2.2, 2, 0,

3.3, 3, 0,

4.4, 4, 0,

5.5, 5, 0,

6.6, 6, 0

};

int irow\_index;

//int icolumn\_index;

for(irow\_index=0; irow\_index < iBASES; irow\_index++)

dpowers[irow\_index][iRESULT]=

pow(dpowers[irow\_index][iBASE],

dpowers[irow\_index][iRAISED\_TO]);

for(irow\_index=0; irow\_index < iBASES; irow\_index++)

{

printf(" %d\n",(int)dpowers[irow\_index][iRAISED\_TO]);

printf(" %2.1f = %.2f\n\n",dpowers[irow\_index][iBASE],

dpowers[irow\_index][iRESULT]);

}

printf ("\n\nPress any key to finish\n");

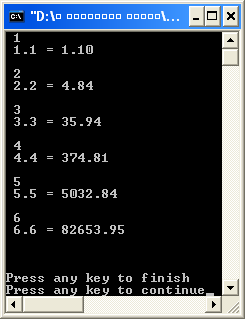
\_getch();

return(0);

}

Массив dpowers имеет тип double, поскольку функция pow() имеет два параметра типа double и возвращает результат типа double. Нужно, конечно, проявлять осторожность при инициализации двухмерных массивов; необходимо точно знать, какой индекс меняется быстрее — это всегда будет самый правый индекс.

Результат работы программы выглядит так:



Темпест Клауд, метеоролог, занимающаяся явлением перистости облаков, хочет проанализировать данные о ежемесячном количестве осадков на протяжении пяти лет. В самом начале она должна решить, как представлять данные. Можно использовать 60 переменных, по одной на каждый месяц. (Мы уже упоминали о таком подходе ранее, но в данном случае он также неудачен.) Лучше было бы взять массив, состоящий из 60 элементов, но это устроило бы нас только до тех пор, пока можно хранить раздельно данные за каждый год. Мы могли бы также использовать S массивов по 12 элементов каждый, но это очень примитивно и может создать действительно большие неудобства, если Темпест решит изучать данные о количестве осадков за 50 лет вместо пяти. Нужно придумать что-нибудь получше.

Хорошо было бы использовать массив массивов. Основной массив состоял бы тогда из 5 элементов, каждый из которых в свою очередь был бы массивом из 12 элементов. Вот как это записывается:

static float rain[5][12];

Можно также представить массив rain в виде двумерного массива состоящего из 5 строк и 12 столбцов.

При изменении второго индекса на единицу мы передвигаемся вдоль строки, а при изменении первого индекса на единицу, передвигаемся вертикально вдоль столбца. В нашем примере второй индекс дает нам месяцы, а первый — годы.

13_03

Используем этот двумерный массив в метеорологической программе. Цель нашей программы — найти общее количество осадков для каждого года, среднегодовое количество осадков и среднее количество осадков за каждый месяц. Для получения общего количества осадков за год следует сложить все данные, находящиеся в нужной строке. Чтобы найти среднее количество осадков за данный месяц, мы сначала складываем все данные в указанном столбце. Двумерный массив позволяет легко представить и выполнить эти действия.

/\* найти общее количество осадков для каждого года, средне\*/

/\* годовое, среднемесячное количество осадков, за несколько лет \*/

#define TWLV 12 /\* число месяцев в году \*/

#define YRS 5 /\* число лет \*/

main ()

{

static float rain [YRS][TWLV] = {

{10.2, 8.1, 6.8, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.3, 6.1, 7.4},

{9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0,0.6, 1.7, 4.3, 5.2},

{6.6, 5.5, 3.8, 2.8, 1.6, 0.2, 0.0, 0.0,0.0, 1.3, 2.6, 4.2},

{4.3, 4.3, 4.3, 3.0, 2.0, 1.0, 0.2, 0.2,0.4, 2.4, 3.5, 6.6},

{8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9,0.3, 0.9, 1.4, 7.2}

};

/\* инициализация данных по количеству осадков за 1970—1974 \*/

int year, month;

float subtot, total;

printf(" ГОД КОЛИЧЕСТВО ОСАДКОВ (дюймы)\n\n" );

for (year = 0, total = 0; year < YRS; year++ )

{

/\* для каждого года, суммируем количество осадков для каждого месяца \*/

for (month = 0, subtot = 0; month < TWLV; month++ )

subtot += rain [year] [month);

printf("%5d %15.1f\n", 1970 + year, subtot);

total + = subtot; /\* общее для всех лет \*/

}

printf(" \n среднегодовое количество осадков составляет %.1f дюймов.\n\n" , total/YRS );

printf(" Янв. Фев. Map. Апр.Май Июн. Июл. Авг.Сент.");

printf(" Окт. Нояб. Дек\n");

for (month = 0; month < TWLV; month++ )

{

/\* для каждого месяца, суммируем количество осадков за все годы \*/

for (year = 0, subtot = 0; year < YRS; year++ )

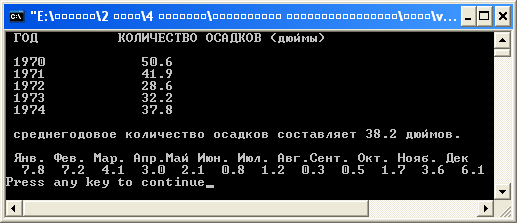
subtot += rain [year] [month];

printf(" %4.1f", subtot/YRS);

}

printf(" \n");

}



В этой программе следует отметить два основных момента: инициализацию и вычисления. Инициализация сложнее, поэтому мы сначала рассмотрим вычисления.

Чтобы найти общее количество осадков за год, мы не изменяем year, а заставляем переменную month пройти все свои значения. Так выполняется внутренний цикл for, находящийся в первой части программы. Затем мы повторяем процесс для следующего значения year. Это внешний цикл первой части программы. Структура вложенного цикла, подобная описанной, подходит для работы с двумерным массивом. Один цикл управляет одним индексом, а второй цикл — другим.

Вторая часть программы имеет такую же структуру, но теперь мы изменяем year во внутреннем цикле, a month во внешнем. Помните, что при однократном прохождении внешнего цикла внутренний цикл выполняется полностью. Таким образом, программа проходит в цикле через все годы, прежде чем изменится месяц, и дает нам общее количество осадков за пять лет для первого месяца, затем общее количество за пять лет для второго месяца и т. д.

### Инициализация двумерного массива.

Для инициализации массива мы взяли пять заключенных в скобки последовательностей чисел, а все эти данные еще раз заключили в скобки. Данные, находящиеся в первых внутренних скобках, присваиваются первой строке массива, данные во второй внутренней последовательности — второй строке и т. д. Правила, которые мы обсуждали раньше, о несоответствии между размером массива и данных применяются здесь для каждой строки. Если первая последовательность в скобках включает десять чисел, то только первым десяти элементам первой строки будут присвоены значения. Последние два элемента в этой строке будут, как обычно, инициализированы нулем по умолчанию. Если чисел больше, чем нужно, то это считается ошибкой; перехода к следующей строке не произойдет.

Мы могли бы опустить все внутренние скобки и оставить только две самые внешние. До тех пор пока мы будем давать правильное количество входных данных, результат будет тем же самым. Однако, если данных меньше, чем нужно, массив заполняется последовательно (не обращается внимание на разделение по строкам), пока не кончатся все данные. Затем оставшимся элементам будут присвоены нулевые значения. Смотри рисунок.

13_04

Все, что мы сказали о двумерных массивах, можно распространить и на трехмерные массивы и т. д. Трехмерный массив описывается следующим образом:

int solido[10][20][30];

Вы можете представить его в виде десяти двумерных массивов (каждый 20x30), поставленных друг на друга, или в виде массива из массивов. То есть это массив из 10 элементов, и каждый его элемент также является массивом. Все эти массивы в свою очередь имеют по 20 элементов, каждый из которых состоит из 30 элементов. Преимущество этого второго подхода состоит в том, что Можно довольно просто перейти к массивам большей размерности, если окажется, что вы не можете представить наглядно четырехмерный объект! Мы же останемся верны двум измерениям.

## Массивы в качестве аргументов функций.

Так же как и другие переменные С, массивы можно передавать от одной функции к другой. Поскольку использование массивов в качестве аргументов функций можно рассматривать в полной мере только после знакомства с указателями, здесь данная тема только обозначена.

В тело функций в качестве аргументов можно передавать значения, хранящиеся в массивах. При вызове функции, параметр типа массива преобразовывается компилятором в указатель на тип массива. Например, если аргумент-массив имеет тип unsigned long, при вызове он будет преобразован в unsigned long\*. Таким образом, изменение в функции значения любого элемента массива, являющегося аргументом, обязательно повлияет и на оригинал. Массивы отличаются от других типов тем, что их нельзя передавать по значению - внутрь тела функции попадает только адрес массива.

Синтаксис вызова функции при этом может быть следующим:

FunctionName (ArrayName);

Тогда прототип функции включает указание в качестве параметра типа передаваемого массива и следующих за ним прямоугольных скобок. Например:

FunctionName(char[]);

Другой вариант синтаксиса передачи массива в функцию - когда прототип функции содержит символ операции взятия адреса после указания типа аргумента:

char FunctionName(char&);

При этом синтаксис вызова функция принимает следующий вид:

FunctionName(\*ArrayName);

Ниже приводится пример, иллюстрирующий оба варианта передачи массива в качестве параметра функции.

#include <iostream>

using namespace std;

// Прототипы функций:

void Out1(int[]);

void Out2(int&);

int main()

{

setlocale(LC\_ALL, "Rus");

int Array[]={10, 8, 6, 4, 2, 0};

// Вызов первой функции:

Out1(Array);

cout << '\n';

// Вызов второй функции:

Out2(\*Array);

cout << '\n';

getchar(); getchar();

return 0;

}

// Реализация обеих функций:

void Out1(int arr[])

{

for(int i=0; I < sizeof(arr); i++)

cout << arr[i] << " ";

}

void Out2 (int& arr)

{

for(int i=0; i<sizeof(arr); i++)

cout<<\*(&(arr)+i) << " ";

}

В рассмотренном примере объявляется массив Array [], содержащий шесть целочисленных значений, и осуществляется его передача в функции Out1 ()и Out (2). Обе функции выполняют одно и то же действие - выводят содержимое массива-аргумента на печать и отличаются только интерфейсом.



Использование в качестве параметра функции многомерного массива затруднено, поэтому на практике чаще всего осуществляется передача массива указателей, что значительно упрощает синтаксис.

### Передача массивов функциям С.

Рассмотрим функцию isum(), которая вычисляет сумму элементов массива inumeric\_values[0], inumeric\_values[1], ..., inumeric\_values[n]. Необходимы два параметра: параметр массива, названный iarray\_address\_received, для передачи копии адреса массива и параметр, названный imax\_size, для указания индекса последнего элемента массива, который нужно добавлять. Предположив, что массив содержит целые числа, и что индекс также имеет тип int, параметры функции isum() можно описать следующим образом:

int isum(int iarray\_address\_received[], int imax\_size)

Квадратные скобки после имени массива, включенные в объявление параметров, указывают функции isum(), что iarray\_address\_received — это имя массива, а не имя обычного параметра. Обратите внимание на то, что размер массива не указан в квадратных скобках. Простой параметр imax\_size объявляется так же, как раньше. Для вызова функции достаточно указать:

isum(inumeric\_values,iactual\_index);

Передать массив inumeric\_values — это просто указать его имя в качестве аргумента. При передаче имени массива в функцию на самом деле передается адрес первого элемента массива. Взгляните на следующее выражение:

inumeric\_values

на самом деле оно представляет собой краткую запись выражения

&inumeric\_values[0]

На практике функцию isum() можно вызывать при помощи любого из следующих допустимых операторов:

isum(inumeric\_values,iactual\_index) ;

itotal = isum(&inumeric\_values[0],iactual\_index);

В любом случае внутри функции isum() имеется доступ к каждому элементу массива.

Когда функции требуется обработать какой-нибудь массив, вызывающая функция включает имя этого массива в список параметров функции. Это означает, что вызываемая функция получает и обрабатывает массив, обращаясь к его реальным элементам, а не к локальной копии, как это происходит в случае одиночных переменных, вызываемых по значению. По умолчанию в С все массивы передаются по имени или по ссылке. Это позволяет избежать частых сообщений об ошибке "stack overruns heap" (наложение стека на динамически выделяемую память), с которыми часто сталкиваются программисты на Паскале, если они забывают включить для массива модификатор var при объявлении списка формальных аргументов. В отличие от С, в языке Паскаль все аргументы типа массив передаются по значению. При вызове по значению компилятор дублирует содержимое всего массива; для больших массивов это приводит к потерям времени и памяти.

Если функция должна получить имя массива в качестве аргумента, существует два способа для локального объявления этого аргумента: объявление массива или объявление указателя. Какой способ выбрать — зависит от того, как функция обрабатывает набор значений. Если для доступа к элементам массива функция использует индекс, то нужно объявлять массив, а вслед за его именем ставить квадратные скобки. Размер массива может быть пустым, так как объявление резервирует место не для всего массива, а только для его начального адреса. Если в начале функции встретилось объявление массива, то затем компилятор разрешает использовать в теле функции имя массива с индексом, заключенным в квадратные скобки.

В следующей программе объявляется массив из пяти элементов, и после печати их значений вызывается функция, определяющая наименьшее значение в массиве. Для этого имя массива и его размер передаются в функцию iminimum(), в которой они объявлены как массив iarray[] и как целая переменная imax. Функция просматривает массив, сравнивая каждый элемент массива с найденным на предыдущих этапах минимальным значением, и всякий раз, когда находит минимум, запоминает новое значение в переменной ikurrent\_minimum. В конце концов, найденное минимальное значение возвращается функции main(), где оно выводится на печать.

/\*09PASARY.C

Программа на С, использующая массивы в качестве параметров функции\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iMAX 10

#define iUPPER\_LIMIT 100

main()

{

int iarray[iMAX]={3,7,2,1,5,6,8,9,0,4};

int i, ismallest;

int iminimum(int iarray[],int imax);

printf("The original list looks like: "); /\* Исходный список \*/

for(i=0; i < iMAX; i++)

printf("%d ",iarray[i]);

ismallest=iminimum(iarray, iMAX);

printf("\nThe smallest value is: %d: \n",ismallest); /\* Минимум \*/

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

int iminimum(int iarray[], int imax)

{

int i, icurrent\_minimum;

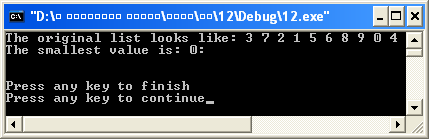
icurrent\_minimum=iUPPER\_LIMIT;

for(i=0; i < imax; i++)

if(iarray[i] < icurrent\_minimum) icurrent\_minimum=iarray[i];

return(icurrent\_minimum);

}



Предположим, что вы хотите иметь функцию, работающую с двумерным массивом, причем со всем целиком, а не с частями. Как вы запишите определения функции и ее описания? Подойдем к этому более конкретно и скажем, что нам нужна функция, управляющая массивом junk[][] в нашем последнем примере. Пусть функция main () выглядит так:

/\* junk в main \*/

main ()

{

static int junk[3][4] = {

{2, 4, 5, 8},

{100, 200, 300, 400},

{10, 40, 60, 90}

};

stuff(junk);

}

Функция stuff () использует в качестве аргумента junk, являющийся указателем на весь массив. Как написать заголовок функции, не зная, что делает stuff ()?

Попробуем написать:

stuff (junk)

int junk[];?

или

stuff(junk)

int junk[][];?

Нет и нет. Первые два оператора еще будут работать некоторым образом, но они рассматривают junk как одномерный массив, состоящий из 12 элементов. Информация о расчленении массива на строки отсутствует.

Вторая попытка ошибочна, потому что хотя оператор и указывает, что junk является двумерным массивом, но нигде не говорится, из чего он состоит. Из шести строк и двух столбцов? Из двух строк и шести столбцов? Или из чего-нибудь еще? Компилятору недостаточно этой информации. Ее дают следующие операторы:

stuff(junk)

int junk[][4];

Они сообщают компилятору, что массив следует разбить на строки по четыре столбца.

Массивы символьных строк являются особым случаем, так как у них нулевой символ в каждой строке сообщает компилятору о конце строки. Это разрешает описания, подобные следующему:

char \*list[];

Символьные строки представляют одно из наиболее частых применений массивов и указателей; мы вернемся к этой теме позднее.

### Передача массивов функциям C++.

Следующая программа напоминает те программы на С, которые рассматривались выше; в ней показано, как объявить и передать массив в качестве аргумента функции.

// 09FNCARY.CPP

// Программа на C++ иллюстрирует использование массивов в функциях

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iSIZE 5

void vadd\_1(int iarray[]);

main()

{

int i, iarray[iSIZE]={0,1,2,3,4};

cout << "iarray before calling add\_1:\n\n"; //массив перед вызовом функции

for(i=0;i < iSIZE; i++)

cout << " " << iarray[i];

vadd\_1(iarray);

cout << "\n\niarray after calling add\_1:\n\n"; // массив после вызова функции

for(i=0;i < iSIZE; i++)

cout << " " << iarray[i];

cout << endl;

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vadd\_1(int iarray[])

{

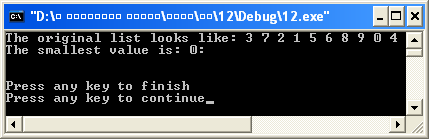
int i;

for(i=0;i < iSIZE; i++)

iarray[i]++;

}

Результат работы программы выглядит следующим образом:



Что можно сказать о массиве, выступающем в качестве параметра, глядя на результат работы программы? Массив передавался по значению или по ссылке? Функция vadd\_1() просто добавляет единицу к каждому элементу массива. Поскольку эти инкрементированные значения отображаются в массиве iarray в функции main(), оказывается очевидным то, что параметр передавался по ссылке. Действительно, это согласуется с изложенными выше рассуждениями о сути имени массива. Запомните: имя массива — это адрес его первого элемента.

В следующей программе на C++ объединены многие описанные выше средства работы с массивами: в том числе инициализация многомерного массива, обращение к его элементам и передача в качестве параметра.

// 092DARAY.CPP

// Программа на C++ иллюстрирует описание многомерного массива,

// обращение к его элементам и передачу их в качестве параметров

// функции

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

void vdisplay\_results(char carray[][3][4]);

char cglobal\_cube[5][4][5]={

{

{'P','L','A','N','E'},

{'Z','E','R','O',' '},

{' ',' ',' ',' ',' '},

{'R','O','W',' ','3'},

},

{

{'P','L','A','N','E'},

{'O','N','E',' ',' '},

{' ',' ',' ',' ',' '},

{'R','O','W',' ','2'},

},

{

{'P','L','A','N','E'},

{'T','W','O',' ',' '},

},

{

{'P','L','A','N','E'},

{'T','H','R','E','E'},

{'R','O','W',' ','2'},

{'R','O','W',' ','3'},

},

{

{'P','L','A','N','E'},

{'F','O','U','R',' '},

{'r','o','w',' ','2'},

{'a','b','c','d','e'},

}

};

int imatrix[4][3]={{1},{2},{3},{4}};

main()

{

//int iplane\_index;

int irow\_index;

int icolumn\_index;

char clocal\_cube[2][3][4];

cout << "size clocal\_cube = " << sizeof(clocal\_cube) << "\n";

cout << "size clocal\_cube[0] = " << sizeof(clocal\_cube[0]) << "\n";

cout << "size clocal\_cube[0][0] = " << sizeof(clocal\_cube[0][0]) << "\n";

cout << "size clocal\_cube[0][0][0] = " << sizeof(clocal\_cube[0][0][0]) << "\n";

vdisplay\_results(clocal\_cube);

cout << "cglobal\_cube[0][1][2] is = " << cglobal\_cube[0][1][2] << "\n";

cout << "cglobal\_cube[1][0][2] is = " << cglobal\_cube[1][0][2] << "\n";

// печать плоскости 0 куба cglobal\_cube

cout << "\nprint part of the cglobal\_cube's plane 0\n";

for(irow\_index=0; irow\_index < 4; irow\_index++)

{

for(icolumn\_index=0; icolumn\_index < 5; icolumn\_index++)

cout << cglobal\_cube[0][irow\_index][icolumn\_index];

cout << "\n";

}

// печать плоскости 4 куба cglobal\_cube

cout << "\nprint part of the cglobal\_cube's plane 4\n";

for(irow\_index=0; irow\_index < 4; irow\_index++)

{

for(icolumn\_index=0; icolumn\_index < 5; icolumn\_index++)

cout << cglobal\_cube[4][irow\_index][icolumn\_index];

cout << "\n";

}

// печать всех значений imatrix

cout << "\nprint all of imatrix\n";

for(irow\_index=0; irow\_index < 4; irow\_index++)

{

for(icolumn\_index=0; icolumn\_index < 3; icolumn\_index++)

cout << imatrix[irow\_index][icolumn\_index];

cout << "\n";

}

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

void vdisplay\_results(char carray[][3][4])

{

cout << "sizeof carray= " << sizeof(carray) << "\n";

cout << "sizeof carray[0]= " << sizeof(carray[0]) << "\n";

cout << "sizeof cglobal\_cube= " << sizeof(cglobal\_cube) << "\n";

cout << "sizeof cglobal\_cube[0]= " << sizeof(cglobal\_cube[0]) << "\n";

}

Во-первых, обратите внимание на то, как описывается и инициализируется cglobal\_cube. Для группировки символов используются фигурные скобки; это внешне напоминает размерности массива, что помогает представить массив визуально. В данном случае фигурные скобки необязательны, так как в данных, инициализирующих массив, отсутствуют пропуски. Если бы инициализировалась только часть какого-либо измерения, то нужны были бы различные наборы внутренних фигурных скобок для того, чтобы определить, какие начальные значения использовать для соответствующей части массива. Проще всего представить трехмерный массив визуально, если изобразить его в виде пяти слоев (плоскостей), каждый из которых образует двухмерный массив из четырех строк и пяти столбцов (см. рис. 9.2). Первые четыре строки программы выводят на экран размер массива clocal\_cube, размеры образующих его массивов и размеры отдельного элемента. Результирующее значение показывает, что общий размер многомерного массива является произведением размеров всех его измерений и размера типа данных массива: 2\*3\*4\* sizeof (char), то есть 24.

09_02

Заметьте, что элемент массива clocal\_cube[0] сам является двухмерным массивом [3][4]; поэтому размер clocal\_cube[] равен 12. Размер элемента clocal\_cube[0][0] равен 4, что составляет число элементов в последнем измерении, так как каждый элемент имеет размер 1 в соответствии с результатом операции sizeof(clocal\_cube[0][0][0]).

Для того чтобы полностью понять организацию многомерных массивов, очень важно помнить о том, что clocal\_cube[0] — это и имя массива, и константа-указатель. Поскольку в программе отсутствует индекс последнего измерения, тип этого выражения не совпадает с типом данных каждого базового элемента массива. Так как выражение clocal\_cube[0] ссылается не на отдельный элемент, а на другой массив, оно не имеет тип char. Поскольку clocal\_cube[0] имеет тип постоянного указателя, оно не является допустимым lvalue и не может появляться в левой части операции присваивания.

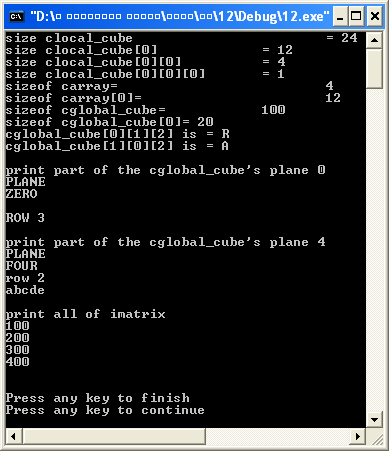
Нечто очень интересное происходит, когда имя массива используется в списке параметров функции, как это было при вызове функции vdisplay\_results() с параметром clocal\_cube. Если, находясь в теле функции, выполнить операцию sizeof() с формальным параметром, представляющим собой имя массива, то вычисление действительного размера массива саrrау осуществляется неправильно. Функция видит только копию адреса первого элемента массива, поэтому sizeof() возвращает размер адреса, а не размер того объекта, на который он ссылается.

В функции vdisplay\_results() результат операции sizeof() с параметром саrrау[0] равен 12, потому что формальный параметр объявлен в функции как массив с двумя измерениями [3] и [4]. При объявлении размера последних двух измерений нельзя использовать произвольные значения, так как в прототипе функции они описаны как [3] и [4]. Если бы не было прототипа, то компилятор не смог бы определить способ задания размерностей массива. Это позволило бы переопределить организацию массива. Функция vdisplay\_results() выдает также размер глобальной переменной cglobal\_cube. Это говорит о том, что, хотя функция может иметь непосредственный доступ к глобальным данным, она обращается только к адресу массива, который передан ей в качестве аргумента.

Два следующих оператора, выполняемых в функции main(), иллюстрируют обращение к отдельным элементам массива cglobal\_cube. Выражение clocal\_cube[0][1][2] ссылается на элемент "R", находящийся на нулевой плоскости во второй строке третьего столбца. Выражение clocal\_cube[1][0][2] адресуется к элементу на второй плоскости в нулевой строке третьего столбца или "А".

Следующий фрагмент кода в main() образован двумя вложенными циклами for, которые призваны проиллюстрировать тот факт, что массивы хранятся в порядке "плоскость – строка - столбец". Как мы уже видели, при последовательном просмотре массива сначала изменяется самый правый индекс массива (столбец). В первой паре циклов for задается вывод нулевой плоскости и определяется строка, при этом внутренний цикл пробегает все колонки в cglobal\_cube. Затем эта структура циклов повторяется, однако печатается только пятая плоскость (plane [4]) куба cglobal\_cube. Последняя пара циклов for используется для вывода элементов массива imatrix в виде прямоугольника, аналогично тому, как многие люди представляют двухмерный массив.

Результат работы программы следующий:



Вас не смущает такой результат? Взгляните на описание, где инициализируется массив imatrix. Поскольку каждый внутренний набор фигурных скобок соответствует одной строке массива и внутри этих скобок недостаточно начальных значений, остальные элементы массива система заполнила нулями. Запомните, что в С все неопределенные статические автоматические числовые массивы получают по умолчанию нулевые начальные значения.

## Ввод и вывод строк.

Хотя в С и имеется тип данных char, в нем отсутствует тип данных для символьных строк. Вместо этого строка должна быть представлена в виде массива символов. Каждый символ строки хранится в отдельной ячейке массива, а в последней ячейке содержится null-символ \0.

В следующей программе показаны три основных способа представления и обработки символьной строки. Массив szmode1 инициализируется посимвольно при помощи операции присваивания, массив szmode2 инициализируется при помощи функции scanf(), а массив szmode3 инициализируется при описании.

// 09STRING.CPP

// В этой программе на C++ показано использование строк

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

main()

{

char szmode1[4], //car

szmode2[6]; //plane

static char szmode3[5]="ship"; //ship

szmode1[0]='c';

szmode1[1]='a';

szmode1[2]='r';

szmode1[3]='\0';

printf("\n\n\tPliase enter the mode --> plane ");

scanf("%s",szmode2);

printf("%s\n",szmode1);

printf("%s\n",szmode2);

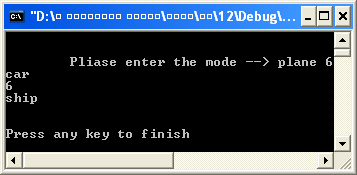
printf("%s\n",szmode3);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Хотя строка в массиве szmode1, "car", имеет три символа, сам массив szmode1 имеет четыре элемента — по одному на символ и еще один для null-символа. Запомните: \0 считается одним символом. Аналогичным образом, строка "plane" содержит пять символов (в строке "ship" — четыре), но требует для хранения шесть ячеек (пять для szmode3), включая null-символ. Напоминаем, что массив символов szmode3[5] можно также инициализировать, используя для этого вариант записи с фигурными скобками:

static char szmode3[5] = {'s','h','i','p','\0'};

Когда при перечислении начальных значений символов массива используются двойные кавычки, компилятор автоматически добавляет null-символ окончания строки \0. Кроме того, эта строка может быть записана следующим образом:

static char szmode3[] = "ship";

Здесь используется безразмерный массив. Конечно, для инициализации символьного массива можно выбрать и более длинный способ, такой как для массива szmode1. Чаще для считывания строки непосредственно в массив используется функция scanf(), как это было сделано с массивом szmode2. В функции scanf() используется спецификация преобразования %s. В этом случае функция пропускает разделительные символы (символы пробела, табуляции и возврата каретки) и считывает в символьный массив szmode2 все символы до следующего разделительного. Затем автоматически добавляется null-символ. Размер массива должен быть указан достаточно большим для размещения всей строки с null-символом. Взгляните еще раз на этот оператор:

scanf("%s",szmode2);

Вас не беспокоит тот факт, что перед szmode2 не стоит операция определения адреса &? Уже говорилось, что функция scanf() получает адрес переменной; однако имя массива, в отличие от имен переменных, и является адресом первого элемента массива.

Когда функция printf() используется со спецификацией %s, ожидается, что соответствующий аргумент будет представлять собой адрес некоторой символьной строки. Эта строка и печатается, но за исключением null-символа. Все эти соображения проиллюстрированы в аналогичной программе на C++:

// 09STRING.CPP

// В этой программе на C++ показано использование строк

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

main()

{

char szmode1[4], //car

szmode2[6]; //plane

static char szmode3[5]="ship"; //ship

szmode1[0]='c';

szmode1[1]='a';

szmode1[2]='r';

szmode1[3]='\0';

printf("\n\n\tPliase enter the mode --> plane ");

scanf("%s",szmode2);

printf("%s\n",szmode1);

printf("%s\n",szmode2);

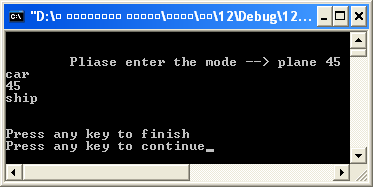
printf("%s\n",szmode3);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Результат работы программы выглядит следующим образом:

саг

plane

ship

## Строковые функции и символьные массивы.

Многие функции, использующие в качестве аргументов символьные массивы, до сих пор не рассматривались — это объясняется особенностями представления в С строковых данных. К этим функциям относятся следующие: gets(), puts(), fgets(), fputs(), sprintf(), stpcpy(), strcat(), strncmp(), and strlen(). Теперь, когда вы знакомы с концепцией символьных массивов и строк, оканчивающихся null-символом, объяснить принципы работы этих функций гораздо легче. Для этого проще всего рассмотреть несколько программных примеров.

### Функции gets(), puts(), fgets(), fputs() и sprintf().

Следующий пример показывает использование функций gets(), puts(), fgets(), fputs() и sprintf() для форматированного ввода/вывода: Последняя пара циклов for используется для вывода элементов массива imatrix в виде прямоугольника, аналогично тому как многие люди представляют двухмерный массив.

/\*09STRIO.C

Программа на С, использующая некоторые функции ввода/вывода строк\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iSIZE 20

main()

{

char sztest\_array[iSIZE];

/\* Введите, пожалуйста, первую строку \*/

fputs("Please enter the first string:", stdout);

gets(sztest\_array);

/\* Первая введенная строка \*/

fputs("The first string entered is:",stdout);

puts(sztest\_array);

fputs("Please enter the second string:", stdout);

fgets(sztest\_array,iSIZE,stdin);

fputs(sztest\_array,stdout);

/\* Это просто тест \*/

sprintf(sztest\_array,"This was %s a test","just");

/\* Функция sprintf() создана\*/

fputs("sprintf() created: ",stdout);

fputs(sztest\_array,stdout);

fputs("\n",stdout);

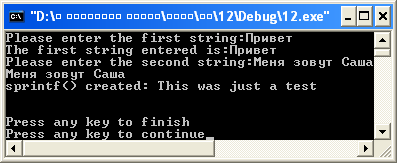
printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}

Результат первого прогона программы следующий:



Поскольку длины введенных строк не превышают размер массива sztest\_array, программа работает нормально. Однако, если при втором прогоне программы ввести строку длиннее, чем массив sztest\_array, может получиться нечто следующее:

Please enter the first string : one two three four five

The first string entered is : one two three four five

Please enter the second string : six seven eight nine ten

The second string entered is : six seven eight ninsprintf() created

: This was just a testPlease enter the first string : The first string

entered is :e ten

The second string entered is :

Будьте осторожны при запуске программы. Функция gets() получает символы от стандартного устройства ввода (stdin, по умолчанию для большинства компьютеров это клавиатура) и помещает их в массив, имя которого задано в качестве параметра. Когда для завершения строки вы нажимаете клавишу <ENTER>, передается символ перевода строки. Когда функция gets() получает этот символ перевода строки, она заменяет его на null-символ, это гарантирует, что в символьном массиве оказывается не просто набор значений, а именно строка. Никаких проверок размера массива и его соответствия числу введенных символов не производится. Функция puts() отображает на экране то, что было введено при помощи функции gets(). Кроме того, в конце строки вместо null-символа она добавляет символ перевода строки. Напоминаем, что null-символ автоматически вставляет в строку функция gets(). Следовательно, строки, введенные без ошибок при помощи функции gets(), можно вывести на экран, используя puts().

Если вы пользуетесь функцией fgets(), то можете задавать максимальное число вводимых символов. Функция прекращает считывание из заданного файлового потока в тот момент, когда число считанных символов на единицу меньше, чем значение второго заданного аргумента. Поскольку размер массива sztest\_array равен 20, функция fgets() считывает из stdin только 19 символов. Последнюю позицию в строке автоматически занимает null-символ; если же с клавиатуры вводился символ перевода строки, то он в строке сохранится. (В режиме отладки его можно увидеть перед null-символом.) Функция fgets() не удаляет символы перевода строки так, как это делает функция gets(); она просто добавляет null-символ в конце, и, следовательно, обеспечивает сохранение допустимой строки. Функции fgets() и fputs() являются в каком-то смысле симметричными, так же как и функции gets() и puts(). Функция fgets() не удаляет символы перевода строки, а функция fputs() не добавляет новые.

Чтобы понять важность символа перевода строки для этих функций, взгляните внимательно на результат второго прогона программы. Обратите внимание на фразу "sprintf() created..."; она следует сразу же за введенными числами "six seven eight nine". Во второй введенной строке на самом деле было на пять символов больше, чем прочитала функция fgets() (на один меньше, чем iSIZE, то есть 19). Остальные символы остались во входном буфере. Также пропущенным оказался символ перевода строки, завершающий ввод с клавиатуры. (Он остался во входном потоке, поскольку оказался по счету следующим после девятнадцатого символа.) Поэтому символ перевода строки в ней не запомнился. Поскольку функция fputs() не добавляет этих символов, следующий вывод начинается в той же строке, где закончился предыдущий. Расчет делался на прочитанный функцией fgets() и выведенный при помощи fputs() символ перевода строки, который должен был обеспечить правильность вывода на экран. Функция sprintf() используется как "строковая printf()". В ней также, как и в printf(), используется управляющая строка с символами преобразования. Особенностью sprintf() является то, что она помещает форматированный результат в строку, а не выдает его сразу же на стандартное устройство вывода. Это может быть удобно в том случае, когда один и тот же результат нужно вывести дважды, например, когда одну и ту же строку необходимо вывести и на дисплей, и на принтер.

Подведем черту:

* Функция gets() преобразует символ перевода строки в null-символ.
* Функция puts() преобразует null-символ в символ перевода строки.
* Функция fgets() оставляет символы перевода строки и добавляет null-символ.
* Функция fputs() опускает null-символ и не добавляет символы перевода строки; она использует сохраненные символы перевода строки (если они вводились).

### Функции strcpy(), strcat(), strncmp() и strlen().

Все функции, обсуждаемые в данном разделе, предопределены в заголовочном файле string.h. Всякий раз, когда вы хотите использовать какую-либо из этих функций, не забудьте включить заголовочный файл в свою программу. Помните и о том, что все строковые функции, имеющие прототипы в файле string.h, получают строковые параметры, оканчивающиеся null-символом. В следующей программе показано, как используется функция strcpy():

/\*09STRCPY.C

Программа на С, использующая функцию strcpyO\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iSIZE 20

main()

{

char szsource\_string[iSIZE]="Initialized String!",

szdestination\_string[iSIZE];

strcpy(szdestination\_string,"String Constant");

cout << "\n" << szdestination\_string;

strcpy(szdestination\_string,szsource\_string);

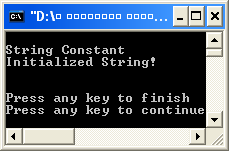
cout << "\n" << szdestination\_string << "\n";

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Функция strcpy() копирует содержимое первой строки (szsource\_string) во вторую строку (szdestination\_string). В приведенной программе переменная szsource\_string при инициализации получает значение "Initialized String!" ("Инициализированная строка"). При первом вызове функции strcpy() строка "String Constant" ("Строковая константа") непосредственно копируется в переменную szdestination\_string, а при втором вызове strcpy() строка szsource\_string копируется в переменную szdestination\_string. Программа выдает следующие сообщения:

String Constant

Initialized String!

Эквивалентная программа на C++ выглядит следующим образом:

// 09STRCPY.CPP

// Программа на C++, использующая функцию strcpy()

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iSIZE 20

main()

{

char szsource\_string[iSIZE]="Initialized String!",

szdestination\_string[iSIZE];

strcpy(szdestination\_string,"String Constant");

printf("%s\n",szdestination\_string);

strcpy(szdestination\_string,szsource\_string);

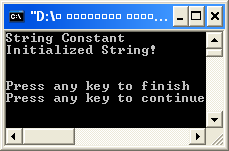
printf("%s\n",szdestination\_string);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Функция strcat() объединяет вместе две отдельные строки, которые должны заканчиваться null-символом; результат также имеет в конце null-символ. В следующей программе используется уже знакомая вам функция strcpy() и представляется функция strcat():

/\*09STRCAT.C

Программа на С, иллюстрирующая использование функции strcat()\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

#define iSTRING\_SIZE 35

main()

{

char szgreeting[]="Good morning!", /\* Доброе утро \*/

szname[]=" Carolyn, ", /\* Кэролин, \*/

szmessage[iSTRING\_SIZE];

strcpy(szmessage,szgreeting);

strcat(szmessage,szname);

strcat(szmessage,"how are you?"); /\* как поживаете? \*/

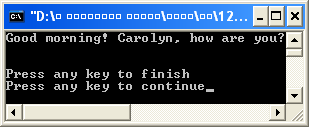
printf("%s\n",szmessage);

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



В этом примере две переменные, szgreeting и szname, инициализируются, а переменная szmessage — нет. При помощи функции strcpy() содержимое переменной szgreeting копируется в szmessage. Затем функция strcat() позволяет соединить содержимое переменной szname ("Carolyn,") и строку "Good morning", хранящуюся в переменной szmessage. Последний вызов функции strcat() осуществляет конкатенацию строковой константы и строки: литеральная строка "how are you?" объединяется с текущим содержимым переменной szmessage ("Good morning Carolyn,"). В результате на выходе программы получается следующее:

Good morning Carolyn, how are you?

В следующей программе для проверки на равенство (совпадение) двух строк используется функция strncmp():

/\*09SRNCMP.C

Программа на С, в которой для сравнения двух строк используется

функция strncmp() с помощью функции strlen()\*/

#include "stdafx.h"

#include <iostream>

#include <conio.h>

#include <stdio.h>

#include <process.h>

#include <ctype.h>

#include <stdlib.h>

#include <math.h>

#include <stdarg.h>

#include <string.h>

using namespace std;

main()

{

char szstringA[]="Adam", szstringB[]="Abel";

int istringA\_length, iresult=0;

istringA\_length=strlen(szstringA);

if(strlen(szstringB) >= strlen(szstringA))

iresult=strncmp(szstringA, szstringB, istringA\_length);

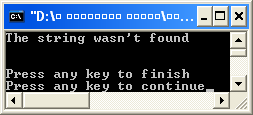
printf("The string %s found\n",iresult ==0 ? "was" : "wasn't");

printf ("\n\nPress any key to finish\n");

\_getch();

return(0);

}



Функция strlen() весьма полезна: она возвращает число символов в заданной строке, не считая null-символа. Эта функция использована в приведенной программе в двух различных ситуациях; это сделало для того, чтобы пояснить способы ее применения. При первом вызове функции длина строки szstringA запоминается в переменной istringA\_length. Второе обращение к функции происходит при обработке if-условия. Напоминаем, что при вычислении условий проверки получается результат ИСТИНА (не ноль или !0) и ЛОЖЬ (0). При вычислении if-условия берутся результаты двух обращений к strlen() и выполняется условная операция >=. Если длина строки szstringB >= длины строки szstringA, то вызывается функция strncmp().

Вас, вероятно, заинтересует, почему в программе используется проверка >=, а не ==. Для ответа на этот вопрос нужно еще раз пояснить принцип работы функции strncmp(). Эта функция сравнивает две строки, начиная с первого символа каждой строки. Если строки идентичны, то функция возвращает ноль. Если же строки не совпадают, strncmp() возвращает значение меньшее нуля, если строка szstringA меньше строки szstringB; или значение больше нуля, если szstringA больше szstringB. Проверочное условие >= используется в том случае, когда в программе нужно учесть случай равенства сравниваемых строк или определить большую или меньшую строку. В последнем операторе программы используется полученное значение iresult и условная операция (?:); с их помощью определяется, какое из строковых сообщений печатать. Для приведенного примера результат программы будет следующий:

The string wasn't found

(Строка не найдена)

Перед тем, как перейти к следующему материалу, напомним, что из числа наиболее часто встречающихся программных ошибок, две связаны с выходом за границы массивов и с отсутствием в конце символьных массивов, используемых в качестве строк, null-символа \0. Обе ошибки могут очень долго не проявляться, пока какой-нибудь пользователь не введет строку, размер которой окажется на символ длиннее, чем нужно.

## Динамическое выделение памяти.

В программе каждая переменная может размещаться в одном из трех мест: в области данных программы, в стеке или свободной памяти (так называемая куча).

Каждой переменной в программе память может отводится либо статически, то есть в момент загрузки программы, либо динамически – в процессе выполнения программы. До сих пор все определяемые массивы объявлялись статически, и, следовательно, хранили значения всех своих элементов в стековой памяти или области данных программы. Если количество элементов массива невелико, такое размещение оправдано. Однако довольно часто возникают случаи, когда в стековой памяти, содержащей локальные переменные и вспомогательную информацию (например, точки возврата из вложенных функций), недостаточно места для размещения всех элементов большого массива. Ситуация еще более усугубляется, если массивов большого размера должно быть много. Здесь на помощь приходит возможность использования для хранения данных динамической памяти.

### Функции malloc, calloc, free и операторы new и delete

Чтобы в дальнейшем можно было разместить в памяти некоторый динамический объект, для него необходимо предварительно выделить в памяти соответствующее место. По окончании работы с объектом выделенную для его хранения память требуется освободить.

Выделение динамической памяти под объект осуществляется при помощи следующих выражений:

* malloc;
* calloc;
* new;

Освобождение выделенных ресурсов памяти производится выражениями:

* free;
* delete;

Функция malloc подключается в одном из заголовочных файлов stdlib.h или alloc.h и имеет синтаксис:

void \*malloc (size\_t size);

Данная функции призвана выделить в памяти блок размером size байт из кучи. В больших моделях памяти (Compact, Large, Huge) для кучи доступно все пространство памяти от вершины программного стека. В моделях Tiny, Small и Medium это пространство сокращено из-за наличия небольшой системной облости, позволяющей стеку расти.

В случае успешного резервирования блока памяти функция malloc возвращает указатель на только что выделенный блок.

При неудачном результате операции с памятью функция возвращает NULL - зарезервированный идентификатор со значением ‘\0’(абсолютное значение 0x00) При этом содержимое остается неизменным. Если в качестве аргумента функции используется значение 0, функция возвращает NULL.

Как видно из синтаксиса, данная функция возвращает указатель типа void, однако, поскольку на практике чаще всего приходится выделять память для объектов конкретного типа, приходится приводить тип полученного пустого указателя к требуемому типу. Например, выделение памяти под три объекта типа int и определение указателя на начало выделенного блока можно произвести следующим образом:

int pint = (int\*)malloc(3\*sizeof (int));

В отличие от malloc. функция calloc кроме выделения области памяти под массив объектов еще производит инициализацию элементов массива нулевыми значениями. Функция имеет следующий синтаксис:

void \*calloc(size\_t num, size\_t size);

Аргумент num указывает, сколько элементов будет храниться в массиве, а параметр size сообщает размер каждого элемента в байтах Приведенный здесь тип size\_t (объявлен в заголовочном файле stddef.h) является синонимом типа unsigned char, возвращаемого оператором sizeof.

Функция освобождения памяти free в качестве единственного аргумента принимает указатель на удаляемый блок объектов (\*block) и имеет синтаксис:

void free(void \*block);

Данная функция подключается в заголовочном файле stdlib.h или alloc.h. Тип объектов удаляемого блока может быть произвольным, на что указывает ключевое слово void.

Рассмотрим пример выделения динамической памяти для десяти объектов типа int. заполнение выделенной области числами от нуля до девяти и освобождение ресурса памяти.

#include <iostream.h>

#include <alloc.h>

int main(void)

{

int\* pRegion;

// Выделение области памяти

if((pRegion =(int\*)malloc(10\*sizeof(int)))=NULL)

{

cout<< "He достаточно памяти\n";

// Завершить программу если не

// хватает памяти для выделения

return 1;

}

// Заполнение выделенного блока памяти

for(int i=0; i<10; i++)

\*(pRegion+i) = i;

// Вывод значений заполненного блока

for(int i=0; i<10; i++)

{

cout<< (\*pRegion) <<'\n';

pRegion++;

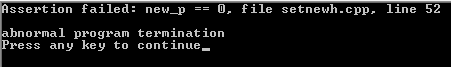
}

// Освобождение блока памяти

free(pRegion);

return 0;

}



#include <iostream>

using namespace std;

int main(void)

{

setlocale(LC\_ALL, "Rus");

int\* pRegion;

// Выделение области памяти

if((pRegion =(int\*)malloc(10\*sizeof(int)))=NULL){

cout<< "He достаточно памяти\n";

// Завершить программу если не

// хватает памяти для выделения

return 1;

}

// Заполнение выделенного блока памяти

for(int i= 0; i< 10; i++)

\*(pRegion+ i) = i;

// Вывод значений заполненного блока

for(int i= 0; i< 10; i++){

cout << (\*pRegion) << '\n';

pRegion++;

}

// Освобождение блока памяти

free(pRegion);

getchar(); getchar();

return 0;

}

Пример динамического выделения памяти под двумерный массив Array [10] [20] переменных типа int выглядит следующим образом:

Array = (int\*)malloc(10\*20\*sizeof(int));

В отличие от функций работы с динамической памятью malloc, calloc и free, заимствованных в C++ из стандарта ANSI С для совместимости, новые операторы гибкого распределения памяти new и delete обладают дополнительными возможностями:

* Как отмечаюсь выше. функции malloc и calloc возвращают пустой указатель, который в дальнейшем требуется приводить к заданному типу. Оператор new возвращает указатель на тип, и дополнительных преобразований уже не требуется.
* Операция new предполагает возможность использования совместно с библиотечной функцией set\_new\_handler, позволяющей пользователю определить свою собственную процедуру обработки ошибки при выделении памяти.
* Операторы new и delete могут быть перегружены с тем, чтобы они, например, могли принимать дополнительные параметры или выполняли специфические действия с учетом конкретной ситуации работы с памятью.

Операторы new и delete имеют две формы:

* управление динамическим размещением в памяти единичного объекта;
* динамическое размещение массива объектов.

Синтаксис при работе с единичными объектами следующий:

тип\_объекта \*имя = new тип \_объекта;

delete имя;

При управлении жизненным циклом массива объектов синтаксис обоих операторов имеет вид:

тип\_объекта \*имя = new тип \_объекта[число];

delete[] имя;

Здесь число в операторе new [] характеризует количество объектов типа тип\_объекта, для которых производится выделение области памяти. В случае успешного резервирования памяти переменная-указатель имя ссылается на начало выделенной области. При удалении массива его размер указывать не нужно.

Форма оператора delete должна обязательно соответствовать форме оператора new для данного объекта: если выделение памяти проводилось для единичного объекта (new), освобождение памяти также должно осуществляться для единичного объекта (delete). В случае применения оператора new[] (для массива объектов), в конечном итоге высвобождение памяти должно быть произведено с использованием оператора delete[].

Поясним вес вышесказанное на примере

#include <iostream.h>

#include <new.h>

void newHandler()

{

cout << "Хм…Недостаточно памяти!\n"

}

int main()

{

int\* ptr;

set\_new\_handler(newHandler);

ptr = new int[100];

for(int i=0; i<100; i++)

\*(ptr+i) = 100-i;

for(int i=0; i<100; i++)

cout << \*(ptr+i) << " ";

delete[] ptr;

long double\* lptr;

if(lptr = new long double[999999])

{

cout << "\nГотово!";

delete[] lptr;

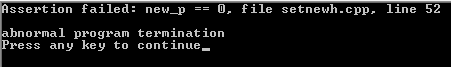
}

else

cout <<"\nНеудача... ";

return 0;

}



#include <new.h>

#include <iostream>

using namespace std;

void newHandler()

{

cout << "Хм…Недостаточно памяти!\n";

}

int main()

{

setlocale(LC\_ALL, "Rus");

int\* ptr;

set\_new\_handler(newHandler);

ptr = new int[100];

for(int i=0; i<100; i++)

\*(ptr+i) = 100-i;

for(int i=0; i<100; i++)

cout << \*(ptr+i) << " ";

delete[] ptr;

long double\* lptr;

if(lptr = new long double[999999]){

cout << "\nГотово!";

delete[] lptr;

}

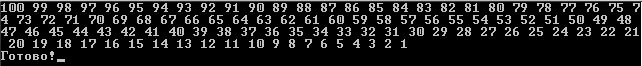
else

cout << "\nНеудача... ";

getchar(); getchar();

return 0;

}



В начале представленной программы подключается уже знакомый заголовочный файл iostream.h для использования операций ввода/вывода, а также файл new.h для определения обработчика ошибок set new\_handler. Функция newHandler (), реализованная в начале программы, является собственно обработчиком ошибок выделения памяти. Она подключается в строке

set\_new\_handler(newHandler);

заменяя тем самым стандартную процедуру обработки. Теперь всякий раз когда программе будет необходимо выделить некоторую область динамической памяти, но в силу аппаратных или программных ограничений осуществить это будет невозможно, на экране будет появляться надпись:

"Недостаточно памяти!".

В главной функции программы следует объявление целочисленного указателя ptr, которому тут же оператором new присваивается адрес выделенного блока динамической памяти для 100 значений типа int. Далее область памяти заполняется в цикле числами от 100 до 1. а в следующем цикле осуществляется вывод ее содержимого, после чего выделенный блок динамической памяти освобождается (оператор delete []). Следующий шаг - попытка выделения памяти для большого массива типа long double. Если результат операции new[] не нулевой, указателю lptr будет присвоено значение адреса зарезервированной памяти с выводом надписи " Готово! " и последующим высвобождением области. В случае неудачного выделения выводится сообщение "Неудача... "

### Функция memset().

Эту функцию можно использовать для присвоения символьных значений ячейкам динамически выделенной памяти — одному или нескольким байтам. Прототип функции memset() выглядит следующим образом:

void \*memset(void \*dest, int cchar, size\_t count);

После вызова memset() переменная dest указывает на count байтов памяти, которые проинициализированы символьным значением cchar. В следующем примере показана разница между статическим и динамическим объявлением структуры:

// 05.17MEMSET.CPP

// функция memset(). Динамическое выделение памяти

//

#include "stdafx.h"

#include <iostream>

#include <math.h>

using namespace std;

struct keybits

{

unsigned char rshift, lshift, ctrl, alt,

scroll, numlock, caplock, insert;

};

void \*memset(void \*dest, int cchar, size\_t count);

void main(void)

{

keybits stkgarbage, \*pstkinitialized;

pstkinitialized=new keybits;

memset(pstkinitialized, 0, sizeof(keybits));

}

Благодаря функции memset(), динамически созданная структура, на которую указывает переменная pstkinitialized, содержит все нули, в то время как статически созданная структура stkgarbage заполнена произвольными значениями. При вызове функции memset() используется операция sizeof () вместо "зашитого" в оператор фиксированного числа. При этом процедура может автоматически выбирать размер любого передаваемого объекта. Напомним также, что в С++ при описании структурных переменных, как в случае с переменными stkgarbage и pstkinitialized, не обязательно использовать ключевое слово struct перед полем тега структуры (keybits).

## Динамические массивы

Обычно, объем памяти, необходимый для той или иной переменной, задается еще до процесса компиляции посредством объявления этой переменной. Если же возникает необходимость в создание переменной, размер которой неизвестен заранее, то используют динамическую память. **Резервирование** и **освобождение** памяти в программах на C++ может происходить в любой момент времени. Осуществляются операции *распределения* памяти двумя способами:

* с помощью функции **malloc**, **calloc**, **realloc** и **free;**
* посредством оператора **new** и **delete**.

Функция **malloc** *резервирует* непрерывный блок ячеек памяти для хранения указанного объекта и возвращает указатель на первую ячейку этого блока. Обращение к функции имеет вид:

**void \*malloc(size);**

Здесь **size** — целое беззнаковое значение, определяющее размер выделяемого участка памяти в байтах. Если резервирование памяти прошло успешно, то функция возвращает переменную типа **void \***, которую можно привести к любому необходимому типу указателя.

Функция — **calloc** также предназначена для выделения памяти. Запись ниже означает, что будет выделено **num** элементов по **size** байт.

**void \*calloc (nime, size);**

Эта функция возвращает указатель на выделенный участок или **NULL** при невозможности выделить память. Особенностью функции является обнуление всех выделенных элементов.

Функция **realloc** *изменяет размер* выделенной ранее памяти. Обращаются к ней так:

**char \*realloc (void \*p, size);**

Здесь **p** — указатель на область памяти, размер которой нужно изменить на **size**. Если в результате работы функции меняется адрес области памяти, то новый адрес вернется в качестве результата. Если фактическое значение первого параметра **NULL**, то функция **realloc** работает также, как и функция **malloc**, то есть выделяет участок памяти размером **size** байт.

Для освобождения выделенной памяти используется функция **free**. Обращаются к ней так:

**void free (void \*p size);**

Здесь **p** — указатель на участок памяти, ранее выделенный функциями **malloc**, **calloc** или **realloc**.

Операторы **new** и **delete** аналогичны функциям **malloc** и **free**. **New** выделяет память, а его единственный аргумент — это выражение, определяющее количество байтов, которые будут зарезервированы. Возвращает оператор указатель на начало выделенного блока памяти. Оператор **delete** освобождает память, его аргумент — адрес первой ячейки блока, который необходимо освободить.

*Динамический массив* — массив переменной длины, память под который выделяется в процессе выполнения программы. Выделение памяти осуществляется функциями **calloc, malloc** или оператором **new**. Адрес первого элемента выделенного участка памяти хранится в переменной, объявленной как указатель. Например, следующий оператор означает, что описан указатель **mas** и ему присвоен адрес начала непрерывной области динамической памяти, выделенной с помощью оператора **new**:

**int \*mas=new int[10];**

Выделено столько памяти, сколько необходимо для хранения 10 величин типа int.

Фактически, в переменной **mas** хранится адрес нулевого элемента динамического массива. Следовательно, адрес следующего, первого элемента, в выделенном участке памяти — **mas**+1, а **mas**+i является адресом i-го элемента. Обращение к i-му элементу динамического массива можно выполнить, как обычно mas[i], или другим способом **\*(mas +i)**. Важно следить за тем, чтобы не выйти за границы выделенного участка памяти.

Когда динамический массив (в любой момент работы программы) перестает быть нужным, то память можно освободить с помощью функции **free** или оператора **delete**.

### Создание динамической переменной

#include <iostream>

using namespace std;

int main()

{

int \*ptrvalue = new int; // динамическое выделение памяти под объект типа int

\*ptrvalue = 9; // инициализация объекта через указатель

//int \*ptrvalue = new int (9); инициализация может выполнятся сразу при объявлении динамического объекта

cout << "ptrvalue = " << \*ptrvalue << endl;

delete ptrvalue; // высвобождение памяти

system("pause");

return 0;

}



### Создание динамического массива

//Пример использования функции malloc и free

#include <iostream>

using namespace std;

int main()

{

setlocale(LC\_ALL,"Rus");

int i, n;

float \*a; //указатель на float

float s;

cout<<"\n"; cin>>n; //ввод размерности массива

//выделение памяти под массив из n вещественных элементов

a=(float \*)malloc(n\*sizeof(float));

cout<<"Введите массив A \n";

//ввод элементов массива

for (i=0; i<n; i++)

{

cin>>\*(a+i);

}

//накапливание суммы элементов массива

for (s=0, i=0; i<n; i++)

s+=\*(a+i);

//вывод значения суммы

cout<<"S="<<s<<"\n";

//освобождение памяти

free(a);

system("pause");

return 0;

}



#include <iostream>

using namespace std;

int main()

{

int num; // размер массива

cout << "Enter integer value: ";

cin >> num; // получение от пользователя размера массива

int \*p\_darr = new int[num]; // Выделение памяти для массива

for (int i = 0; i < num; i++) {

// Заполнение массива и вывод значений его элементов

p\_darr[i] = i;

cout << "Value of " << i << " element is " << p\_darr[i] << endl;

}

delete [] p\_darr; // очистка памяти

cout << "memory deleted";

cin >> num;

return 0;

}



### Создание одномерного динамического массива, заполненного случайными числами

#include <iostream>

// в заголовочном файле <ctime> содержится прототип функции time()

#include <ctime>

// в заголовочном файле <iomanip> содержится прототип функции setprecision()

#include <iomanip>

using namespace std;

int main()

{

srand(time(0)); // генерация случайных чисел

float \*ptrarray = new float [10]; // создание динамического массива вещественных чисел на десять элементов

for (int count = 0; count < 10; count++)

ptrarray[count] = (rand() % 10 + 1) / float((rand() % 10 + 1)); //заполнение массива случайными числами с масштабированием от 1 до 10

cout << "array = ";

for (int count = 0; count < 10; count++)

cout << setprecision(2) << ptrarray[count] << " ";

delete [] ptrarray; // высвобождение памяти

cout << endl;

system("pause");

return 0;

}



### Объявление и удаление двумерного динамического массива

#include <iostream>

#include <ctime>

#include <iomanip>

using namespace std;

int main()

{

srand(time(0)); // генерация случайных чисел

// динамическое создание двумерного массива вещественных чисел на десять элементов

float \*\*ptrarray = new float\* [2]; // две строки в массиве

for (int count = 0; count < 2; count++)

ptrarray[count] = new float [5]; // и пять столбцов

// заполнение массива

for (int count\_row = 0; count\_row < 2; count\_row++)

for (int count\_column = 0; count\_column < 5; count\_column++)

ptrarray[count\_row][count\_column] = (rand() % 10 + 1) / float((rand() % 10 + 1)); //заполнение массива случайными числами с масштабированием от 1 до 10

// вывод массива

for (int count\_row = 0; count\_row < 2; count\_row++)

{

for (int count\_column = 0; count\_column < 5; count\_column++)

cout << setw(4) <<setprecision(2) << ptrarray[count\_row][count\_column] << " ";

cout << endl;

}

// удаление двумерного динамического массива

for (int count = 0; count < 2; count++)

delete []ptrarray[count];

system("pause");

return 0;

}

