# 4. УКАЗАТЕЛИ

## 4.1. Определение указателей

Указатель – это переменная, содержащая адрес некоторого объекта, например, другой переменной, точнее адрес первого байта этого объекта. Это дает возможность косвенного доступа к этому объекту через указатель. Пусть **x** – переменная типа **int**. Обозначим через **px** указатель. Унарная операция & выдает адрес объекта, так что оператор px = &x;

присваивает переменной **px** адрес переменной **x**. Говорят, что **px** «указывает» на **x**. Операция & применима только к адресным выражениям, так что конструкции вида &(**x**-1) и &3 незаконны.

Унарная операция  называется операцией разадресации или операцией разрешения адреса. Эта операция рассматривает свой операнд как адрес и обращается по этому адресу, чтобы извлечь объект, содержащийся по этому адресу.

Следовательно, если **y** тоже имеет тип **int**, то y

= px;

присваивает **y** содержимое того, на что указывает **px**. Так, последовательность px = &x; y = px;

присваивает **y** то же самое значение, что и оператор y

= x;

Все эти переменные должны быть описаны:

int x, y; int px;

Последнее – описание указателя. Его можно рассматривать как мнемоническое. Оно говорит, что комбинация **px** имеет тип **int**, или, иначе, **px** есть указатель на **int**. Это означает, что если **px** появляется в виде **px**, то это эквивалентно переменной типа **int**.

Из описания указателя следует, что он может указывать только на определенный вид объекта (в данном случае **int**). Разадресованный указатель может входить в любые выражения там, где может появиться объект того типа, на который этот указатель ссылается. Так, оператор y = px + 2;

присваивает **y** значение, на **2** больше, чем **x**.

Заметим, что приоритет унарных операций  и & таков, что эти операции связаны со своими операндами более крепко, чем арифметические операции, так что выражение y = px + 2

берет то значение, на которое указывает **px**, прибавляет 2 и присваивает результат переменной **y**.

Если **px** указывает на **x**, то

px = 3; полагает

**x** равным 3, а

px + = 1; увеличивает **x** на 1 так же,

как и выражение

(px)++

Круглые скобки здесь необходимы. Если их опустить, то есть написать **px**++, то, поскольку унарные операции, подобные  и ++, выполняются справа налево, это выражение увеличит **px**, а не ту переменную, на которую он указывает.

Если **py** – другой указатель на **int**, то можно выполнить присвоение py = px;

Здесь адрес из **px** копируется в **py**. В результате **py** указывает на то же, что и **px**.

## 4.2. Указатели и массивы

Массив – это совокупность элементов одного типа, которые расположены в памяти ЭВМ подряд, один за другим.

Признаком объявления массива являются квадратные скобки. Объявить массив из 8 элементов типа **double** можно так:

double arr [8];

Чтобы обратиться к элементу этого массива, нужно применить операцию индексирования **arr** [**ind**]. Здесь **ind** – целое выражение, которое называется индексом. Нумеруются элементы массива **начиная с 0**, и поэтому вышеприведенное описание означает, что в памяти ЭВМ зарезервировано место под 8 переменных типа **double**, а сами эти переменные есть **arr** [0], **arr** [1], …, **arr** [7].

Напишем программу, в которой идет подсчет числа появлений в потоке ввода цифр, пробельных символов и всех прочих символов.

Число пробельных символов обозначим через **nw**, прочих символов – **no**. Число появлений цифр будем хранить в массиве **nd**:

```
void main( ){ int ch, nw = 0, no = 0;
int nd [10]; for (int i=0; i < 10; i++)
nd[i] = 0; while ( ( ch = cin.get( ) )
!= EOF) if(c >= '0' &&  c <= '9')
++nd[c - '0'];
else if(ch == ' '|| ch == '\n' || ch == '\t') ++nw; else ++no;
cout << " цифра \n"; for(int i = 0; i < 10; i ++) cout <<
i<< "  вошла" << nd[i] << "  раз \n"; cout << "
пробельных символов – "
<< nw << " прочих символов – "<< no << "\n";  }
```

При объявлении массива его можно инициализировать:

```
int mas[ ] = { 1, 8, 7, 0, 3, 15, -5 };
char arr [ ] = { 'h', 'e', 'l', 'l', 'o', '\n', '\0'};
```

Последнюю инициализацию разрешается выполнять проще:

```
char arr[ ] = "hello\n";
```

Такой синтаксис инициализации разрешен только для строк. Компилятор сам вычислит необходимый размер памяти с учетом автоматически добавляемого в конец строки символа '\0' с кодом 0, который является признаком завершения строки.

В языке C++ имя массива является **константным указателем** на нулевой элемент этого массива:

```
int mas[30]; int
pm; pm =
&mas[0];
```

Последний оператор можно записать и так: pm = mas;

Операция индексирования массива [ ] имеет 2 операнда – имя массива, т. е. указатель, и индекс, т. е. целое: **arr**[**i**]. В языке C++ любое выражение **указатель[индекс]** трактуется как

(**указатель** + **индекс**) и всегда автоматически преобразуется к такому виду при компиляции.

Таким образом, **arr**[**5**] эквивалентно (**arr** + 5), и это можно записать даже как **5**[**arr**], так как это все равно проинтерпретируется как (**5** + **a**). Здесь складываются указатель **arr** и целое **5**. В связи с этим рассмотрим так называемую арифметику над указателями, или адресную арифметику.

## 4.3. Адресная арифметика

***Указатель можно складывать с целым.***

Если к указателю **pa** прибавляется целое приращение **i**, то приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель **pa**.

Таким образом, **pa + i** – это адрес **i**-го элемента после **pa**, причем считается, что размер всех этих **i** элементов равен размеру объекта, на который указывает **pa**. Если **a** – массив, то a + i – адрес i-го элемента этого массива, т. е.

&a[i] равен a + i и a[i] равняется □(a + i).

doble b[100]; double □pb = b; pb++; // Это эквивалентно pb = pb + 1. // Здесь указатель pb будет указывать на элемент массива b[1].

pb += 4; // Здесь pb указывает на элемент массива b[5].

Однако нельзя написать **b**++ или **b = b + i**, так как имя массива **b** – это константный указатель и его изменять нельзя.

***Указатели можно сравнивать.***

Если **p** и **q** указывают на элементы одного и того же массива, то отношения сравнения, такие как > >= и т. д., работают обычным образом. Например, p < q истинно, т. е. равно 1, если **p** указывает на более ранний элемент массива, чем **q**. Любой указатель можно сравнить с помощью операций == и != с так называемым нулевым указателем **NULL**, который ни на что не указывает. Однако не рекомендуется сравнивать указатели, указывающие на различные массивы.

***Указатели можно вычитать.***

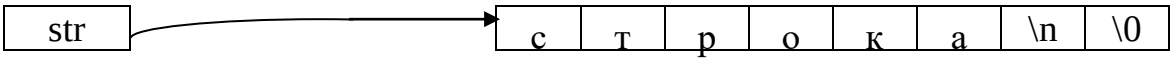
Если **p** и **q** указывают на элементы одного и того же массива, то **p - q** дает количество элементов массива между **p** и **q**.

## 4.4. Символьные массивы и строки

Строка является массивом символов. При этом значением строки является указатель на ее начальный символ:

char □str = (char*)"строка\n";

Здесь указатель на символы **str** будет содержать адрес первого символа 'с' строки "строка\n", которая размещается в некоторой области памяти, начиная с этого адреса:



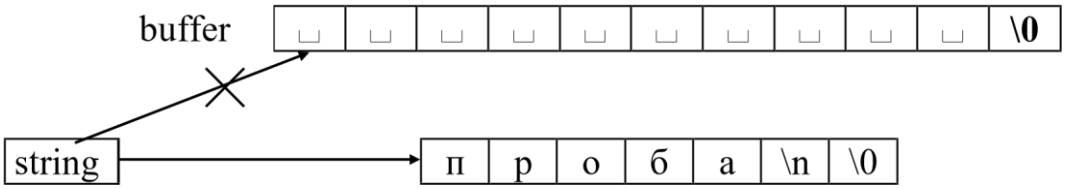

Здесь **str**[3] есть символ 'о'.

Рассмотрим фрагмент программы:

```
char buffer[ ] = "␣␣␣␣␣␣␣␣␣␣";        // Инициализация
                                        // строки из 10 пробелов.
char *string = buffer;              // string указывает на начало буфера.
string = (char*)"проба\n";          // Присваивание!
```

При инициализации создается строка **buffer** и в нее помещаются символы (здесь 10 пробелов). Инициализация char *string=buffer устанавливает указатель **string** на начало этой строки.

Операция же присваивания в последней строке не копирует приведенную строку "проба\n" в массив **buffer**, а изменяет значение указателя **string** так, что он начинает указывать на строку "проба\n":




Чтобы скопировать строку "**проба\n**" в **buffer**, можно поступить так:

```
char buffer[ ] = "␣␣␣␣␣␣␣␣␣␣";
char *p = (char*)"проба\n"; int i
= 0; while((buffer[i] = p[i]) != '\0')
i++;
```

Или так:

```
char buffer[ ] = "␣␣␣␣␣␣␣␣␣␣";
char * p = (char*)"проба\n"; char
*buf = buffer; while (*buf++ =
*p++ );
```

Здесь сначала ***p** копируется в ***buf**, т. е. символ 'п' копируется по адресу **buf**, который совпадает с адресом **buffer**, т. е. **buffer**[0] становится равен 'п'. Затем происходит увеличение указателей **p** и **buf**, что приводит к продвижению по строкам "**проба\n**" и **buffer** соответственно.

Последним скопированным символом будет '\0'. Значение этого символа – 0, поэтому оператор while завершит работу.

Еще проще воспользоваться библиотечной функцией, прототип которой находится в файле **cstring**:

strcpy( buffer, "проба\n" );

При копировании необходимо обеспечить, чтобы размер памяти, выделенной под **buffer**, был достаточен для хранения копируемой строки.

Отметим, что при компиляции возможно появление предупреждения, что использование функции strcpy() небезопасно. Для исключения такого предупреждения нужно просто добавить директиву

#define _CRT_SECURE_NO_WARNINGS

в начало файла перед всеми #include (в среде Visual Studio).

## 4.5. Многомерные массивы

Двумерный массив рассматриваются как массив элементов, каждый из которых является одномерным массивом. Трехмерный – как массив, элементами которого являются двумерные массивы и т. д. После объявления

int a[5][6][7];

в программе могут появиться выражения:

a[i][j][j] – объект типа int;
a[2][0] – объект типа int* – одномерный массив из 7 целых;
a[1] – двумерный массив из 6*7 = 42 целых; a – сам трехмерный массив.

Так как элементом массива **a** является двумерный подмассив размером 6*7, то при выполнении выражения **a + 1** происходит смещение на величину элемента массива **a**, т. е. переход от **a**[0] к **a**[1]. Значение адреса при этом увеличивается на 6*7*sizeof(int) = 84.

Для двумерного массива **mas** выражение **mas**[**i**][**j**] интерпретируется как *(*(**mas + i**) +**j** ). Здесь **mas**[**i**] – константный указатель на **i**-ю строку массива **mas**.

В памяти массивы хранятся по строкам, т. е. при обращении к элементам в порядке их размещения в памяти быстрее всего меняется самый правый индекс.

Так, для массива c[2][3] его шесть элементов расположены в памяти так:

c[0][0] c[0][1] c[0][2] c[1][0] c[1][1] c[1][2].

Многомерные массивы также можно инициализировать при описании: int d[2][3]={ 1, 2, 0, 5 };

В этом случае первые 4 элемента массива получат указанные значения, а остальные два инициализируются нулями.

Если инициализируется многомерный массив, то самую первую размерность можно не задавать. В этом случае компилятор сам вычисляет размер массива:

```
int f [ ][2] = { 2, 4, 6, 1 };                    // массив  f [2][2]; int a
[ ][2][2] = { 1, 2, 3, 4, 5, 6, 7, 8 };      // массив  a [2][2][2].
```

Инициализирующее выражение может иметь вид, отражающий факт, что массив является, например, двумерным:

int c[2][3]={ {1, 7}, {-5, 3} };

В этом случае в матрице **c** инициализированы нулевой и первый столбцы, а второй столбец, т. е. элементы c[0][2] и c[1][2], инициализируется нулями.

## 4.6. Указатели и многомерные массивы

Рассмотрим разницу между объектами **a** и **b**, описанными следующим образом:  double a[20][20];  double * b[20];

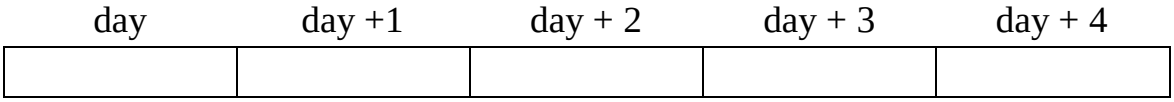
И a и b можно использовать одинаково в том смысле, что как a[6][5], так и b[6][5] являются обращениями к отдельному значению типа  double. Но a – настоящий массив: под него отводится 400 ячеек памяти и для нахождения любого указанного элемента проводятся обычные вычисления с индексами, которые требуют умножения. Для b описание выделяет только 20 указателей. Каждый из них должен быть определен так, чтобы он указывал на массив double.

Если предположить, что каждый из них указывает на массив из 20 элементов, то тогда где-то будет отведено 400 ячеек памяти плюс еще 20 ячеек для указателей. Таким образом, массив указателей использует несколько больше памяти и может требовать наличия явного шага инициализации. Но при этом возникают два преимущества: доступ к элементу осуществляется косвенно через указатель, а не посредством умножения и сложения, а строки массива могут иметь различные длины.

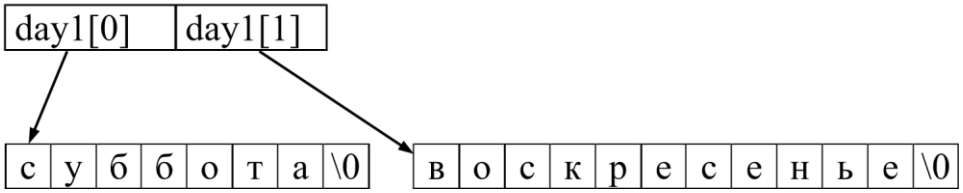
Это означает, что каждый элемент b не должен обязательно указывать на вектор из 20 элементов. Эта разница демонстрируется в следующем примере: char day [5][12] = {// В каждой строке 12 символов.

"понедельник",
"вторник",
"среда",
"четверг",
"пятница"
};

Здесь константные указатели day[0], day[1], …, day[4] адресуют участки памяти одинаковой длины – 12 байт каждый:




```
const char * day1[2] =
{"суббота",                 // 7 символов + '/0'
 "воскресенье"              // 11 символов + '/0'
};
```




Переменные-указатели day1[0] и day1[1] адресуют участки памяти соответственно в 8 и 12 байт.