Лекция 8 Массивы и указатели

8.1. *Указатели*

8.1.1. ***Организация памяти в компьютере*** – массив последовательно пронумерованных (адресуемых) ячеек, которые можно обрабатывать по отдельности, либо группами. Указатель – группа ячеек памяти, которая может содержать адрес.

8.1.2. ***Пример***: пусть **c** – переменная типа **char**. Операция **p = &c;** помещает в ячейку **p** адрес **c**. Одноместная операция **&** называется операцией *адресации*. Применив к указателю **p**, одноместную операцию *разыменования* *****, можно получить объект памяти (переменную или константу), на который указывает **p**.

8.1.3. ***Пример***: фрагмент программы.

```
int a = 1;
int *p;
p = &a;
*p = 2;
printf ("Значение переменной a = %d\n", *p);
printf ("Адрес переменной a = %d\n", p);
```

В результате выполнения фрагмента будет напечатано:

```
Значение переменной a = 2
Адрес переменной a = 0xbffff7a4
```

Напечатанный адрес – 16-чное число: **0x** – признак 16-ичности, затем само число: $b*16^7+ f*16^6+ f*16^5+ f*16^4+ f*16^3+ 7*16^2+ a*16+ 4 = 11*16^7+ 15*16^6+ 15*16^5+ 15*16^4+ 15*16^3+ 7*16^2+ 10*16+ 4 =$ …

8.2. *Указатели и массивы.*

8.2.1. ***Создание указателя на массив***. Указатель на первый элемент массива можно создать, присвоив переменной типа указатель на тип элемента массива имя массива без индекса. Пример:

```
int array[15];
int *p;
p = array;
```

Теперь значением и **array**, и **p** является адрес первого элемента массива **array[15]**. Различие в том, что значение **array** изменить нельзя, а значение **p** – можно. Еще один способ присвоить указателю **p** адрес первого элемента массива **array[15]**:

```
p = &array[0];
```

хотя это и не принято.

Отметим, что в отличие от **p**, **array** не является именем переменной, так что писать **p = array** и **p++** можно, а такие конструкции, как **array = p** и **array++** запрещены.

8.2.2. Адресная арифметика и обращение к элементам массива. В следующем фрагменте Си-программы два последних присваивания помещают в **y** значение 8-го элемента массива **array**:

```
int array[10];
int *p;
a = &array[0];
x = *p;
y = *(p + 7);
y = array[7];
```

Указатель **p** имеет тип (в данном случае – **int**): следовательно, значение **p + 7** равно *p + 7·sizeof*(*int*).

Операции **y = *(p + 7);** и **y = array[7];** эквивалентны. Но во многих реализациях Си операция адресной арифметики ***(p + 7)** выполняется быстрее, чем операция индексирования **array[7]**, что и объясняет использование адресной арифметики при обращении к элементам массива.

8.2.3. ***Индексирование указателей***. В 8.2.1. было отмечено, что имя массива является указателем. Если **p** – указатель на массив, то **p** можно индексировать как массив:

В следующем фрагменте программы второе и третье присваивания заносят число 100 в 6-ой элемент массива **a**:

```
int *p, a[10];
p = a;
*(p + 5) = 100; /* адресная арифметика */
*p[5] = 100;    /* индексирование указателя */
```

8.2.4. ***Сравнение указателей***. Если **p** и **q** являются указателями элементов одного и того же массива и **p < q**, то **q – p + 1** равно количеству элементов массива от **p** до **q** включительно.

8.3. ***Передача одномерного массива в функцию***. Передается указатель на первый элемент массива. Для этого в объявлении функции соответствующий аргумент должен быть описан как **int *x**, либо **int x[15]**, либо **int x[]** (массив без указания размера).

8.4. ***Двухмерный массив*** – это массив одномерных массивов, поэтому при объявлении массива и при обращении у массиву длина (соответственно – индексное выражение) по каждому измерению заключается в свои квадратные скобки. Пример:

```
#include <stdio.h>
int main() {
     int j, i, matr[3][4];
     for(j = 0; j < 3; j++)
         for(i = 0; i < 4; i++)
              matr[j][i] = 4 * j + i + 1;

     /* вывод массива на экран */
     for(j = 0; j < 3; j++) {
         for(i = 0; i < 4; i++)
              printf ("%3d", matr[j][i]);
         printf ("\n");
     }
```

```
    return 0;
}
```

Будет сформирована и напечатана следующая матрица **matr**: размера 3×4:

	0	1	2	3
0	**1**	**2**	**3**	**4**
1	**5**	**6**	**7**	**8**
2	**9**	**10**	**11**	**12**

Первый индекс (**j**) – номер строки, второй индекс (**i**) – номер столбца.

В памяти сначала первая строка, потом – вторая, потом – третья.

Если двухмерный массив используется в качестве аргумента функции, то в нее передается только указатель на первый элемент массива.

8.5. ***Многомерные массивы*** объявляются как массивы двухмерных, трехмерных и т.д. массивов.

8.6. ***Инициализация массивов***.

тип имя_массива[*размер1*]…[*размерN*] = {*список_значений*};

Список_значений – это список констант типа, совместимого с типом массива, разделенных запятыми. Пример:

```
int sqrs[10][2] = {
   1,   1,
...2,   4,
   3,   9,
...4,   16,
   5,   25,
...6,   36,
   7,   49,
...8,   64,
...9,   81,
...10,   100
}
```

```
int sqrs[10][2] = {
  {1,   1},
..{2,   4},
  {3,   9},
..{4,   16},
  {5,   25},
..{6,   36},
  {7,   49},
..{8,   64},
..{9,   81,
..{10,   100}
}
```

Во втором случае для наглядности используются дополнительные фигурные скобки, в которые заключаются элементы инициализации каждого измерения (*группировка подагрегатов*). Если внутри группы недостаточно констант инициализации, оставшиеся элементы группы автоматически инициализируются нулями.

8.7. ***Операция*** **sizeof**.

8.7.1. Одноместная операция **sizeof** позволяет определить длину операнда в байтах. Операндами операции **sizeof** могут быть типы, либо переменные. Результат операции **sizeof** имеет тип **size_t**, специально определенный в языке Си (он приблизительно соответствует целому беззнаковому типу).

8.7.2. Операция **sizeof** выполняется во время компиляции, ее результат представляет собой константу.

8.7.3. Операция **sizeof** помогает улучшить переносимость программ. Если программа должна выполняться на разных компьютерах, нельзя полагаться на то, что, размер, например, целого числа на всех компьютерах будет одинаковым. В программе нужно определить этот размер с помощью **sizeof**.

8.7.4. Операция **sizeof** позволяет определить, например, объем памяти в байтах, занимаемый двумерным массивом. Его можно вычислить по формуле:

$$Number\_of\_bytes = d1 \times d2 \times sizeof(element\_type)$$

где $d1$ – количество элементов по первому измерению, $d2$ – количество элементов по второму измерению, *element_type* – тип элемента массива.

А в некоторых случаях можно поступить проще:

Number_of_bytes = sizeof(*имя_массива*).

8.7.5. ***Применение* sizeof *к массивам***. Применив **sizeof** к массиву, можно получить размер памяти (в байтах), занимаемой соответствующим массивом. ***Пример*:**

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

int main(int argc, char **argv) {
char buffer[10]; /* массив из 10 элементов типа char */

/* Копирование 9 символов из argv[1] в buffer.
 *  sizeof(char) считается равным 1, так что число
 *  элементов массива buffer равно его размеру в байтахs.
*/
strncpy(buffer, argv[1], sizeof(buffer) - sizeof(char));

    /* Присваивание последнему элементу buffer значения null
    */
buffer[sizeof(buffer) - 1] = '\0';

return 0;
}
```

В рассмотренном примере **sizeof buffer** эквивалентно **10*sizeof(char)**, или 10.

8.7.6. **sizeof** можно применять только к «полностью» определенным типам. Для массивов это означает, что размерности массива должны присутствовать в его объявлении и что тип элементов массива должен быть полностью определен. Например, если объявление массива имеет вид: **extern int arr[];**, то операция **sizeof(arr)** в данном программном файле ошибочна, так как у компилятора нет возможности узнать, сколько элементов содержит массив **arr**.

8.8. ***Операции над указателями. Адресная арифметика.***

8.8.1. В языке Си допустимы только *три* операции над указателями: *сложение* указателя с целым числом, *вычитание* целого числа из указателя и *вычитание указателей*. ***Пример***. Пусть переменная типа **int** занимает в памяти 4 байта и пусть текущее значение указателя **p1** типа **int*** равно 2012. Тогда после операции **p1++** указатель **p1** будет иметь значение 2016, а не 2013, а после операции **p1 - 3** –

значение 2000. То есть при увеличении (уменьшении) на целое число **i** указатель будет перемещаться на **i** целочисленных ячеек в сторону увеличения (уменьшения) их адресов. Это справедливо для указателей на объекты любых типов.

Вычитание указателей позволяет определить, сколько объектов рассматриваемого типа поместится между соответствующими указателями.

8.8.2. ***Сравнение указателей***. Два указателя модно сравнивать между собой. Например, следующий оператор правильный:

```
if(p < q) printf("p ссылается на меньший адрес, чем q");
```

8.9. ***Указатели и аргументы функций***.

8.9.1. ***Пример***. Функция **void swap(int x, int y)**, меняющая местами значения переменных **x** и **y**.

Неправильный вариант:

```
void swap(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

Правильный вариант:

```
void swap(int *px, int *py) {
     int tmp;
     tmp = *px;
     *px = *py;
     *py = tmp;
}
```

Неправильный вариант не учитывает того обстоятельства, что *все аргументы функции передаются только по значению*: операция **x = y;** не имеет смысла (ошибочна), так как значение **x** (целое число) не может быть левой частью присваивания.

Используя аргументы-указатели, функция может обращаться к объектам вызвавшей ее функции.

8.9.2. Использование указателей позволяет не дублировать массивы, передавая их функции: функции достаточно передать указатель на первый элемент массива.

8.10. ***Преобразование типа указателя***. Указатель можно преобразовать к другому типу, но такое преобразование типов обязательно должно быть явным: должна быть обязательно указана операция приведения типов. Однако такое преобразование типов может вызвать непредсказуемое поведение программы.

8.10.1. ***Пример***. Рассмотрим программу:

```
#include <stdio.h>
int main() {
    double x = 200.35, y;
    int *p;
    p = (int *)&x;/* здесь явное преобразование типов
                  /* необходимо, так как &x ссылается на
                  /* double, а p имеет тип *int
    y = *p;       /* ожидается, что y будет присвоено
                  /* значение 200.35
    printf("значение x равно %f \n", x);
```

```
    printf("значение x равно %f", y);
    return 0;
}
```

Ожидается, что будет дважды напечатано: **`значение x равно 200.350000`**.

Однако при использовании *GCC* (и многих других систем программирования) на самом деле будет напечатано:

```
значение x равно 200.350000
значение x равно <другое число>
```

`<другое число>` зависит от содержания области памяти, адресуемой **`&y`** длиной в 8 байт (например, 858993459.000000). Это объясняется тем, что поскольку **`p`** имеет тип **`(int *)`**, присваивание **`y = *p;`** меняет только первые четыре байта области памяти с адресом **`&y`**.

Таким образом, необходимо учитывать, что операции с указателями выполняются в соответствии с базовым типом указателей: синтаксически допускается ссылка на объект, имеющий тип, отличный от типа указателя, но указатель все равно будет считать, что он ссылается на объект своего типа.

8.10.2. Разрешено также *преобразование целого в указатель и наоборот*. Однако пользоваться этим нужно очень осторожно, так как при этом легко получить непредсказуемое поведение программы (в частности, разыменование **`NULL`**).

8.10.3. Допускается присваивание указателя типа **`void *`** указателю любого другого типа (и наоборот) без явного преобразования типа указателя. Это позволяет использовать указатель типа **`void *`**, когда тип объекта неизвестен.

Примеры. (1) Использование типа **`void *`** в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа. (2) Функция **`malloc()`** возвращает значение типа **`void *`**.

8.11.

8.11.1. Массивы указателей.

8.11.2. Инициализация указателей.

8.11.3. Квалификатор **`restrict`**.