ШАГ компьютерная академия

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

Урок №12

Программирование на языке C

Содержание

1. Статическое и динамическое выделение памяти

Статическая память — это область хранения всех глобальных и статических переменных. Переменные статической памяти объявляются лишь единожды и уничтожаются по завершении программы.

Динамическая память или память свободного хранения отличается от статической тем, что программа должна явным образом запросить память для элементов, хранимых в этой области, а затем освободить память, если она больше не нужна. При работе с динамической памятью программа может позволить себе, например, узнать количество элементов массива на этапе выполнения.

2. Указатели

Указатель — это переменная, содержащая адрес другой переменной. Указатели очень широко используются в языке С. Это происходит отчасти потому, что иногда они дают единственную возможность выразить нужное действие, а отчасти потому, что они обычно ведут к более компактным и эффективным программам, чем те, которые могут быть получены другими способами.

Так как указатель содержит адрес объекта, это дает возможность «косвенного» доступа к этому объекту через указатель. Предположим, что **x — переменная**, например, типа **int**, а **px — указатель**, созданный неким еще не указанным способом. Унарная операция **&** выдает адрес объекта, так что оператор:

```
px = &x;
```

присваивает адрес x переменной px; говорят, что px «указывает» на x. Операция & применима только к переменным и элементам массива, конструкции вида:

```
&(x-1) и &3
```

являются незаконными. Нельзя также получить адрес регистровой переменной.

Унарная операция * рассматривает свой операнд как адрес конечной цели и обращается по этому адресу, чтобы извлечь содержимое. 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;
```

переменные, участвующие во всем этом необходимо описать:

```
int x, y;
int *px;
```

С описанием для **x** и **y** мы уже неоднократно встречались. Описание указателя:

```
int *px;
```

является новым и должно рассматриваться как мнемоническое; оно говорит, что комбинация ***px** имеет тип **int**. Это означает, что если **px** появляется в контексте ***px**, то это эквивалентно переменной типа **int**. Фактически синтаксис описания переменной имитирует синтаксис выражений, в которых эта переменная может появляться. Это замечание полезно во всех случаях, связанных со сложными описаниями. Например:

```
double  atof(), *dp;
```

говорит, что **atof()** и ***dp** имеют в выражениях значения типа **double**.Вы должны также заметить, что из этого описания следует, что указатель может указывать только на определенный вид объектов.

Указатели могут входить в выражения. Например, если **px** указывает на целое **x**, то *px может появляться в любом контексте, где может встретиться **x**. Например:

```
y = *px + 1;//присваивает y значение, на 1 большее значения x;
cout<< *px;//выводит текущее значение x;
d = sqrt((double) *px)//получает в d квадратный корень из x,
/*причем до передачи функции sqrt значение x преобразуется типу double */
```

В выражениях вида:

```
y = *px + 1;
```

унарные операции * и **&** связаны со своим операндом более крепко, чем арифметические операции, так что такое выражение берет то значение, на которое указывает **px**, прибавляет 1 и присваивает результат переменной **y**. Мы вскоре вернемся к тому, что может означать выражение:

```
y = *(px + 1);
```

Ссылки на указатели могут появляться и в левой части присваиваний. Если **px** указывает на **x**, то:

```
px = 0;
```

полагает **x** равным нулю, а:

```
*px += 1;
```

увеличивает его на единицу, как и выражение:

```
(*px) + 1;
```

Круглые скобки в последнем примере необходимы; если их опустить, то поскольку унарные операции, подобные * и **++**, выполняются справа налево, это выражение увеличит **px**, а не ту переменную, на которую он указывает.

И наконец, так как указатели являются переменными, то с ними можно обращаться, как и с остальными переменными. Если **py** — другой указатель на переменную типа **int**, то:

```
    py = px;
```

копирует содержимое px в py, в результате чего py указывает на то же, что и px.

3. Указатели и массивы

В языке С существует сильная взаимосвязь между указателями и массивами, настолько сильная, что указатели и массивы действительно следует рассматривать одновременно. Любую операцию, которую можно выполнить с помощью индексов массива, можно сделать и с помощью указателей. Вариант с указателями обычно оказывается более быстрым, но и несколько более трудным для непосредственного понимания, по крайней мере для начинающего. Описание:

```
int a[10];
```

определяет массив размера 10, т.е. набор из 10 последовательных объектов, называемых **a[0], a[1], ..., a[9]**. Запись **a[i]** соответствует элементу массива через i позиций от начала. Если **pa** — указатель целого, описанный как:

```
int *pa;
```

то присваивание:

```
pa = &a[0]
```

приводит к тому, что **pa** указывает на нулевой элемент массива **a**. Это означает, что **pa** содержит адрес элемента **a[0]**. Теперь присваивание:

```
x = *pa
```

будет копировать содержимое **a[0]** в **x**.
Если **pa** указывает на некоторый определенный элемент

массива **a**, то по определению **pa+1** указывает на следующий элемент, и вообще **pa–i** указывает на элемент, стоящий на **i** позиций до элемента, указываемого **pa**, а **pa+i** на элемент, стоящий на **i** позиций после. Таким образом, если **pa** указывает на **a[0]**, то:

```
*(pa+1)
```

ссылается на содержимое **a[1]**, **pa+i** — адрес **a[i]**, **a*(pa+i)** — содержимое **a[i]**.

Эти замечания справедливы независимо от типа переменных в массиве **a**. Суть определения «добавления 1 к указателю», а также его распространения на всю арифметику указателей, состоит в том, что приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель. Таким образом, **i** в **pa+i** перед прибавлением умножается на размер объектов, на которые указывает **pa**.

Очевидно существует очень тесное соответствие между индексацией и арифметикой указателей. В действительности компилятор преобразует ссылку на массив в указатель на начало массива. В результате этого имя массива является указательным выражением. Отсюда вытекает несколько весьма полезных следствий. Так как имя массива является синонимом местоположения его нулевого элемента, то присваивание:

```
pa = &a[0]
```

можно записать как **pa = a**.

Еще более удивительным, по крайней мере на первый взгляд, кажется тот факт, что ссылку на **a[i]** можно запи-

сать в виде ***(a+i)**. При анализировании выражения **a[i]** в языке С оно немедленно преобразуется к виду ***(a+i)**; эти две формы совершенно эквивалентны. Если применить операцию **&** к обеим частям такого соотношения эквивалентности, то мы получим, что **&a[i]** и **a+i** тоже идентичны: **a+i** — адрес **i-го** элемента от начала **a**. С другой стороны, если **pa** является указателем, то в выражениях его можно использовать с индексом: **pa[i]** идентично ***(pa+i)**. Короче, любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причем даже в одном и том же утверждении.

Имеется одно различие между именем массива и указателем, которое необходимо иметь в виду. Указатель является переменной, так что операции **pa=a** и **pa++** имеют смысл. Но имя массива является константой, а не переменной: конструкции типа **a=pa** или **a++**,или **p=&a** будут незаконными.

Когда имя массива передается функции, то на самом деле ей передается местоположение начала этого массива. Внутри вызванной функции такой аргумент является точно такой же переменной, как и любая другая, так что имя массива в качестве аргумента действительно является указателем, т.е. переменной, содержащей адрес.

```
/* показывает на экран массив m */
  void ShowElements(int *m, int size)
  {
      int n;
      for (n = 0; n < size; m++,n++)
              cout<<*m<<"\t";

  }
```

Операция увеличения m совершенно законна, поскольку эта переменная является указателем, **m++** никак не влияет на массив в обратившейся к **ShowElements** функции, а только увеличивает локальную для функции **ShowElements** копию адреса.

Описания формальных параметров в определении функции в виде:

```
int m[];
```

и

```
int *m;
```

совершенно эквивалентны; какой вид описания следует предпочесть, определяется в значительной степени тем, какие выражения будут использованы при написании функции. Если функции передается имя массива, то в зависимости от того, что удобнее, можно полагать, что функция оперирует либо с массивом, либо с указателем, и действовать далее соответствующим образом. Можно даже использовать оба вида операций, если это кажется уместным и ясным.

4. Указатели — аргументы функций. Передача аргументов по указателю

Так как в С передача аргументов функциям осуществляется «по значению», вызванная процедура не имеет непосредственной возможности изменить переменную из вызывающей программы. Что же делать, если вам действительно надо изменить аргумент? Например, программа сортировки захотела бы поменять два нарушающих порядок элемента с помощью функции с именем **swap**. Для этого недостаточно написать:

```
swap(a, b);
```

определив функцию swap при этом следующим образом:

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

Из-за вызова по значению **swap** не может воздействовать на аргументы **a** и **b** в вызывающей функции.

К счастью, все же имеется возможность получить желаемый эффект. Вызывающая программа передает указатели подлежащих изменению значений:

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

Так как операция **&** выдает адрес переменной, то **&a** является указателем на **a**. В самой **swap** аргументы описываются как указатели и доступ к фактическим операндам осуществляется через них:

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

      temp = *px;
      *px = *py;
      *py = temp;
  }
```

5. Домашнее задание

1. Дан массив целых чисел. Воспользовавшись указателями, поменяйте местами элементы массива с четными и нечетными индексами (т.е. те элементы массива, которые стоят на четных местах, поменяйте с элементами, которые стоят на нечетных местах).

2. Даны два массива, упорядоченных по возрастанию: A[n] и B[m]. Сформируйте массив C[n+m], состоящий из элементов массивов A и B, упорядоченный по возрастанию.

3. Даны два массива : A[n] и B[m]. Необходимо создать третий массив, в котором нужно собрать:

- Элементы обоих массивов;
- Общие элементы двух массивов;
- Элементы массива A, которые не включаются в B;
- Элементы массива B, которые не включаются в A;
- Элементы массивов A и B, которые не являются общими для них (то есть объединение результатов двух предыдущих вариантов).