Оглавление

1. Общие сведения о ссылках
2. Ссылочные параметры. Передача аргументов по ссылке
3. Ссылки в качестве результатов функций
4. Операторы выделения памяти new и delete
5. Домашнее задание

Общие сведения о ссылках

С этого урока мы начнем рассматривать другой механизм передачи параметров, в частности, с использованием **ссылок**.

Использование указателей в качестве альтернативного способа доступа к переменным таит в себе опасность - если был изменен адрес, хранящийся в указателе, то этот указатель больше не ссылается на нужное значение.

Язык С предлагает альтернативу для более безопасного доступа к переменным через указатели. Объявив ссылочную переменную, можно создать объект, который, как указатель, ссылается на другое значение, но, в отличие от указателя, постоянно привязан к этому значению. Таким образом, **ссылка на значение *всегда* ссылается на это значение**.

Ссылку можно объявить следующим образом:

```
<имя типа>& <имя ссылки> = <выражение>;

                или
<имя типа>& <имя ссылки>(<выражение>);
```

Раз ссылка является **другим именем уже существующего объекта**, то в качестве инициализирующего объекта должно выступать **имя некоторого объекта, уже расположенного в памяти**. Значением ссылки после выполнения соответствующего определения с инициализацией становится адрес этого объекта. Проиллюстрируем это на конкретном примере:

```
#include <iostream>

using namespace std;
void main()
{
   int ivar = 1234;    //Переменной присвоено значение.
   int *iptr = &ivar; //Указателю присвоен адрес ivar.
   int &iref = ivar;  //Ссылка ассоциирована с ivar.
   int *p = &iref;    //Указателю присвоен адрес iref.

   cout << "ivar = " << ivar << "\n";
   cout << "*iptr = " << *iptr <<"\n";
   cout << "iref = " << iref << "\n";
   cout << "*p = " << *p << "\n";
}
```

Результат работы программы:

```
ivar = 1234

*iptr = 1234
iref = 1234
```

```
*p = 1234
```

Комментарии к программе. Здесь объявляются четыре переменные. Переменная **ivar** инициализирована значением 1234. Указателю на целое ***iptr** присвоен адрес **ivar**. Переменная **iref** объявлена как ссылочная. Эта переменная в качестве своего значения принимает адрес расположения в памяти переменной **ivar**. Оператор:

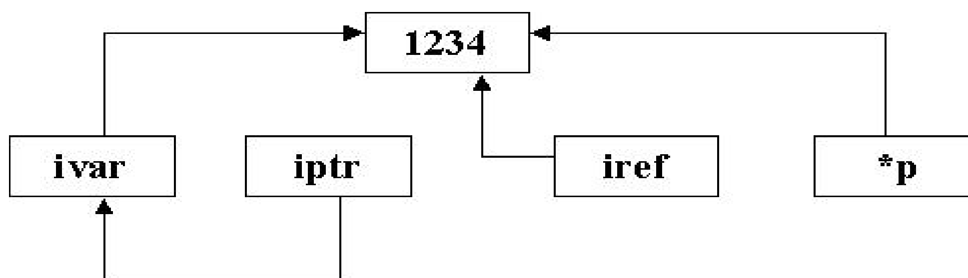
```
cout << "iref = " << iref << "\n";
```

выводит на экран значение переменной **ivar**. Это объясняется тем, что **iref** - ***ссылка*** на местоположение **ivar** в памяти.

Последнее объявление **int *p = &iref;** создает еще один указатель, которому присваивается адрес, хранящийся в **iref**. Строки:

```
int *iptr = &ivar;
          и
int *p = &iref;
```

дают одинаковый результат. В них создаются указатели, ссылающиеся на **ivar**. На рис.1 проиллюстрирована взаимосвязь переменных из приведенной программы:



При использовании ссылок следует помнить одно правило: **однажды инициализировав ссылку ей нельзя присвоить другое значение!** Все эти конструкции:

```
a) int iv = 3;        b) iref++;     c) iref = 4321;
   iref = iv;
```

приведут к изменению переменной **ivar**!

Замечания.

1.В отличие от указателей, которые могут быть объявлены не инициализированными или установлены в нуль **(NULL)**, ссылки **всегда ссылаются на объект**. Для ссылок ОБЯЗАТЕЛЬНА инициализация при создании и не существует аналога нулевого указателя.

2.Ссылки нельзя инициализировать в следующих случаях:

- при использовании в качестве параметров функции.
- при использовании в качестве типа возвращаемого значения функции.
- в объявлениях классов.

3.Не существует операторов, непосредственно производящих действия над ссылками!

Ссылочные параметры. Передача аргументов по ссылке

Ссылочные переменные используются достаточно редко: значительно удобнее использовать саму переменную, чем ссылку на нее. В качестве параметров функции ссылки имеют более широкое применение. Ссылки особенно полезны в функциях, возвращающих несколько объектов (значений). Для иллюстрации высказанного положения рассмотрим программу:

```
#include <iostream>

using namespace std;
//Обмен с использованием указателей.
void interchange_ptr (int *u,int *v)
{
      int temp=*u;
      *u = *v; *v = temp;
}
/* ------------- */
//Обмен с использованием ссылок.
void interchange_ref (int &u,int &v)
{
      int temp=u;
      u = v; v = temp;
}
/* ------------- */
void main ()
{
      int x=5,y=10;
      /* --------------------------------------- */
      cout << "Change with pointers:\n";
      cout << "x = " << x << "  y = " <<y <<"\n";
      interchange_ptr (&x,&y);
      cout << "x = " << x << "  y = " << y <<"\n";
          cout  <<  "-------------------------------------------"
<<"\n";
      cout << "Change with references:\n";
      cout << "x = " << x << "  y = " << y <<"\n";
```

```
        interchange_ref (x,y);
        cout << "x = " << x << "  y = " << y <<"\n";
    }
```

В функции **interchange_ptr()** параметры описаны как указатели. Поэтому в теле этой функции выполняется их разыменование, а при обращении к этой функции в качестве фактических переменных используются адреса **(&x,&y)** тех переменных, значения которых нужно поменять местами. В функции **interchange_ref()** параметрами являются ссылки. Ссылки обеспечивают доступ из тела функции к фактическим параметрам, в качестве которых используются обычные переменные, определенные в программе.
Ссылки и указатели в качестве параметров функций тесно связаны. Рассмотрим следующую небольшую функцию:

```
    void f(int *ip)
    {
            *ip = 12;
    }
```

Внутри этой функции осуществляется доступ к переданному аргументу, адрес которого хранится в указателе **ip**, с помощью следующего оператора:

```
    f(&ivar); //Передача адреса ivar.
```

Внутри функции выражение ***ip = 12;** присваивает 12 переменной **ivar**, адрес которой был передан в функцию **f()**. Теперь рассмотрим аналогичную функцию, использующую ссылочные параметры:

```
    void f(int &ir)
    {
       ir = 12;
    }
```

Указатель **ip** заменен ссылкой **ir**, которой присваивается значение 12. Выражение:

```
    f(ivar); //Передача ivar по ссылке.
```

присваивает значение ссылочному объекту: передает **ivar** по ссылке функции **f()**. Поскольку **ir** ссылается на **ivar**, то **ivar** присваивается значение 12.
Теперь, когда мы познакомились с ссылками, перейдём к следующему разделу и рассмотрим одно из их предназначений.

Ссылки в качестве результатов функций

Здесь мы рассмотрим использование ссылок в качестве результатов функций.

Функции могут возвращать ссылки на объекты при условии, что эти объекты **существуют, когда функция неактивна**. Таким образом, функции не могут возвращать ссылки на локальные автоматические переменные. Например, для функции, объявленной как:

```
double &rf(int p);
```

необходим аргумент целого типа, и она возвращает ссылку на объект **double**, предположительно объявленный где-то в другом месте.

Проиллюстрируем сказанное конкретными примерами.

Пример 1. Заполнение двумерного массива одинаковыми числами.

```
#include <iostream>

using namespace std;
int a[10][2];
void main ()
{
        int & rf(int index); //Прототип функции.
        int b;
        cout << "Fill array.\n";
        for (int i=0;i<10;i++)
        {
          cout << i+1 << " element: ";
          cin >> b;
          a[i][0] = b;
          rf(i) = b;
        }
        cout << "Show array.\n";
        cout << "1-st column   2-nd column" << "\n";
        for (int i=0;i<10;i++)
                cout << a[i][0] << "\t\t" << rf(i) << "\n";
}

int &rf(int index)
{
  return a[index][1]; //Возврат ссылки на элемент массива.
}
```

Здесь объявляется глобальный двумерный массив a, состоящий из целых чисел. В начале функции **main()** содержится прототип ссылочной функции **rf()**, которая возвращает ссылку на целое значение второго столбца массива a, которое однозначно идентифицируется параметром-индексом **index**. Так как функция **rf()** возвращает ссылку на целое значение, то имя функции может оказаться ***слева*** от оператора присваивания, что продемонстрировано в строке:

```
rf(i) = b;
```

Пример 2. Нахождение максимального элемента в массиве и замена его на нуль.

```
#include <iostream>

using namespace std;
//Функция определяет ссылку на элемент
//массива с максимальным значением.
int &rmax(int n, int d[])
{
  int im=0;
  for (int i=1; i<n; i++)
       im = d[im]>d[i]?im:i;
  return d[im];
}

void main ()
{
        int x[]={10, 20, 30, 14};
        int n=4;
        cout << "\nrmax(n,x) = " << rmax(n,x) << "\n";
        rmax(n,x) = 0;
        for (int i=0;i<n;i++)
          cout << "x[" << i << "]=" << x[i] << "  ";
        cout << "\n";
}
```

Результаты работы программы:

```
rmax (n,x) = 30

x[0]=10  x[1]=20  x[2]=0  x[3]=14
```

При выполнении строки:

```
cout << "\nrmax(n,x) = " << rmax(n,x) << "\n";
```

происходит первое обращение к функции **rmax()**, первый аргумент которой - количество элементов в массиве, а второй - сам массив. В результате возвращается ссылка на максимальный элемент массива, используя которую, это максимальное значение выводится на экран. При выполнении строки:

```
rmax(n,x) = 0;
```

снова осуществляется обращение к функции **rmax()**. Теперь уже по найденной ссылке максимальное значение меняется на 0.

Операторы выделения памяти new и delete

Операция выделения памяти new

С помощью вышеозначенной операции мы можем себе позволить выделять память динамически - т. е. на этапе выполнения программы.

Часто выражение, содержащее операцию new, имеет следующий вид:

```
указатель_на_тип_= new имя_типа (инициализатор)
```

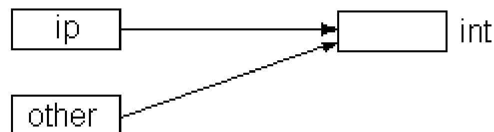
Инициализатор - это необязательное инициализирующее выражение, которое может использоваться для всех типов, кроме массивов.

При выполнении оператора

```
int *ip = new int;
```

создаются 2 объекта: динамический безымянный объект и указатель на него с именем ip, значением которого является адрес динамического объекта. Можно создать и другой указатель на тот же динамический объект:

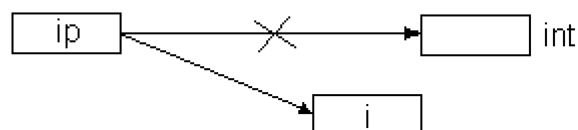
```
int *other=ip;
```



Если указателю ip присвоить другое значение, то можно потерять доступ к динамическому объекту:

```
int *ip=new (int);

int i=0;
ip=&i;
```



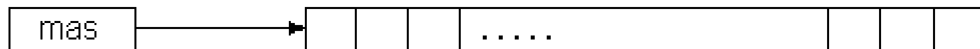
В результате динамический объект по-прежнему будет существовать, но обратиться к нему уже нельзя. Такие объекты называются мусором.

При выделении памяти объект можно инициализировать:

```
int *ip = new int(3);
```

Можно динамически распределить память и под массив:

```
double *mas = new double [50];
```



Далее с этой динамически выделенной памятью можно работать как с обычным массивом:

```
#include <iostream>

#include <stdlib.h>
#include <time.h>
using namespace std;
void main(){
        srand(time(NULL));
        int size;
        int * dar;
        // запрос размера массива с клавиатуры
        cout<<"Enter size:\n";
        cin>>size;
        /*выделение памяти под массив с количеством элементов size*/
        dar=new int [size];
        if(!dar){
               cout<<"Sorry, error!!!";
               exit(0);// функция организует выход из программы
        }
        // заполнение массива и показ на экран
        for(int i=0;i<size;i++){
                dar[i]=rand()%100;
                cout<<dar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

В случае успешного завершения операция new возвращает указатель со значением, отличным от нуля.

Результат операции, равный 0, т.е. нулевому указателю **NULL**, говорит о том, что не найден непрерывный свободный фрагмент памяти нужного размера.

Операция освобождения памяти delete

Операция **delete** освобождает для дальнейшего использования в программе участок памяти, ранее выделенной операцией **new**:

```
     delete ip; // Удаляет динамический объект типа int,
                   // если было ip = new int;
     delete [ ] mas; // удаляет динамический массив длиной 50, если
было
                       // double *mas = new double[50];
```

Совершенно безопасно применять операцию к указателю **NULL**. Результат же повторного применения операции delete к одному и тому же указателю не определен. Обычно происходит ошибка, приводящая к зацикливанию.

Чтобы избежать подобных ошибок, можно применять следующую конструкцию:

```
     int *ip=new int[500];

     . . .
     if (ip){
             delete ip; ip=NULL;
     }
     else
     {
             cout <<" память уже освобождена \n";
     }
```

В наш, вышеописанный пример, мы теперь можем добавить освобождение памяти.

```
     #include <iostream>

     #include <stdlib.h>
     #include <time.h>
     using namespace std;
     void main(){
             srand(time(NULL));
             int size;
             int * dar;
             // запрос размера массива с клавиатуры
             cout<<"Enter size:\n";
             cin>>size;
              //выделение памяти под массив с количеством элементов
size
             dar=new int [size];
             if(!dar){
                     cout<<"Sorry, error!!!";
                          exit(0);// функция организует выход из
программы
             }
             // заполнение массива и показ на экран
             for(int i=0;i<size;i++){
                     dar[i]=rand()%100;
                     cout<<dar[i]<<"\t";
             }
             cout<<"\n\n";
```

```
            // освобождение памяти
            delete[]dar;
     }
```

Домашнее задание

1. Через указатели на указатели посчитать сумму двух чисел и записать в третье.

2. Написать примитивный калькулятор, пользуясь только указателями.

3. Найти факториал числа, пользуясь только указателями.

4. Найти заданную степень числа, пользуясь только указателями.

5. Произвести, используя указатель на указатель проверку на нуль при делении.