STEP computer ACADEMY

PROGRAMMING C

Lesson No. 13

Programming in C

Contents

1. Reference overview

This lesson is devoted to another parameter passing mechanism specifically using the references.

Using pointers as an alternative way to access variables is fraught with danger — if the address stored in the pointer has been changed, this pointer no longer refers to the desired value.

C language provides an alternative more secure access to variables via pointers. Declaring a reference variable it is possible to create an object referring to a different value, that is similar to a pointer. However, unlike the pointer, this object is permanently attached to the value. **Thus, a reference to value always refers to this value.**

Reference can be declared as follows:

```
<type name >& <reference name> = <expression>;
                or
<type name >& <reference name>(<expression>);
```

Once a reference is another name of an existing object, the name of object available in memory should appear for the initiating object. After determining and initializing, the address of object becomes the reference value. Let us illustrate this with a particular example:

```
#include <iostream>
using namespace std;
void main()
{
   int ivar = 1234;   //A value was assigned for the variable.
   int *iptr = &ivar; //ivar address was assigned for the pointer.
   int &iref = ivar;  //A reference was associated with ivar.
   int *p = &iref;    //iref address was assigned for the pointer.

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

The result of program:

```
ivar = 1234
*iptr = 1234
iref = 1234
*p = 1234
```

Program comments. Four variables are declared. The **ivar** variable is initialized to 1234. A pointer to * **iptr** integer was assigned with ivar address. iref variable was declared as a reference. This variable takes the memory location of ivar variable in the capacity of its value. Operator:

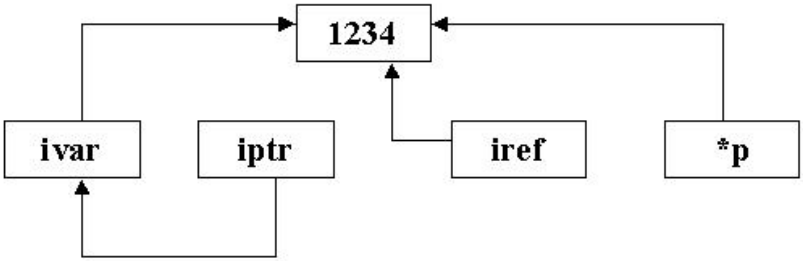
```
cout << "iref = " << iref << "\n";
```

displays the value of **ivar** variable. This is because **iref** is a reference to the memory location of ivar.

The latest declaration **int *p = &iref**; creates another pointer that is assigned with the address stored in iref. The lines:

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

yield a similar result. They contain pointers referencing to **ivar**. Figure 1



shows the connection of variables of the given program:

Using references, you should remember one rule: once a reference has been initialized, it is impossible to assign a different value to it! All of these constructs:

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

will result in changing the **ivar** variable!

Considerations.

1. Unlike the pointers, which may be declared as not initialized ones or set to NULL, references always refer to an object. When creating references it is MANDATORY to initialize them and there is no analogue of the null pointer.

2. References cannot be initialized in the following cases:

- When used as function parameters.
- When used as a type of return value of function.
- When declaring the classes.

3. There are no operators executing operations directly over the references!

2. Reference parameters. Passing arguments by reference

Reference variables are rarely used: it is much more convenient to use the variable itself than a reference to it. References are more widely applicable in the capacity of the parameters of functions. References are especially useful in functions returning multiple objects (values). In order to illustrate this statement, let's consider a program:

```
#include <iostream>
using namespace std;
//Change with pointers.
void interchange_ptr (int *u,int *v)
{
      int temp=*u;
      *u = *v; *v = temp;
}
/* ------------- */
//Change with references.
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";
}
```

Parameters of the **interchange_ptr ()** function are described as pointers. Therefore, their dereferencing is performed in this function. When referencing to the function, the **(&x, &y)** addresses of the variables, the values of which should be reversed, are used as actual variables. References are the parameters of **interchange_ref()** function. References provide the access from functions to actual parameters, which are regular variables defined in the program.

References and pointers as function parameters are closely related. Let's consider the following small function:

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

Access to the passed argument stored in the **ip** pointer is executed within this function by means of the following operator:

```
f(&ivar); //ivar address pass.
```

The expression ***ip = 12**; available within the function sets 12 to the **ivar** variable, whose address was passed to **f()** function. Now we are going to consider a similar function using reference parameters:

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

Ip pointer is replaced with **ir** reference that is set to 12. The expression:

```
f(ivar); //Passing ivar by reference.
```

assigns a value to the reference object: it passes **ivar** by the reference of the **f()** function. As **ir** refers to **ivar**, **ivar** is set to 12.

After considering the references, let's move on to the next section and consider one of their purposes.

3. References in the capacity of function's results

Here we are going to consider the use of references in the capacity of function's results.

Functions can return references to objects subject to the condition that these objects exist when the function is inactive. Thus, functions are not able to return references to automatic local variables. For example, a function declared as:

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

requires an argument of integer type. It returns a reference to **double** object presumably declared somewhere else.

Let us show this by means of particular examples.

Example 1. Filling a two-dimensional array with identical numbers.

```
#include <iostream>
using namespace std;
int a[10][2];
void main ()
{
        int & rf(int index); //Function prototype.
        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]; //Return reference to the array element.
}
```

Global two-dimensional array «a» consisting of integers is declared here. At the beginning of the **main()** function there is a reference **rf()** function prototype that returns a reference to an integer value of the second column in the «a» array unambiguously identified by the **index** parameter. Since the rf() function returns the reference to an integer value, the name of function may appear on the left of the assignment operator, as demonstrated in the line:

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

Example 2. Determining the maximum element in the array and replacing it with null.

```
#include <iostream>
using namespace std;
//This function defines a reference to the element
//of array with maximum value.
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";
}
```

Program output:

```
rmax (n,x) = 30
x[0]=10  x[1]=20  x[2]=0  x[3]=14
```

When executing the line:

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

rmax() function is called for the first time. Its first argument is the number of elements in the array, and the second one is the array itself. As a result, a reference to the maximum element of array is returned. When using it the maximum value is displayed. When executing the line:

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

rmax() function is called again. Now the maximum value is changed to 0 by the found reference.

4. New and delete define storage statements

Allocating the new storage

By means of the abovementioned operation we can allocate memory dynamically, i.e. at the stage of program execution.

Often an expression containing new operation is the following:

```
access_to_type_= new name_of type (initializer)
```

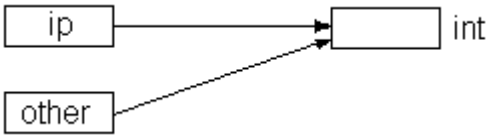
Initializer is an optional initializing expression that can be used for all types except the arrays.

When executing the statement

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

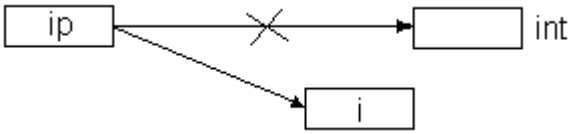
2 objects are created: a dynamic nameless object and a pointer to it named ip with the value of the dynamic object address. You can create another pointer to the same dynamic object:

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



If a different value was assigned to ip pointer, you can lose access to the dynamic object:

```
int *ip=new (int);
int i=0;
ip=&i;
```



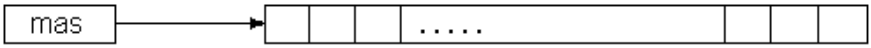
As a result, the dynamic object will survive, but it will be impossible to refer to it anymore. Such objects are called garbage.

When allocating memory, an object can be initialized:

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

It is also possible to dynamically allocate memory for the array:

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



Then this dynamically allocated memory may be operated as a common array:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main(){
        srand(time(NULL));
        int size;
        int * dar;
        // keyboard inquiry of array size
        cout<<"Enter size:\n";
        cin>>size;
      /*allocating memory for an array with size number of elements*/
        dar=new int [size];
        if(!dar){
               cout<<"Sorry, error!!!";
               exit(0);// function organizes the program exit
        }
        // filling and displaying the array
        for(int i=0;i<size;i++){
                dar[i]=rand()%100;
                cout<<dar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

Upon successful completion, the new operation returns a pointer to a value other than null.

The result of operation is equal to 0, i.e. **NULL** pointer, is a reflection of the fact that a continuous available fragment of memory of the required size was not found.

Delete memory deallocation operation

The fragment of memory previously allocated by the **new** operation is deallocated by the **delete** operation for further use in the program:

```
delete ip; // Deletes a dynamic object of int type,
             // if ip = new int;
delete [ ] mas; // deletes a dynamic array of the length of 50, if
                  // double *mas = new double[50];
```

It is safe to apply the operation to the **NULL** pointer. The result of reapplying the delete operation to the same index is not defined. Usually there is an error leading to the sink state.

In order to avoid such errors, you can use the following construct:

```
int *ip=new int[500];
. . .
if (ip){
        delete ip; ip=NULL;
}
else
{
        cout <<" memory is already released  \n";
}
```

Now we can add the memory deallocation to the above example.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
void main(){
        srand(time(NULL));
        int size;
        int * dar;
        // keyboard inquiry of array size
        cout<<"Enter size:\n";
        cin>>size;
        //allocating memory for an array with size number of elements
        dar=new int [size];
        if(!dar){
                cout<<"Sorry, error!!!";
                exit(0);// function organizes the program exit
        }
        // filling and displaying the array
        for(int i=0;i<size;i++){
                dar[i]=rand()%100;
                cout<<dar[i]<<"\t";
        }
        cout<<"\n\n";
        // memory deallocation
        delete[]dar;
}
```

5. Home assignment

1. Calculate the sum of two numbers and write the third one using the pointers to pointers.
2. Develop a primitive calculator using pointers only.
3. Find a factorial of number using pointers only.
4. Find a predetermined power of number using pointers only.
5. Perform checking for zero using a pointer to a pointer.