Discussions on References and Returning References in C++ Programming Language

Mu Jingqin, Li Xiaohui, Jiang Haiyang, Zhang Jun, Du Ruiqing
Department of Computer Science
Tangshan Normal University
Tangshan, Hebei Province, P.R. China
e-mail: mujingqin@gmail.com, lxhami@gmail.com, jianghy456@163.com, 21551470@qq.com, durq@163.com

Abstract—References are widely used in C++ programming language. A reference is a value that enables a program to indirectly access a particular data item in the computer's memory or in some other storage device. References are difficult for programmers to understand and to master. The purpose of this paper is to illustrate the common knowledge of references and returning references, and discuss how to understand them soundly and use them properly. One way to this problem is to regard these references as some certain aliases. Several examples, such as stand-along reference and returning reference, are presented to discuss the use of references and their "aliases", and a typical example is presented to illustrate how to analyze programs about the special returning reference with "aliases". At the end of this paper, a conclusion is made based on the above discussions that one of the ways mastering references and returning references is to regard them as aliases for variables, objects, and functions.

Keywords-references; returning references; memory space; C++ programming language; function

I. INTRODUCTION

C++ is an objected-oriented programming language. It is regarded as a "middle-level" language, as it comprises a combination of both high-level and low-level language features. It was developed by Bjarne Stroustrup starting in 1979 at Bell Labs as an enhancement to the C programming language and originally named C with Classes. It was renamed C++ in 1983.

As one of the most popular programming languages ever created, C++ is widely used in the software industry. The returning reference is an important part in C++, because it is able to improve execution efficiency of programs. However, it is difficult for programmers to understand and master, because it is much different with the returning variables which can be understood in the way of the common sense. The aim of this paper is to illustrate the rules of returning references and the application of it by presenting several typical examples.

II. REFERENCE

Before discussing returning references, we should know about some knowledge of reference in C++ programming language.

A reference is a value that enables a program to indirectly access a particular data item, such as a variable or a record, in the computer's memory or in some other storage device. The reference is said to refer to the data item, and accessing that data is called dereferencing the reference.

A reference is distinct from the data itself. Typically, a reference is the physical address of where the data is stored in memory or in the storage device. For this reason, a reference is often called a pointer or address, and is said to point to the data. However a reference may also be the offset between the datum's address and some fixed "base" address, or an index into an array.

References are widely used in programming, especially to efficiently pass large or mutable data as arguments to procedures, or to share such data among various uses. In particular, a reference may point to a variable or record that contains references to other data. This idea is the basis of indirect addressing and of many linked data structures, such as linked lists.

Two ways to pass arguments to functions or return values to calling functions in C++ programming languages are pass-by-value and pass-by-reference. When an argument is passed by value, a copy of the argument's value is made and passed on the function call stack to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

The reference parameter is one of two means C++ provides for performing pass-by-reference. With pass-by-reference, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses to do so. Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data. But, pass-by-reference can weaken security, because the called function can corrupt the caller's data. So using the reference properly and avoiding not corrupting the caller's data are the necessary requirements for programmers.

III. Examples of Reference

The essence of a reference is an implicit pointer. One of the ways of mastering it is to regard it as aliases for other variables, objects, and functions. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference.

A. *Stand-along references*

When a reference is declared as a variable, it is a stand-along reference, also called an independent reference. All independent references must be initialized when they are created.

Example (1)

```
#include <iostream>
using namespace std;

void main()
{
    int a=8;
    int &ra=a;

    ra=8;
    cout<<"a="<<a<<",ra="<<ra<<endl;

    a=88;
    cout<<"a="<<a<<",ra="<<ra<<endl;
}
```

In Example (1), we declared a reference *ra* that refers to the integer variable *a*, then ra is the alias of a. When *ra*'s value is being changed to 8, *a*'s value is changed to 8 too. When *a*'s value is being changed to 88, *ra*'s value is changed to 88 too. The out come of Example (1) is:

Outcome (1)

```
a=8,ra=8
a=88,ra=88
```

A reference is able to refer to other type variables or an object. It is also the alias of the referred data.

B. *Reference parameters*

The most important use for a reference is acting as parameters that automatically use call-by-reference to pass parameters. Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. In that way, it will take a considerable amount of time and memory space when pass large objects. Call-by-reference only passes the address of the argument to the function. It saves the time and memory space. By default, C++ uses call-by-value, while it provides two ways to achieve call-by-reference parameter passing. First, we can explicitly pass a pointer to the argument. Second, we can use a reference parameter. For most circumstances the best way is to use a reference parameter.

Example (2)

```
class Point
{public:
   Point(int vx=0, int vy=0):x(vx),y(vy){}
   void print(){cout<<"x="<<x<<",y="<<y<<endl;}
   friend Point add(Point &pleft, Point &pright);
private:  int x, y;
};
Point add(Point &pleft, Point &pright)
{     Point pt;
      pt.x=pleft.x+pright.x; pt.y=pleft.y+pright.y;
      return pt;}

void main()
{     Point p1(1,1), p2(2,2), p3;
      p3=add(p1,p2);   p3.print();}
```

The function *add()* is a friend function of class *Point*. It is able to access to private data of class *Point* from outside of the class. Its function is to add two objects' data to construct a new object. The function has two reference parameters, *pleft* and *pright*. When the function is called to execute, *pleft* and *pright* are created and initialized with the addresses of *p1* and *p2*. As a result, they are alias as *p1* and *p2*. The calculation of *pleft* and *pright* in function *add()* can be regarded as the calculation of *p1* and *p2.* So, the outcome of Example (2) is:

Outcome (3)

```
x=3, y=3
```

C. *Returning references*

A function may return a reference. This has the rather startling effect of allowing a function to be used on the left side of an assignment statement! But this can be dangerous. When returning a reference to a variable declared in the called function, the variable should be declared static within that function. Otherwise, the reference refers to an automatic variable that is discarded when the function terminates; such a variable is said to be "undefined," and the program's behavior is unpredictable. References to undefined variables are called dangling references.

The returning references should be static and global variables that can not be discarded at the end of function execution.

Example (3)

```
#include <iostream>
using namespace std;
#include <string>

char city[20]="Beijing";

void main()
{
    char &replace(int i);

    cout<<"Source city:"<<city<<endl;
    if(strlen(city)>=3)
    {       replace(0)='N';    replace(1)='a';
            replace(2)='n';
    }
    cout<<"Target city:"<<city<<endl;
}

char &replace(int i) {   return city[i];  }
```

We declared a function *replace()* it has a returning *char* reference. Its return value is *city[i]*. The function can be regarded as an alias of *city[i]* that has a parameter *i*. In function *main()*, *replace(0)='N'* can be changed to *city[0]='N'* during the execution, and etc.. So the outcome of Example (3) is:

Outcome (3)

```
Source city:Beijing
Target city:Nanjing
```

Although references are widely used in C++, there are a number of restrictions that apply to references: (1) references did not refer to another reference, because we cannot obtain the address of a reference; (2) references cannot refer to arrays of references. (3) a pointer cannot point to a reference; (4) a reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value; (5) null references are prohibited.

IV. A Typical Example of Returning References

Example (4)

```
#include <iostream>
using namespace std;
#include <string>
#include <iomanip>

struct VecElem{ char *index;     int value;};

class AssocVec
{
private:
        VecElem *elems;
        int dim, used;
public:
        AssocVec(int dim);
        ~AssocVec();
        int &operator[](char *idx);
        void print();
};

AssocVec::AssocVec(int dim)
{       AssocVec::dim=dim; used=0;
       elems=new VecElem[dim];}

AssocVec::~AssocVec()
{       for(int i=0;i<used;i++){delete elems[i].index;}
        delete elems;}

int &AssocVec::operator [](char *idx)
{       for(int i=0;i<used;i++)
        {if(strcmp(idx,elems[i].index)==0)
          return elems[i].value;}
        if(used<dim &&
      (elems[used].index=new char [strlen(idx)+1])!=0)
        {           strcpy(elems[used].index,idx);
                    elems[used].value=used+1;
                    return elems[used++].value;
        }
        static int dummy=0;
        return dummy;
}

void AssocVec::print()
{       cout<<setiosflags(ios::left);
        for(int i=0;i<used;i++)
        {cout<<setw(10)<<elems[i].index;
          cout<<setw(6)<<elems[i].value<<endl;}
}
void main()
{
        AssocVec count(5);
        count["apple"]=5;
        count["orange"]=10;
        count["fruit"]=count["apple"]+count["orange"];
        count.print();
}
```

A returning reference declared in Example (4) :

*int &operator[](char *idx);*

Function *operator[]()* is a special function. It is a operator overloading function and has a returning reference. For this reason, many programmers are confused with this Example. Let us illustrate the execution of this program step by step from function *main()*.

A. *AssocVec count(5);*

It declares an object *count* with a parameter of 5 passed from *dim* and the constructor of *AssocVec* is called automatically by the system. Then the program set values to data members *dim* and *used* all with *0*. After that, the memory space of *5* times *VecElem* construct type is assigned to the program and set its first memory address to *elems* (displayed in Fig .1 as *&elems[0]*). The values of *index* and *value* are all random. Fig .1 shows the situation in memory after these operations.

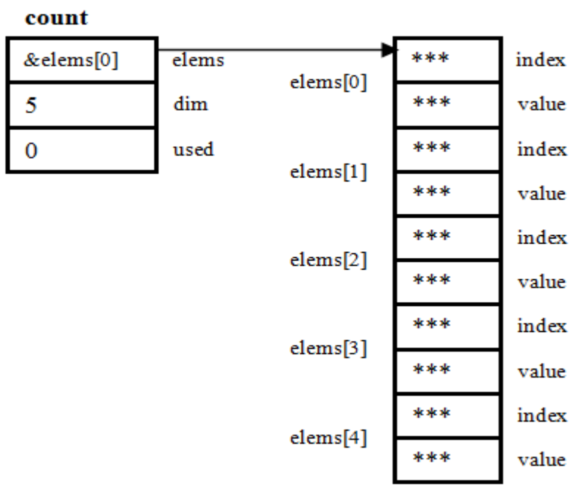

Figure 1. Schematic diagram of object count

B. count["apple"]=5;

The class *AssocVec* have overloaded operator *[]*, so when the execution system meets *count["apple"]*, it calls the overloaded operator *[]* in declaration with a parameter of *"apple"* passed by *idx*. And then the *idx* is compared to all *index* in array *elems[i]* (*i<used*). Now the *used* an *i* are all 0, so the compare is not carried out. Then compare *used* and *dim*. Now *dim* is 5, then *used<dim* gets true. Afterwards, assigned some space for *elems[0].index* and copy the values of *idx* to it, and *elms[0].value* is set to *1* (*used+1*). And then return *elems[0].value*, and *used* became *1* by *used++*.

Since the return type is a conference, the *count["apple"]* is able to regard as the alias of *elems[0].value*, then the truth of *count["apple"]=5* is *elems[0]=5*. Fig .2 shows the results after the execution of *count["apple"]=5*.

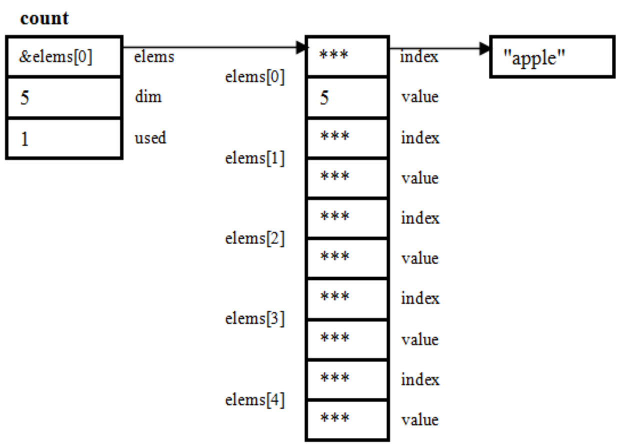

Figure 2. Schematic diagram of memory after *count["apple"]=5*

C. count["orange"]=10;

The execution of this sentence is the same like *count["apple"]=5*. It assigns new memory space to string *"orange"* and copy its address to *elems[1].index*. And set *elems[1].value* to *10*, *used* to *2*. Fig .3 shows the result.

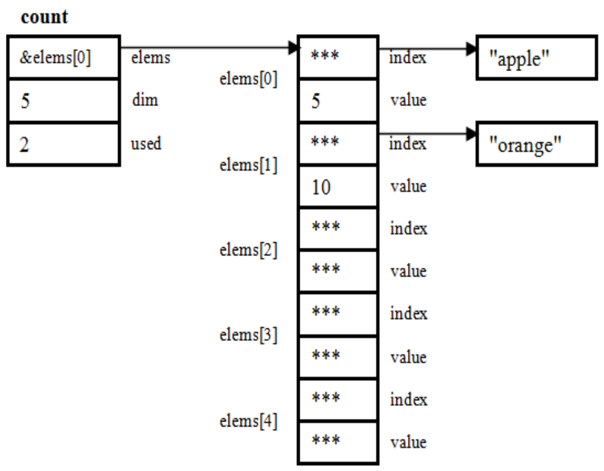

Figure 3. Schematic diagram of memory after *count["apple"]=5*

D. count["fruit"]=count["apple"]+count["orange"];

Firstly, the values of *count["apple"]* and *count["orange"]* are gotten. The strings of *"apple"* and *"orange"* are compared with *elems[i].index* (*i<2*) , if they are equal, the corresponding *elems[i].value* is returned to calling function *main()*. So *count["apple"]+count["orange"]* is equivalent to *elems[0].value+ elems[1].value.* The result of it is 15.

Secondly, *count["fruit"]=15*. Its function is same as step *B* and step *C*. It set *elems[2].index* with the address of string *"fruit", elems[2].value* with 15. Fig .4 shows the result.

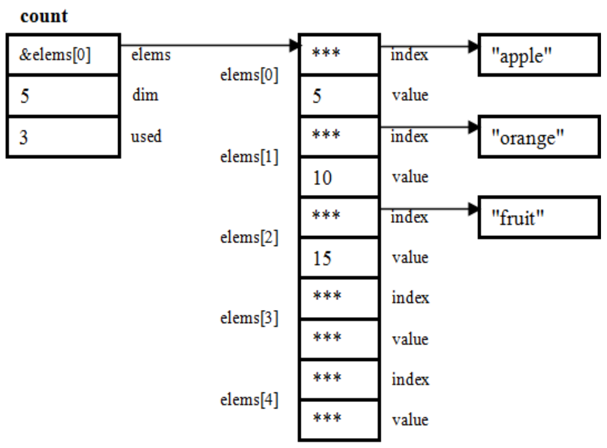

Figure 4. Schematic diagram of memory after *count["fruit"]=count["apple"]+count["orange"]*

E. count.print();

It finds out all the values of *elems[i].index*'s strings and *elems[i].value* and output them on the screen. So the outcome is:

```
Outcome (4)
apple    5
orange   10
fruit    15
```

V. Conclusion

References are widely used in programming. The essence of a reference is an implicit pointer. There are three ways that a reference can be used: as a function parameter, as a function return value, or as a stand-alone reference. References have several restrictions in their application in programs. Reference and returning references are difficult to understand and master. One of the ways of mastering them is to regard them as aliases for variables, objects, and functions.

Acknowledgment

The paper is supported by the Computer Application Technology Platform, one of the Discipline Construction Platforms of Tangshan Normal University, and the research was sponsored by Doctorial Scientific Fund of Tangshan Normal University (Project No. 2014A08). The author also gratefully acknowledges the helpful comments and suggestions of the reviewers, which have improved the presentation.

References

[1] H. Schildt, "C++: The Complete Reference, Fourth Edition," McGraw-Hill, 2003.

[2] S. B. Lippman, J. Lajoie, B. E. Moo, "C++ Primer, Fifth Edition," Addison Wesley Professional, 2013.

[3] H. M. Deitel, P. J. Deitel, "C++ How to Program, Fifth Edition," Prentice Hall, 2005.

[4] ISO/IEC 14882, "Programming languages — C++, Second Edition ," American National Standards Institute, 2003.

[5] Bruce Eckel, "Thinking in C++," Prentice Hall, 2007.

[6] Andrew Koenig, Barbara E.Moo, "Ruminations on C++," Addison-Wesley, 2013.

[7] Du Ruiqing, Li Xiaohui, "Discussions on the teaching of C++ programming language in colleges," Proceedings-2010 International Conference on Artificial Intelligence and Education, (ICAIE 2010), pp. 579-581, doi: 10.1109/ICAIE.2010.5641458.

[8] Li Xiaohui, Du Ruiqing, "The use of the this pointer in helping students to understand C++ programs soundly," ICEIT 2010 - 2010 International Conference on Educational and Information Technology, Proceedings, v2, pp. V2381-V2384, doi: 10.1109/ICEIT.2010.5607597.

[9] Luo Fafen, Du Ruiqing, "Discussion on copy constructor in C++ programming language," Proceedings of SPIE - The International Society for Optical Engineering, v 8350, 2012.

[10] D. Ryan Stephens, Christopher Diggins, Jonathan Turkanis, Jeff Cogswell, "C++ Cookbook," O'Reilly Media, 2005.