Understanding and Low Level Implementation Basic OOP Constructions

Valery Sklyarov

Resumo **- A experiência no ensino da programação orientada a objectos mostra que os alunos usam algumas construções básicas incorrectamente, tendo muitas dificuldades no uso de ponteiros e referências, objectos constantes e estáticos, funções virtuais, etc. Este artigo apresenta explicações detalhadas dessas construções em C++ e na linguagem *assembly*, permitindo-lhes perceberem não só as construções, mas também a sua eficiência. O artigo recomenda ainda algumas regras para construir uma *nice class*.**

Abstract **- Experience in the teaching of object-oriented programming shows that there are some basic constructions which are frequently used incorrectly by students. For example, many students have difficulties with the use of references and pointers, const and static objects, and virtual functions. This paper presents a detailed explanations of such constructions in C++, with examples of their implementation in assembly language as generated by the compiler. This will enable students to not only understand how these constructions work, but also to get a feeling for their efficiency. There are also some recommended rules for good class design.**

I. Introduction

The goal of this paper is to show how the object-oriented style of programming is used and how the efficiency of C++ code can be improved. It is aimed at students since previous experience shows that they have particular difficulties with the design of C++ programs and with the understanding of some of the basic constructions in object-oriented programming.

The paper presents the following topics:

- Guidance on how to use various C++ constructions correctly;
- Detailed explanations of the constructions that are the most difficult for students to understand with a discussion of the more common errors that are made;

II. Values, pointers and references

C++ provides both **direct** and **indirect** access to objects. An object is a **value** that is accessed directly through an object name, and indirectly through pointers and references. The distinction between a pointer and a reference is that the programmer may use a reference as an ordinary object, even though it accesses the object indirectly, whereas a pointer must be de-referenced in order to access the object. Even though pointers and references differ in the way they are used, the compiler will build the same code for both references and pointers. A reference is an **implicit pointer** to a value whereas a pointer requires an **explicit** definition. Consider an example:

```
void main(void)
{   int i=3;        // mov word ptr[bp-2],3
    int &j=i;       // lea ax,[bp-2]
                    // mov [bp-4],ss
                    // mov [bp-6],ax
    j=2;            // les bx,[bp-6]
                    // mov es:word ptr [bx],2
// ..........
}
```

The comments show possible assembly language code created by the compiler (I used TASM compiler for 16 bits Intel microprocessors). After the assignment (j=2), the object **i** also has the value **2** (see figure 1). As you can see from Figure 1, the reference j can be considered to be an **implicit pointer**. Indeed, for the following program the compiler will build the same assembly language code:

```
void main(void)
{ int i=3;
  int *j=&i;// j is an explicit pointer to i
  *j=2;     // explicit use of the pointer j
// ..........
}
```

Since a reference is actually an implicit pointer to a value, it sometimes may produce unexpected results. Consider the following example:

```
#include <iostream.h>
int F(int& ri)
{   ++ri;
    return ri;     };
void main(void)
{   int m1=1;
    cout << F(m1) << endl;  // The result: 2
    cout << m1 << endl;     // The result: 2
}
```

In the example, the reference to the object **m1** is passed to the function **F**. The function **F** changes the value of **m1** indirectly through its reference.

The next example shows another potential error:

```
#include <iostream.h>
int* F(int& ri)
{   ++ri;
    return &ri;    };
void main(void)
{   int m1=1,*m2;
    m2=F(m1);
    *m2=10;
    cout << m1 << endl; // The result: 10
}
```

The pointer **m2** stores the address of the value returned from F. As a result it points to **m1**. Therefore the statement ***m2=10** assigns the value **10** to the variable **m1**. If the function **F** were defined as **int* F(int ri)** we would not get such an error. The use of references allows you to change data even in the private section of a class (see, for example, [1]).

As a consequence, it is generally better to **avoid** the use of references with the basic data types. However they can be very efficient when used with user-defined data types.

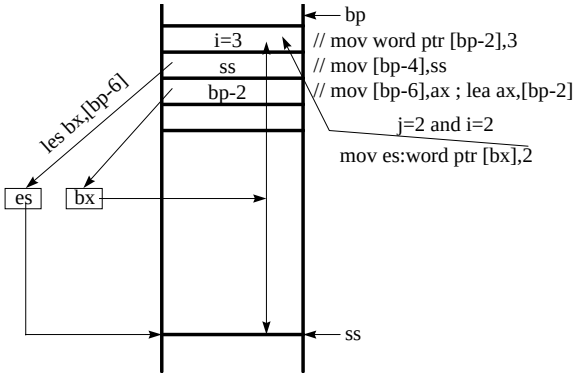

Figure 1. Access to the object i vie reference

Consider the following assignment:

```
X = Y;
```

This assignment is correct if the expression on the left side produces an **lvalue** (a left value which is an address) and the expression on the right side produces an **rvalue** (a right value which is an acceptable value for the specified type). As a result of this statement, the value on the right side is copied into the address location on the left. Since a reference is considered to be an **lvalue,** it can appear on the *left side* of the assignment statement. Passing a function argument by value means that a copy of its value (the rvalue) is placed on the stack. The function has no direct access to the value in the calling function. Passing an argument by reference copies the lvalue itself onto the stack, so the address of the argument is passed to the function allowing the function direct access to the value in the calling function.

A reference can be returned by a function. Consider the following two examples.

```
int& F(int& i)      // push bp
                    // mov bp,sp
{ return i; };      //** mov dx,[bp+8]
                    //** mov ax,[bp+6]
                    // pop bp
                    // retf
void main(void)
{   int mi1=10,mi2;// mov word ptr [bp-2],0Ah
                          // here ss=bp-4
    mi2=F(mi1);     // push ss
                    // lea ax,[bp-2]
                    // push ax
                    // push cs
                    // call F
                    // add sp,4
                    //** mov bx,ax
                    //** mov es,dx
                    //** mov ax,es:[bx]
                    //** mov [bp-4],ax
}
int F(int& i)
{ return i;};       //** les bx,[bp+6]
                    //** mov ax,es:[bx]
void main(void)
{   int mi1=10,mi2;
    mi2=F(mi1); }  //** mov [bp-4],ax
```

In the first example the function **F** returns an lvalue (as before, the comments show possible assembly language code created by the compiler). Because it returns an lvalue, **F** can be used on the left side of the assignment statement. In the second example the function **F** returns an rvalue. This means that **F** cannot be used on the left side of the assignment statement. Additional explanations are given in figure 2. The assembly language instructions that are different in the second example are marked with //**.

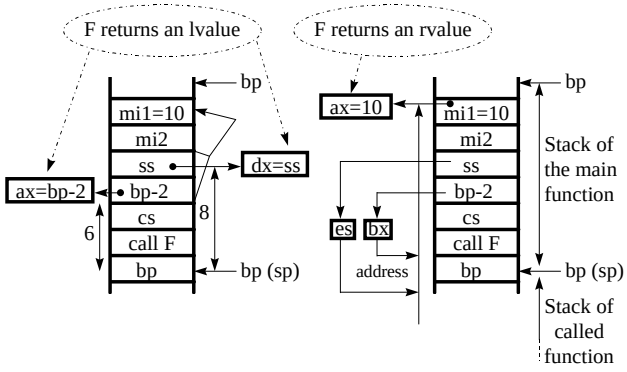

Figure 2. Lvalue and rvalue returned from the function

Using references as return values enables us to design better code. Suppose we want to overload the subscript operator []. Consider the following program:

```
#include <iostream.h>
#include <string.h>
class X {
    char *s;
public:
    X(char* S)
    {   s = new char[strlen(S)+1];
```

```
        strcpy(s,S);              }
    char& operator[](int i) { return s[i]; }
    void display(void) { cout << s << endl; }
        };
void main(void)
{   X x="01234";
    x.display();   // The result: 01234
    x[2]='#';  // operator[] function call
    x.display();   // The result: 01#34
    cout<<x[2];// operator[] function call
}   // The result: #
```

In the example above, the operator[] function can indirectly change a private string of the class X, pointed to by s (and we do want to allow such changes to be made). Trying to use alternative approaches leads us to worse code. Indeed consider the following function:

```
char operator[](int i);
```

In this case the compiler gives an error for the statement x[2]='#'; (lvalue required).

Returning a pointer to value by declaring the function as:

```
char* operator[](int i);
```

is worse because we have to use expressions such as *x[2]='#'; which are much less readable.

Because a reference to the result is returned from the function, the return value cannot be an **automatic** variable. Since the function can be used more than once in an expression, the result cannot be a **static local** variable.

Consider the following program:

```
class X {
    int *i;
public:
    X(void) {      i = new int[2];
                   i[0]=0; i[1]=1;  }
    ~X(void) { delete[] i;  }
    void display()
    { cout << i[0] << '\t' << i[1] << endl; }
};
void f(X x)        // f(X& x) - OK
{   int *j = new int[2];
    // do something
}
void main(void)
{   X o;
    o.display();
    f(o);          // destructor invoked
    o.display();
}                 // destructor invoked
```

The function **f** takes a value of type **X**. If even the function does nothing we will still have an error related to dynamic memory deallocation (see figure 3). This is probably the error that occurs most frequently.

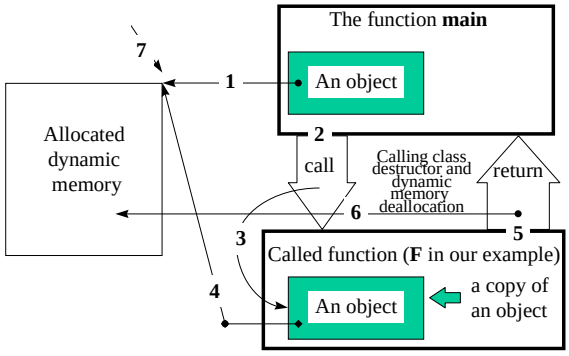

Figure 3. An error of unexpected dynamic memory deallocation

Let us examine the following function call:

```
f(o);
```

The parameter **o** of the function **f** is a local (automatic) object in the function body. An automatic object is created each time its declaration is **encountered** in the execution of the program, and destroyed each time the block in which it is declared is **left**. As a result, after the termination of the function **f,** the destructor for the object **o** will be called. Figure 3 shows the whole sequence of steps in this process which are the following:

1) in the definition **X o;** a new object **o** is constructed. The object constructor allocates memory dynamically using the **new** operator;
2) the function **f** is called (see the statement **f(o)**);
3) since the argument to the function **f**, the object **o,** is passed by value, this value is copied from the function **main** onto the stack for the function **f**;
4) the object copy contains a pointer to the same memory allocated for the original object;
5) the function **f** is terminated (the block which contains the copied object is left);
6) the destructor for the copy of the object **o** is called, and it **deallocates** the dynamically allocated memory for the object;
7) now the pointer in the original object **o** points to nowhere since its memory has been discarded.

You can see that the constructor of **o** was called once (in the function **main**), whereas the destructor of **o** was called twice (both in the function **main** and in the function **f**). Besides the allocated memory was unexpectedly destroyed. However if the function **f** would be declared as **f(X&)** then the program would run properly. Suppose you want to leave the previous declaration (**f(X x)**). We can do it but we have to eliminate the error in the design of the class **X**. Classes should not be designed so that if objects are passed by value you get an error. This is due to the default copy constructor (see section VI) being called which does memberwise copy to produce a copy object. The class **X** should define a copy constructor:

```
X::X(const X& o)
{   i=new int[2];
    i[0] = o.i[0]; i[1] = o.i[1]; }
```

Now we can use arguments by value or by reference. It is a basic rule that any class allocates memory in the constructor should implement a copy constructor as well as a destructor. It should also implement the assignment operator (see section VI) since the default assignment operator produces the same problem as the default copy constructor.

Additional information about these topics can be found in [2].

III. USE OF CONST

Consider a function **F** which has the following prototype:

```
X F(const X& r_o);
```

Here **X** is a class name. From an abstract point of view this declaration promises not to change the abstract value of the object referenced by **r_o** [3].

A member function of a class can be declared such that it will only **read** data from the object to which it belongs, but not change it. The declaration not to change the object is indicated by a **const** suffix to the argument list, for instance:

```
class X {       // .........
    void F(void) const;       };
```

Now any attempt to change **X** in the function **F** is illegal. A **const** member function can be called for a **const** object. A **non const** member function cannot be called for a **const** object. The type of the pointer **this** in a **const** member function of class **X** is **const X *const** [4]. This means that you cannot change the value of an object without an **explicit cast** (see the example below). Changes can be also made by using relatively new features of C++ such as the **mutable** declaration and **cast** away **const** capability [3].

Consider an example.

```
class X {
    int i;
public:
     X(int I) : i(I) {}
  void display(void) const;
// explicit cast
    void exc(int e) const {((X*)this)->i=e;}
    void f(void)  { i=5;}
// cannot modify a const object
// void ff() const {i=100;}
    };
    void X::display(void) const
    { cout << i << endl; }
void main(void)
{  X x=10;
   X const y=20;
// non-const function called for const object
// y.f();
 // ........
}
```

This program will show some error messages generated by the compiler.

IV. USE OF STATIC

The memory used for a C++ program is divided into three parts:

- the **static part** which contains the program code and static data;
- the **stack** that holds automatic variables and function arguments;
- the **free store** (the heap) which is available for dynamic allocation and deallocation.

There are two meanings of the keyword static [4]:

- as in **statically allocated,** which is opposed to on the *stack* or on the *free store* (on the heap);
- as in **with restricted visibility,** which is opposed to with *external linkage*.

In C++ class members can be declared as **static**. Static members are considered to be a property of a class and they are **shared** among all objects of a class. A static member declaration is only a **declaration** and the member (even a private member) must have a **definition** somewhere in the program, for example:

```
int X::i=0;
```

The name of the static member is a **fully qualified name** (for example, X::i). Static data members have both the meanings considered above. For member functions, **static** has only the second meaning.

The example below demonstrates the use of static members.

```
#include <iostream.h>
class X {
    static int i;
public:
    void ff() { ++i; }
    static void f()
    { cout << "i=" << i << endl; }
        };
int X::i=0;        // definition of i
void main(void)
{   X x1,x2;
    x1.ff();
    x2.ff();
    x1.ff();
    X::f(); }  // The result: i=3
```

Here the function **X::f()** was called for the class **X**, but not for a particular object of the class **X**. A static member function does not have a **this** pointer and cannot be **virtual**. Such a function can access non static members of its class **explicitly** by using class member access operators (**.** and **->**). It is not allowed to have static and non static member functions with the same name and the same argument types. Constructors for global static objects are called in the order of the object declarations. Destructors are called in the opposite order. Constructors for local static objects are called the first time the object definition occurs. Any static object is constructed **once** according to the rules above, and destroyed at the termination of the program.

By calling constructors and destructors for static objects explicitly, we can **initialise** and **cleanup** data in libraries (before execution and after termination of the function **main**).

Suppose the C++ project file my.ide includes the files my1.cpp and my2.cpp which contain the following code:

```
#include <iostream.h>     // my1.cpp
class Y {
public:
  Y() { cout << "\nY constructor\n"; }
  ~Y() { cout << "Y destructor\n"; }
       };
class X {
  static int i;
public:
  X();
  ~X();
       };
static X x;        // object x definition
int X::i=0;        // definition of i
X::X()
{   cout << i << '\t';
    if(i++ == 0) cout << "initialise\n";  }
X::~X()
{   cout << i << '\t';
    if(--i == 0) cout << "cleanup\n";  }
void main(void)
{   Y y;    }

class X {          // my2.cpp
    static int i;
public:
    X();
    ~X();
        };
static X x;
```

The program produces the following results:

```
0   initialise
1
Y constructor
Y destructor
2   1   cleanup
```

V. Inline functions

The code for an **inline function** is inserted in the program at each point where the function is called (instead of the ordinary calling mechanism which branches to the code for the function). Consider an example:

```
class X {
    int i;
public:
    void f(int I) { i=I; }// inline function
    int ff();             // inline function
    void fff();    // non inline function
        };
inline int X::ff() {return i; }
void X::fff() { i++; }    // push bp
                  // mov bp,sp
                  // les bx,[bp+6]
                  // inc es:word ptr [bx]
                         // pop bp
                         // retf
void main(void)
{   X x;
    x.f(5);       // mov word ptr [bp-2],5
    x.fff();              // push ss
                          // lea ax,[bp-2]
                          // push ax
                          // push cs
                          // call X::fff
                          // add sp,4
    _AX=x.ff();    }  // mov ax,[bp-2]
```

The comments show possible assembly language code created by the compiler. The code for the non inline function **X::fff()** allows for the possibility of the function being either inline or not.

VI. Some remarks on class design

There are some functions (regular functions [3]) whose semantics are the same in all well-designed classes. They are the following:

- the copy constructor;
- the destructor;
- the principal assignment operator (=);
- the equality (==) and inequality (!=) operators.

These functions are declared as follows [3]:

```
class X {
// ..........
public:
    X(const X&); // construct an object whose
// abstract value is the same as the argument
    ~X();          // destroy the object
    const X& operator=(const X&); // set the
// value of this object to the value of the
// argument, and return a reference
    bool operator==(const X&) const;// return
// true if and only if this object and the
// argument object have the same value
    bool operator!=(const X&) const;//return
// true if and only if this object and the
// argument object have the different values
// .........
    };
```

In [3] it was suggested that a class which provides all the regular functions and a default constructor should be called a **nice class** (these functions could be included in a certain minimal standard interface).

Consider a nice **array** class which implements all the regular functions (it would be a good idea to declare this class as a class template).

```
enum bool {false,true};
template <class T> class array {
    T* a;
```

```
    int s;
    T& operator[](int j) { return a[j]; }
public:
    array(int S=1) { a = new T[s=S]; }
    array(const array&);
    ~array() { delete[] a; }
    const array& operator=(const array&);
    bool operator==(const array&) const;
    bool operator!=(const array&) const;
    friend ostream&
        operator<<(ostream&,array<T>&);
    friend istream&
        operator>>(istream&,array<T>&);
                                  };
template <class T>
array<T>::array(const array& A)
{   a = new T[s=A.s];
    for(int i=0;i<A.s;i++)
     a[i] = A.a[i];       }
template <class T> const array<T>&
array<T>::operator=(const array& A)
{  if(this != &A)                {
        delete[] a;
        a = new T[s=A.s];
        for(int i=0;i<A.s;i++)
             a[i] = A.a[i];       }
    return *this;
}
template <class T> bool
array<T>::operator==(const array<T>& A) const
{   if(s!=A.s)  return false;
    for(int i=0;i<A.s;i++)
        if (a[i] != A.a[i]) return false;
    return true;
  }
template <class T> bool
array<T>::operator!=(const array<T>& A) const
{   if(s!=A.s) return true;
    for(int i=0;i<A.s;i++)
        if (a[i] != A.a[i]) return true;
    return false;
  }
template <class T> ostream&
operator<<(ostream& stream,array<T>& A)
{   for(int i=0;i<A.s;i++)
        stream << A[i] << '\t';
    stream << endl;
    return stream;               }
template <class T> istream&
operator>>(istream& stream,array<T>& A)
{   for(int i=0;i<A.s;i++)
        stream >> A[i];
    return stream;               }
```

Now we can use for example the following statements in the function main:

```
    array<int> ai1=6,ai3;
    cin >> ai1;
    cout << ai1;
    array<int> ai2=ai1;    // initialisation
    ai3 = ai1;             // assignment
    if (ai3 == ai1) // do something;
    if (ai2 != ai1)  // do something;
```

We can also define an array of objects of a user-defined type, for instance:

```
    array<student> ai1=6,ai3;
```

The type **student** must have been declared and defined previously. For example:

```
class student {
    char *name;
public:
    student();
    ~student();
    bool operator!=(const student&) const;
    bool operator==(const student&) const;
    friend ostream&
        operator<<(ostream&,student&);
    friend istream&
        operator>>(istream&,student&);
// . . . . . . . . . . . . . . . . . . . .
            };
```

V11. INHERITANCE AND CONTAINMENT

The two most common **relationships** between classes are **inheritance** and **containment**. If one class **X** has a member which is an object of another class **Y** (see figure 4, a), then we can say there is a *containment relationship*. Because **X** *HAS A* member of type **Y, it** is often said that there is a **HAS A** relationship [2,4]. When a class **D** is derived (usually publicly) from another class **B,** we can say that **D** *IS A* kind of **B** and we have an **IS A** relationship (see figure 4, b).

Inheritance is one of the most powerful tools of object-oriented programming. It is a process of building a new class (derived class) from an existing class (base class). When we build a derived class we want to **inherit** properties from its base class. A very important characteristic of inheritance is that it enables us to **reuse** existing code.

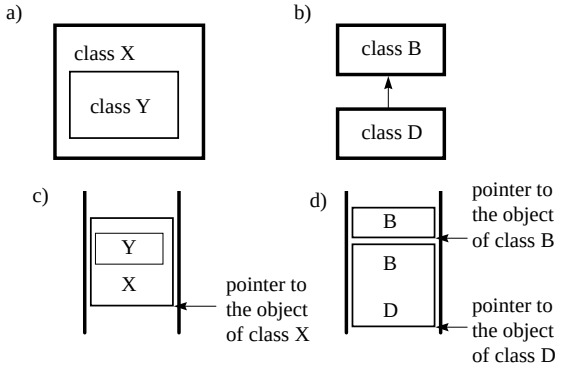

Figure 4. Containment relationship (a), inheritance relationship (b), structures and locations of respective objects in computer memory (c,d)

When we **derive** a new class we can:

- **add** new data members;
- **add** new member functions;
- **override** (change or modify) inherited member functions.

As a result the derived class is **more powerful** (is more extensive) than its base class (see figure 4, d). Consider an example.

```
class B {  // base class
protected:
    int i;
public:
// . . . . . .
};
class D : public B {      // derived class
    int j;
public:
// . . . . . .
};
void f(B* pb)
{  // . . . . . .
}
void main(void)
{   B b;
    D d;
    f(&b);
    f(&d); }       // implicit conversion
```

Here class **D** is publicly derived from class **B**. Let us declare pointers to **B** and to **D**:

```
B *pb=&b;
D *pd=&d;
```

Now we can provide access to members of **b** through the pointer **pb** and access to members of **d** through the pointer **pd**. Since a base class can be covered by a derived class (the derived class is more extensive) we can use a pointer to a derived class to access to members of its base class. In other words we can assign **&d** to **pb** without the use of an explicit type conversion:

```
pb=&d;
```

The opposite implicit conversion (**pd=&b**) is not allowed. However the conversion can be **explicit**:

```
pd=(D*)&b;
```

The same implicit conversion can be used when we pass arguments to function (see the function **f** in the example above).

VIII. VIRTUAL FUNCTIONS

Virtual functions allow the programmer to provide **overriding** inherited member functions. Functions can be declared in a base class and then **redefined** in each derived class. This idea is implemented through **pointers to functions**. The determination of which function is called for an object can be done at **runtime** and is therefore called **late binding** (or dynamic binding). This contrasts with **early binding** (static binding) which is done during compilation. The mechanism of dynamic binding is explained in figure 5.

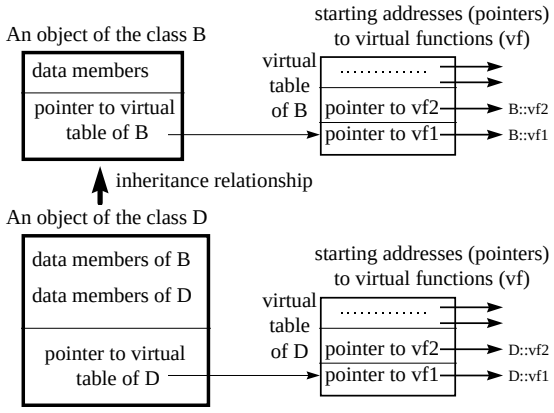

Figure 5. A mechanism of virtual functions calls

All objects of the same class have an associated **virtual table** that contains the addresses (pointers to the first instructions) of the **actual functions** to be called. A virtual function is specified by the keyword **virtual**. The type of a function declared in the base class cannot be redefined in a derived class. Consider an example:

```
class B {
protected:
    int i;
public:
    B(int I) : i(I) {}
    void decrement() { i--; }
    virtual void increment() { i++; }
};
class D : public B {
    int j;
public:
    D(int I,int J) : B(I), j(J) {}
    void decrement() { j-=2; }
    void increment() { j+=2; }
};
void f(B* pb)              // push bp
                           // mov bp,sp
{   pb->decrement();// les bx,[bp+6]
                     // dec es:word ptr[bx+2]
    pb->increment();  }
        // push word ptr[bp+8]
        // push word ptr[bp+6]
        // les bx,[bp+6]
        // mov bx,es:[bx]
        // call far [bx]
        // add sp,4
        // pop bp
void main(void)
{   B b=5;          // mov word ptr[bp-4],9Eh
                    // mov word ptr[bp-2],5
    D d(10,20); // mov word ptr[bp-8],0Ah
                // mov word ptr[bp-0Ah],92h
                // mov word ptr[bp-6],14h
    f(&b);  // push ss
            // lea ax,[bp-4]
            // push ax
```

```
            // push cs
            // call f
            // add sp,4
    f(&d);                  // push ss
                            // lea ax,[bp-0Ah]
                            // push ax
                            // push cs
                            // call f
                            // add sp,4
}
```

The comments show possible assembly language code created by the compiler. Since the function **decrement** is not virtual, in **f** it will be called for type **B** because the type of parameter for **f** is a **pointer to B**. The virtual function **increment** is called for **B** if **f** takes a **pointer to B,** and is called for **D** if **f** takes a **pointer to D**. Passing a pointer to the class is implemented through the stack before the call of the function **f**. Two assembly language instructions were underlined in order to demonstrate how a pointer to the object **b** of class **B** (in the call **f(&b);**) and a pointer to the object **d** of class **D** (in the call **f(&d);**) are passed to the function **f**. Figure 6 explains the call of a virtual function for the object **b** of the base class **B**. Figure 7 explains the call of a virtual function for the object **d** of the derived class **D**. In both cases the sequence of steps is the following:

- a pointer to an object is passed to **f**. For the call **f(&b)** it will be the pointer (0FFC) to **b** (lea ax,[bp-4], bp=1000h). For the call **f(&d)** it will be the pointer (0FF6) to **d** (lea ax,[bp-0Ah], bp=1000h);
- **f** calls the inline function, **decrement,** of the base class;
- **f** saves the address of the appropriate object on the stack (push word ptr[bp+8], push word ptr[bp+6]) and performs an indirect far call of a virtual function through its address stored in the object (les bx,[bp+6], mov bx,es:[bx], call far [bx]). In figures 6, 7 bp = 0FECh;
- the virtual function that is called accesses the appropriate object through its pointer stored on the stack (see addresses 0FE8 and 0FEA in figures 6, and 7).

When a class has at least a single virtual function it is wise to supply a **virtual destructor** in this class. The base class should have its destructor declared as **virtual** to avoid problems when derived class objects are destroyed. Consider an example.

```
class B {
    int *a;
public:
    B() { a = new int[2]; }
    ~B()   { delete[] a;  }
//  virtual ~B()   { delete[] a;  }
// ............................
};
class D : public B {
    int *a;
public:
    D() { a = new int[4];  }
    ~D()   { delete[] a;  }
// ............................
        };
void main(void)
{   B *pb=new D;
    delete pb;              }
```

Here **pb** is a pointer to the base class **B** but actually **pb** points to the derived class **D**. As a result the sequence of constructors and destructors is the following:

- constructor of **B**;
- constructor of **D**;
- destructor of **B**.

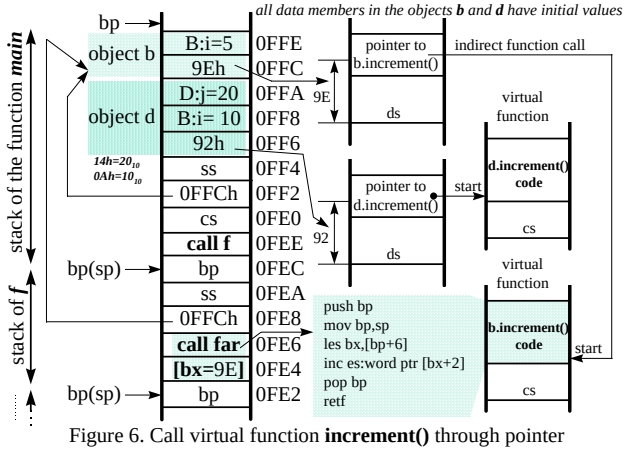

Figure 6. Call virtual function **increment()** through pointer to the object **b** of the base class **B**

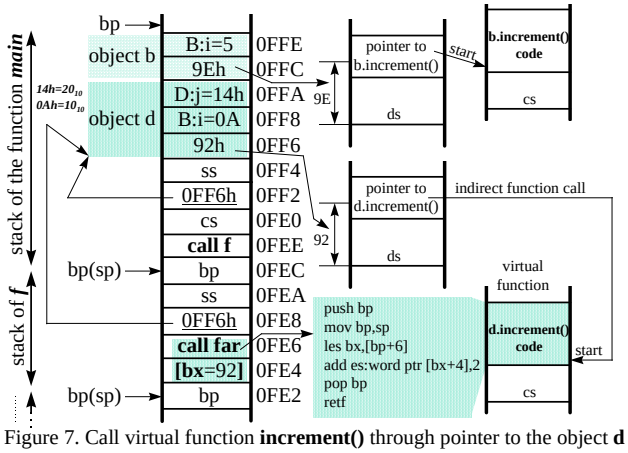

Figure 7. Call virtual function **increment()** through pointer to the object **d** of the derived class **D**

So, we have a problem when derived class object is destroyed. If we declare the destructor of **B** as virtual (see comments above) then we will have the correct sequence of constructors and destructors:

- constructor of **B**;
- constructor of **D**;
- destructor of **D**;
- destructor of **B**.

A constructor **cannot be virtual** because it requires the exact type of the object.

IX. SMART POINTERS

Smart pointers are objects that act like pointers. In order to create smart pointers we have to overload de-referencing operator **->**. Let us consider an example:

```
template <class T> class P {
    T *p;
public:
    P(T *mp=NULL) : p(mp) {}
    ~P() { delete p; }
    T* operator->() const { return p; }
// another functions like copy constructor,
// assignment operator, etc.
};
```

Now we can use objects of class **P** to access members of class **T** in a very similar manner to the way pointers are used. Consider the following function:

```
void F(int I)
{       P<X> px=new X(I);
        px->fX();          }
```

The statement **px->fX();** is interpreted as follows:

```
    (px.operator->())->fX();
```

The constructor of an object **px** was provided with a pointer to a new object of class **X** (see the first statement which is equivalent to **P<X> px(new X(I));**). Now **P::p** points to a new object of class **X**. The function **px.operator->()** returns the pointer **P::p** which can be further used as a pointer to the object of class **X** in order to access the member function **fX()** of **X**.

Consider an example which demonstrates how smart pointers can be used. Suppose we want to provide a class **X** with an **exception handling** mechanism. Let us assume that a function **fX()** which is a member of class **X** can detect a problem that it cannot cope with. As a result **fX()** throws an exception , hoping that its caller can handle the problem. The **main** function contains a **try** statement block enclosing the code in which it wants to catch any error. If an error occurs, the respective **exception handler** (catch block) will be invoked (see the example below).

```
class X {
    int i;
public:
    X(int I=0) : i(I) {}
    void fX() { if (++i>10) throw exc(); }
    class exc {};
};
void F(int I)
{   X *px = new X(I);
        px->fX();
    delete px;
}
void main(void)
{   int j=10;
s:  try {   F(j);   }
    catch(X::exc)
    {   // do something
        j--;
        goto s;
            }
}
```

Now consider what would happened if the statement **px->fX();** throws an exception. Since the exception will propagate to its caller (to the function **main**) all statements in the function **F** after the call to **px->fX();** will be skipped. This means that ***px** will never be deleted. So, each time **px->fX();** throws an exception, the function **F** will contain a **memory leak**.

In order to avoid a memory leak when an exception is thrown, we can add **try** and **catch** statements in the code for the function **F**, for instance:

```
void F(int I)
{   X *px = new X(I);
    try {
        px->fX();
            }
    catch(...)     // catch all exceptions
    {  delete px;  // avoid memory leak
        throw; }   // propagate the exception
    delete px;
}
```

However this is difficult to maintain [5]. A better way is to replace the pointer **px** with an object that acts like a pointer (a smart pointer) [5]. When the pointer-like object is automatically destroyed (because this object is local) we can invoke the operator **delete** in its destructor. Consider an example:

```
template <class T> class P {
    T *p;
public:
    P(T *mp=NULL) : p(mp) {}
    ~P() { delete p; }
    T* operator->() const { return p; }
    // another functions
};
class X  {
    int i;
public:
    X(int I=0) : i(I) {}
    void fX() { if (++i>10) throw exc(); }
    class exc {};
};
void F(int I)
{       P<X> px=new X(I);
        px->fX();
}
void main(void)
{   int j=10;
s:  try {   F(j);   }
    catch(X::exc)
    {   // do something
        j--;
        goto s;
            }
}
```

The idea of using automatic objects (smart pointers) instead of objects allocated on a free store can be applied in a variety of ways to many different tasks.

X. Virtual base classes

With multiple inheritance, a base class can be indirectly passed to the derived class more than once. Consider an example.

```
class B {
    int a;
public:
    B() : a(5) {}
    };
class D1 : public B {
// class D1 : virtual public B    {
    int a;
public:
    D1() : a(6) {}
                    };
class D2 : public B  {
//class D2 : virtual public B     {
    int a;
public:
    D2() : a(7) {}
                    };
class D1D2 : public D1, public D2        {
// class D1D2 : virtual public B, public D1,
// public D2  {
    int a;
public:
    D1D2() : a(8) {}
                                          };
void main(void)
{   B b;
    D1 d1;
    D2 d2;
    D1D2 d1d2;              }
```

In this case, each object of the class **D1D2** has two sub-objects of the class **B** (structures of the objects **d1d2**, **d1**, **d2** and **b** in computer memory are shown in figure 8,a). If this causes problems (see, for example, [4]) the base class **B** can be specified as **virtual** (see lines of the code shown in comments above). In this case, each object of the class **D1D2** has just one sub-object of the class **B** (structures of the objects **d1d2**, **d1**, **d2** and **b** in computer memory are shown in figure 8,b). Now it is provided indirect access to the sub-object **b** through its address stored in the objects **d1**, **d2** and **d1d2** (see figure 8,b).

XI. Acknowledgements

Many thanks to ***Ivor Horton*** for his help with this article.

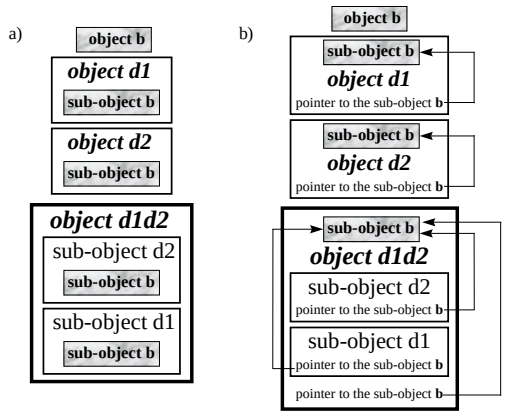

Figure 8. Structures of non virtual (a) and virtual (b) objects in computer memory

XII. Conclusion

We have discussed some of the constructions accepted in object-oriented programming. On the one hand these constructions are very useful and enable us to build effective object-oriented code. On the other hand the teaching experience shows that they are difficult for students to apply and are error-prone. The article examines some of the more common errors, shows the correct implementation in each case, and explains how the computer executes different C++ instructions at different levels. Finally it allows the students to eliminate errors and to better understand object-oriented techniques. The paper assumes a prior knowledge of C++ (I can recommend the book [6] which contains a full tutorial to the C++ language). The basic approaches involved in the use of object-oriented programming are also presented and discussed in the paper [7].

References

[1] Anton Eliens Principles of Object-Oriented Software Development. Addison-Wesley, 1995, 513 p.

[2] Michael C. Daconta C++ Pointers and Dynamic Memory Management. John Wiley & Sons, Inc., 1995, 464 p.

[3] Martin D.Carroll, Margaret A.Ellis Designing and coding reusable. Addison-Wesley, 1995, 317 p.

[4] Bjarne Stroustrup The C++ Programming Language. Second Edition, Addison Wesley Publishing Company, 1994, 691 p.

[5] Scott Meyers More Effective C++. Addison Wesley Publishing Company, 1996, 318 p.

[6] Ivor Horton Beginning Visual C++ 4. WROX, 1996, 825 p.

[7] Valery Sklyarov From Procedural to Object-Oriented Programming. Electrónica e Telecomunicações, 1995, vol.1, N 3, pp 217-223.