10 Polymorphism and Virtual Functions

As seen before, in the case of inheritance there it the possibility to redefine in the derived class of some member functions from the base classes. The redefinition of a function with the same name, but with different parameters and with different content, has the name ***overloading***.

In addition to the overloading operation, the C++ language has another method for defining of some functions with the same name in a class hierarchy: the method of ***virtual functions***. The advantage of using virtual functions instead of the overloaded ones it is very important: in the case when of using a pointer to the base class for accessing an object from the (public) derived class, the compiler can exactly determine the functions which must be called, knowing the current object type to which the pointer shows (even if the pointer belongs to a base class). In this case the functions are ***overridden***: they have the same name and the same formal parameters.

The difference between the two methods is important, but in fact it refers to the moment when the zone of the memory containing the code of the functions that are called is determined. This operation is known as ***function binding***. In the case overloaded functions there is a ***static binding*** (***earlier binding***), because the compiler can determine the memory address of the functions when compiling the program; in the case of virtual functions there is a ***dynamic binding*** (***late binding***), because the determination of the memory address of the called functions is determined at runtime.

The objects with the same name which belong to some classes which contain virtual functions are called *polymorphic*. These objects have the same "face" determined by the interface of the base class, but more "forms", determined by different virtual functions. A language which allows the use of polymorphic objects it is said to support the polymorphism concept. The polymorphic objects and the notion of polymorphism represent the main characteristic of the object-oriented programming paradigm.

The virtual functions are strictly related to the notions of polymorphism *and* inheritance, while the functions' overloading is related only to the inheritance concept. An example is the operator overloading functions.

The polymorphism represents the distinctive element of the object-oriented programming, while virtual functions represent the implementation method of the polymorphism in the C++ language. From this reason, a programmer which uses only class composition and the inheritance mechanism, did not reached a sufficient stage of mastery of the C++ language (according to Stroustrup).

10.1 Overloaded functions and virtual functions

The use of inheritance and overriding the member functions represents a method of generating polymorphic objects. In the case when there are used *objects* and not *memory addresses of objects* (*pointers* or *references*), the call of overridden functions is performed in safe conditions.

Example:

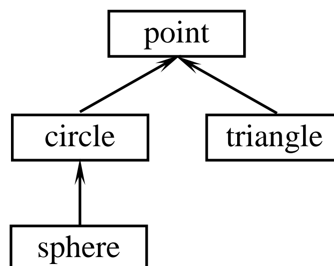
```
class point {
public:
  double x, y;
  point (double x0 = 0, double y0 = 0) { x = x0; y = y0; }
  double Aria() const { return 0.0; }
};

class circle: public point {
public:
  double r;
  circle(double x0 = 0, double y0 = 0,
      double r0=1): point(x0, y0), r(r0) {}
  double Surface() const { return 3.1415*r*r; }
};

int main () {
  point p(1, 1);
  circle c(1, 1, 1);
  cout << "Point surface= " << p.Surface() << endl;
  cout << "Circle surface= " << c.Surface() << endl;
  return 0 ;
}
```

The output of the previous program is correct; it displays the values 0 and 3.1415.

A very used method when calling the methods of instances of some classes from a class hierarchy is called ***upcasting*** and it consists in using pointers to objects from base classes, for referring objects both from base classes and from derived classes (which, in this case, are considered ***subtypes*** of base classes).

The advantage of using the `upcastintg` method is relevant when using containers that contain objects of different classes, derived from a unique base class.

Example: Let's consider the class hierarchy:



```
class point {
  // as before
};

class circle: public point {
  // as before
};

class sphere: public circle {
public:
  sphere(double x0, double y0, double r0): circle(x0, y0, r0) { }
  double Area () const { return 4*3.1415*r*r; }
};

class triangle: public point {
public:
  double x1, y1;
  double x2, y2;
  triangle(double x00, double y00, double x01, double y01,
     double x02, double y02):
     point(x00, y00), x1(x01), y1(y01), x2(x02), y2(y02) { }
  double Area () const
     { return (x*y1+x1*y2+x2*y-y*x1-y1*x2-y2*x)/2; }
};

int main () {
  point *v[] = { new circle(1, 1, 1), new sphere(1, 1, 1),
     new triangle(0, 0, 1, 0, 0, 1) };
  cout << v[0]->Area() << endl;
  cout << v[1]->Area() << endl;
  cout << v[2]->Area() << endl;
  return 0;
}
```

In the previous example the objects of the respective hierarchy can be uniformly treated. In this case, of using the `upcasting` method appears a problem: which from the overridden functions will be executed when using pointers instead of objects?

In the previous example, in all three cases the function *Area* from the *point* class will be called, even if the three objects belong to different classes.

Such a *problem* appears because of the way in which the compiler determines the function which will be called. This action is called ***function binding*** and it supposes the association between the function prototype and its content. When it is not specified otherwise, the *function binging* is ***static*** and it is done during the compilation process. Because the pointers can point to different objects

during the program execution, for the safety of the call mechanism it is considered that the objects towards they point, belong to the class specifies at the pointer declaration (the *point* class in the previous example).

The solution to this problem is ***dynamic binding*** of functions, which is realized at runtime and not at compilation. The dynamic binding mechanism is different at different programming language. In the case of the C++ language it consists at the insertion of some supplementary information type in the instance objects.

From the syntactic point of view, in the C++ language, the *dynamic binding* of functions is realized by using the keyword ***virtual***, which is specified in the class declaration from which the functions belong. For this reason, a function for which the dynamic binding is desired, is called ***virtual function***.

Syntactically speaking, the keyword `virtual` precedes the declaration of a function and it is not repeated when the function is implemented outside of the class (with the exception of functions declared `inline` inside of the class). A function virtual declared in a base class preserves this property in all derived classes, so that the keyword `virtual` must not be specified when redefining of such a function in a derived class.

In order to the previous program to work correctly, the function *Area* from class *point* must be declared `virtual`:

```
class point {
public:
  double x, y;
  punct(double x0 = 0, double y0 = 0): x(x0), y(y0) { }
  virtual double Area () { return 0.0; }
};
```

In this case the program will display the correct the values of the three objects.

Using virtual functions is advantageous from at least two points of view:

- assure the correctness of the upcasting method for uniformly treating the instance objects from a class hierarchy with a single base class,
- allow the extension of an application created upon an already existent class hierarchy.

For example, if at the previous class hierarchy the following class will be added:

```
class pyramid:public triangle {
public:
  double x3,y3;
  pyramid(double x00, double y00, double x01, double y01,
      double x02, double y02, double x03, double y03):
      trangle(x00, y00, x01, y01, x02, y02),
      x3(x03), y3(y03) { }
  double Area () const; // calculates the area of the sides
```

```
                                // of a pyramid
};
```

and also the function:

```
double CostPrice(point* p, double unitary_ price) {
  return p->Area() * unitary_price;
}
```

the call of the function *CostPrice* is correct for both the objects from previous classes, and for the objects of the class *pyramid*:

```
point* v[]={ new circle(1, 1, 1), new sphere(1,1,1),
     new triangle(0, 0, 1, 0, 0, 1),
     new pyramid(0, 0, 0, 1, 1, 0) };
cout << CostPrice(v[0], 10) << endl;
cout << CostPrice(v[1], 10) << endl;
cout << CostPrice(v[2], 10) << endl;
cout << CostPrice(v[3], 10) << endl;
// ...
```

Except fot the binding mechanism of functions, between the overloaded and overridden (virtual) functions there exists an important difference imposed by the unique prototype of virtual functions: the overloaded functions can have different parameters as number and as data type, while the virtual functions **must** have the same number and type of formal arguments (except for the hidden argument `this`).

For this reason it is easy to make the mistake of writing an overloaded function instead of a virtual one. The redefinition of a virtual function in a derived class is usual called ***overriding.***

Example:

```
class A {
  virtual int f(int k) { return k; }
};

class B: public A {
  // overloaded function
  int f(unsigned int k) { return k+1; }
};

class C: public A {
  //virtual function overriding
  int f(int k) { return k+2; }
};

int main () {
  A *p1 = new B;
  A *p2 = new C;
```

```
    cout << p1->f(7) << endl;
    cout << p2->f(7) << endl;
    B b;
    cout << b.f(7) << endl;
    return 0;
  }
```

The function *f* of the class *B* is overloaded because the parameter *k* does not correspond as data type of the argument with the virtual function *f* from the base class, so that at the first two calls will be displayed 7 and 9, but at the last call will be displayed the value 8.

Observation: When the *f* function is called through the pointer *p1*, the compiler chooses the virtual function from the class, because the class *B* has no virtual function. In the case of calling *f* through the object *b*, the overloaded function *f* from the class *B* will be selected, because in this case the upcasting technique is not used.

10.2 Implementing of dynamic binding of functions (and thus the polymorphism)

For determining which function will be called in the case of dynamic binding, most C++ compilers use a table called *VFTABLE*, related to those classes that contain virtual functions. Moreover, objects of a class containing virtual functions use a hidden pointer, called *VFPTR*, which points towards the *VFTABLE* table of the associated class.

Even this method is very used in several C++ compilers, the method of implementing virtual functions is not imposed by the standard of the C++ language.

As previously specified, an instance of a class has associated a zone of memory, where the object stores the values for its data members. For classes with virtual functions there exists, in addition, a pointer to the *VFTABLE* table of its class.

Example:

```
class A { // class without virtual function
  int k;
public:
  void f() const {}
  int g() const { return k; }
};

class B { // class with one virtual function
  int k;
public:
  void f() const {}
  virtual int g() const { return k; }
};
```

```
class C { // class with two virtual function
  int k;
public:
  virtual void f() const {}
  virtual int g() const { return k; }
};

int main() {
  A a;
  B b;
  C c;
  int x = sizeof(void *);
  int y = sizeof(a);
  int z = sizeof(b);
  int u = sizeof(c);
  cout << x << endl << y << endl << z << endl << u << endl;
  return 0;
}
```

From the previous paragraph one can observe that:

```
u = z = x + y
```

In other words, in the case of classes, which contain virtual functions, or which are derived from other classes with virtual functions, their instances contain an additional component having the type `void*`.

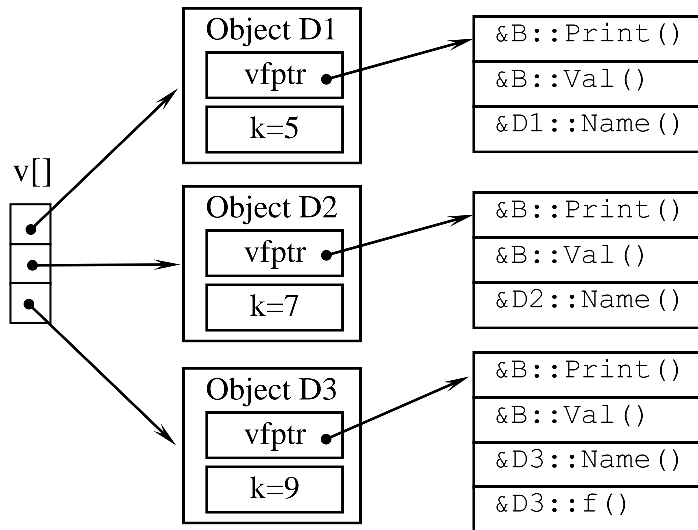
The generation of VFTABLE tables for these classes, and also the insertion and initialization of the VFPTR pointer for the instances of these classes is realized automatically by the compiler.

To better understand the implementation of the dynamic binding mechanism, we consider the following sequence:

```
class B {
  int a;
public:
  B(int n=0): a(n) {}
  virtual void Print() const { cout << a << endl; }
  virtual int Val() const { return a; }
  virtual void Name() const { cout << ”B” << endl; }
};

class D1: public B {
public:
  D1(int n): B(n) {}
  void Name() const { cout << ”D1” << endl; }
};

class D2: public B {
public:
  void Name() const { cout << ”D2” << endl; }
  D2(int n): B(n) {}
```

```
};

class D3: public B {
public:
  void Name() const { cout << "D3" << endl; }
  D3(int n): B(n) {}
  virtual void f() const {}
};

B *v[] = { new D1(5), new D2(7), new D3(9) };
```

The graphic description of *v* array is shown in the following figure:



One can see that for each class containing at least a virtual function, or in the case when the class is derived from another class which contains virtual functions, the compiler creates a single *VFTABLE* table. In this table there are written the addresses of the virtual functions that are specific to the class and, eventually inherited, in the order in which they appear in the class declaration. When a class inherits a virtual function from a base class without overriding it, the address written in the *VFTABLE* table is the address of the function from the class where it was defined, on the same position as in table of the base class. The *VFPTR* pointer from each instance of such a class will always indicate to the first component of the *VFTABLE* table of the associated class.

Remark: Usually the base class offers through the virtual functions an interface for the derived classes, and these derived classes do not add new virtual functions. When this case happens, and a pointer at the base class is used, for calling a virtual function which exists only in a derived class, the respective function call must be prefixed with a *cast* operator. For example, for the previous example, the next call is correct:

```
(D3*)v[2]->f()
```

while the call:

```
v[2]->f()
```

is incorrect.

Consider the following sequence:

```
int main() {
  B *p = v[2];
  p->Name();
  // ...
}
```

The code inserted by the compiler at the call `p->Name()` is different from the case when virtual functions did not exist. In the case of *static binding* will be called the function which is at the address `&B::Name()`, and the string "B" string will displayed.

In the case of *dynamic binding* (in the presence of virtual functions), will be called the function starting at the address written at *VFPTR+2* (because the function *Name* is on the third position in the order of defining virtual functions), that is the function at the address `&D::Name()` and thus the string "D2" will be displayed. Because of the address of the object *v[2]*, and thus the value written at the address *VFPTR + 2* are determined only at the runtime, the binding for this function is dynamic type in this case.

Usually, the code generated by the compiler when calling a virtual function through a pointer *p*, is similar to the following sequence:

```
(*p->vfptr[k])();
```

where *k* is the index associated to the position of the virtual function in the table *VFTABLE*.

The *VFPTR* initialization is realized automatically by the compiler, in the class constructors, even in the case of the default constructor generated by the compiler.

Unlike other object-oriented programming languages, such as Smalltalk or Java, where the function binding is always dynamic, in the case of C++ language it is possible as in certain cases the binding of virtual functions to be static.

When using the upcasting technique, the binding for virtual functions is always dynamic, because the compiler does not know enough information for generating the calling sequence. When a virtual function is called through an object (and not trough a pointer or a reference), the compiler has enough information for generating the calling sequence during the compilation process. Usually, in this case, the function binding will be static (for most of the C++ compilers).

For example, for the previous class hierarchy, we consider the following sequence:

```
void processing() {
  D1 o1;
  B *p = &o1;
  B &q = o1;
  D2 o2;
  p->Name();     // dynamic binding
  q.Name();      // dynamic binding
  o2.Name();     // static binding
```

```
  // ...
}
```

In the first two cases the binding is dynamic, because there are used objects addresses, while in the last case the binding is static.

The reason for which the C++ language uses static binding for virtual functions called through objects is efficiency. The generated code for functions call using dynamic binding is significantly greater than in the case of static binding.

Because of the fact that the table *VFTABLE* must contain the memory address of the code of the virtual functions, the virtual functions cannot be `inline`. In the case of `inline` functions the compiler will not generate standard code for calling functions (jump to the address of the function code). Even a class will contain inline virtual functions, compiler will treat these functions as usual functions (not `inline`).

10.3 Abstract classes and pure virtual functions

In most cases in the design phase of a class hierarchy it is desired as the base class of the hierarchy (especially in case of simple inheritance) to offer only an interface for derived classes. This means that such a class should not have instance objects, but to be used for the upcasting operation. Such a class is called ***abstract class*** and this fact can be realized by using ***pure virtual functions.***

A pure virtual function is a virtual function for which the function prototype is followed by "***= 0***". Usually, such a function has not an implementation in the base class. In the same time it is imposed as in the derived classes to exist implementations for the respective function.

Although an abstract class cannot create instance objects, one can use pointers or references to such a class. In fact, this is the usual method, in which by using upcasting there are referred objects of the derived classes by using pointers to the base class.

Example:

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

class value {
public:
  virtual void Print() const = 0;
};

class integer: public value {
public:
  int n;
  integer(int k = 0): n(k) {}
  void Print() const { cout << n << endl; }
```

```
};

class real: public value {
public:
  double x;
  real(double v = 0.0): x(v) {}
  void Print() const { cout << x << endl; }
};

class myString: public value {
public:
  char *s;
  sir(char *str) {
    s = new char[strlen(str)];
    strcpy(s, str);
  }
  void Print() const { cout << s << endl; }
};

int main() {
  value *v[] = { new integer(5), new real(7.5),
    new myString("example")};
  for (int i=0; i<3; i++)
    v[i]->Print();
  return 0;
}
```

When a function is declared as being *pure virtual*, the compiler reserves an entry in the *VFTABLE* table of the respective class, but it does not write any value in that location. In this way, the table *VFTABLE* for an abstract class is incomplete and it cannot be used to create instance objects of the class.

The C++ language allows as *pure virtual functions* to have also csome definitions in a base class. These are called ***pure virtual definitions*** and they are not viewed as the implementations of the virtual functions. The reason of allowing *pure virtual definitions* is the simplification of writing programs: in the case when two or more implementations of a pure virtual functions in derived classes contain some common code, it is simpler that the respective part of code to be written only once in the base class as a *pure virtual definition*. Because of the inheritance mechanism, the pure virtual definitions are inherited in derived classes and can be used in the implementation of the pure virtual functions.

Example: The modification of the previous example, where the pure virtual function *Print* from the *value* class has a pure virtual definition:

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

class value {
public:
```

```
    virtual void Print() const = 0;
  };

  void value::Print() const { cout << "The value : "; }

  class integer: public value {
  public:
    int n;
    integer(int k = 0): n(k) {}
    void Print() const {
      value::Print();
      cout << n << endl;
    }
  };

  class real: public value {
  public:
    double x;
    real(double v): x(v) {}
    void Print() const { value::Print(); cout << x << endl; }
  };

  class myString: public value {
  public:
    char *s;
    sir(char *str) {
      s = new char[strlen(str)];
      str copy(s, str);
    }
    void Print() const { value::Print(); cout << s << endl; }
  };
```

Another advantage of pure virtual definitions is that one can transform a common virtual function in a pure virtual function without changing the program part already written, only by insertion of the characters "***= 0***" in the respective function prototype of the base class.

10.4 Virtual functions, virtual constructors and destructors

As already seen before, the constructors cannot be inherited in a class hierarchy. From this reason it is not justified their declaration as virtual functions (***constructors cannot be virtual functions***). The role of a constructor role is very important, especially in the cases of classes that contain virtual functions. Usually, at the beginning of such a constructor the compiler inserts the code for the *VFPTR* initialization of the current object to the first element of the VFTABLE table.

Even if the constructors cannot be virtual functions, theoretic it is possible as inside of a constructor to be *called a virtual function*. From practically point of view this variant it is not indicated, because in this case the dynamic binding mechanism will not work, and it will be

selected for execution always the local version from the current class of the respective virtual function.

There exist two important reasons for this mode of treating:

1. First, while executing a constructor, the associated object is only partial built, without knowing what objects are derived from the object of the base class. If it used the dynamic binding for a virtual function from inside the constructor, this would imply that will be used members which are not yet initialized (which are at another level of the hierarchy, possible lower), which is a source of potential errors.

2. The second reason relates to the successive calls of constructors in the case of a derived object. Each called constructor uses the *VFPTR* pointer that points to the *VFTABLE* table of its class, so that only the last called constructor generates completely the associated *VFTABLE* table. Because in the case of dynamic binding it is used the *VFTABLE* table for determining the called virtual function, it will be used only the local table of the class, and not the last determined table, which is valid only after the last constructor call (the constructors are always called in the order: base class, first derived class, second derived class, e.t.c).

In contrast to constructors, ***the destructors can be virtual functions***; in certain cases is strongly indicated to be virtual functions.

The calling order of destructors is backward to the one for constructors: from the derived class on the lowest level of the class hierarchy, to its base class.

In the case when there are not used pointers and the `delete` operator, it is not necessary as the destructors to be virtual functions. However, if the `delete` operator is used on a pointer to a base class and there are not virtual destructors, the compiler will not have enough information on the object indicated by the pointer and the destructor from the base class will be called.

Example:

```
class A {
public:
  ~A() { cout << ″ A class destructor″ << endl; }
};

class B: public A {
public:
  ~B() { cout << ″ B class destructor″ << endl; }
};

int main() {
  A *p = new B;
  delete p;
  {
     B b;
  }
```

```
    return 0;
  }
```

From the previous program output one can observe that for the pointer *p* it is called the destructor of the class *A*, while for the object *b* it is called the destructor of the class *B* (in both cases it is called the constructor of the class *B*).

Calling a destructor of the base class can cause different errors in a program if the object has been created as an instance from another class, and for certain data member its memory is not deallocated (in the previous example for the *p* pointer it is called the constructor from the *B* class and the destructor from the *A* class).

The solution to this problem is to declare the destructor from the base class as a virtual function. For the previous program, the correct code will be:

Example:

```
class A {
public:
  virtual ~A() { cout << ”Destructor class A” << endl; }
};

class B: public A {
public:
  ~B() { cout << ”Destructor class B” << endl; }
};
```

In this case the dynamic binding mechanism works (the destructors are also functions) and the statement `delete p;` calls the destructor for the *B* class (in addition the destructor of the *A* class will be also called, because *A* is the base class for *B*).

Unlike constructors, in the case of destructors it is knew the object type indicated by a pointer because the object has already been built and the *VFPTR* pointer has been initialized, so the dynamic binding mechanism can take place.

Besides virtual destructors, the C++ language allows also ***pure virtual destructors***. In comparison with the other pure virtual functions, the *pure virtual destructors* have some specific characteristics:

- Their implementation must be specified also in the base class;
- In the derived classes it is not compulsory to replace the implementation from the base class.

In fact, the only difference between a *virtual destructor* and a *pure virtual destructor* is only in the case when the base class has no other *pure virtual functions* and in this way the pure virtual destructor makes a class to be abstract, not allowing object instances for the respective class. This is in fact the goal of using pure virtual destructors.

Example: The next program is similar to the previous one:

```
class A {
public:
  virtual ~A() = 0;
};

A::~A() { cout << ”Destructor class A” << endl; }

class B: public A {
public:
  ~B() { cout << ”Destructor class B” << endl; }
};

int main() {
  A *p = new B;
  delete p;
  return 0;
}
```

In conclusion:

- The *virtual destructors* are used or in the case when the respective class has at least a virtual function, or in the case when there are used pointers and the *upcasting* mechanism;
- The *pure virtual destructors* are used in the case when it is desired that a certain base class to be an abstract class and it does not has other pure virtual functions.

As for constructors, but from different reasons, the dynamic binding mechanism for calling functions from inside the destructors it is allowed by the compiler, so that when calling a virtual function inside of a destructor will be executed the local variant of the function from the class.

If the mechanism of dynamic binding should function, when calling a function from a constructor, it is possible to be called a function of an already destroyed object, because of the call order of the destructors (from the derived class towards the base class).

Example:

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

class A {
public:
  virtual ~A() {
    cout<< ”Destructor class A” << endl;
    f();
  }
  virtual void f() { cout << ”function in classA” << endl; }
};

class B: public A {
```

```
public:
  ~B() { cout << "Destructor class B" << endl; }
  void f() {cout << "function in class B" << endl; }
};

int main() {
  A *p = new B;
  delete p;
  return 0;
}
```

One can observe on the previous example, that it is called the function *f* from the base class *A*, even if *f* is a virtual function and it is redefined in the *B* class. The reason for this behavior of the compiler is that if the dynamic binding would work, it should be called a function member of an object which has been already destroyed. In the previous example it is called the destructor for the *B* class, then the destructor for *A* class, and from this moment it cannot be called the function *f* from the *B* class (the object have already been destroyed) and it is called the local function *f* from the *A* class.

10.5 Examples of class hierarchy with a single root

One of the problems encountered in the case of ***containers*** is the "owner's problem": the determination of the class which is responsible for destroying the dynamic objects with through the `delete` operator. Usually, for more flexibility, so that the container can contain objects of different types, the objects are specified by pointers to `void`. The call of a `delete` operator on a pointer to `void` does not call the destructor of a certain class, so that the container must be responsible with the destruction of the contained objects.

A used solution in applications is a class hierarchy with a single root.

In the next example a stack will be defined as a container which can have derived objects from a general class called *object*.

```
//file stack.h
#ifndef STACK_H
#define STACK_H

class object {
public:
  virtual ~object() = 0;
};

inline object::~object() {}

class stack {
  struct Link {
    object *data;
```

```
    Link *next;
    Link(object* o, Link* n): data(o), next(n) {}
  } *top;
public:
  stack(): top(0) {}
  ~stack() {
    while(top)
      delete pop();
  }
  void push(object* o) {
    top = new Link(o, top);
  }
  object *pop() {
    if (top == 0) return 0;
    object *tmp = top->data;
    Link *l = top;
    top = top->next;
    delete l;
    return tmp;
  }
};

#endif
```

The destructor of the *object* class and the *pop* function have been defined inline for simplicity, so that to kept all the definitions inside the header file *stack.h.*

The stack previous defined is very flexible, because its elements store pointers to *object*. The only restriction is that the objects kept in the stack to be instances of some derived classes from the *object* class. If it is desired that the elements from the stack to belong to other certain class, this method needs multiple inheritance (from the *object* class and from the respective class).

For example, the *MyString* class uses the the *string* library class, and also the *object* class:

```
#include “stiva.h”
#include <string>
using namespace std;

class MyString: public string, public object {
public:
  MyString(char *s): string(s) {}
};

int main() {
  stack st;
  st.push(new MyString(“item 1”));
  st.push(new MyString(“item 2”));
  st.push(new MyString(“item 3”));
  MyString *s;
  for(int i=0; i<3; i++) {
```

```
    s = (MyString*)st.pop();
    cout << *s << endl;
  }
  return 0;
}
```

Because the stack class knows the objects types it contains, the appropriate class destructor is called (here *MyString* class) at the call of the `delete` operator.

10.6 Multiple inheritance and virtual base classes

In the Section 7.4 the main problems generated by multiple inheritance have been specified: doubling the hidden objects and the existence of some functions from different base classes which have the same name. If the last problem has a simpler solution by redefining the functions from the derived class, the first problem can lead to special situations in the case of using the upcasting method.

In the case of multiple inheritance, an instance object of a derived class from more base classes has more pointers **`this`**, each one for a sub-object of a base class.

Example:

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

class B1 {
public:
  void Print1() const
    { cout << "B1: this= " << this << endl; }
};

class B2 {
public:
  void Print2() const
    { cout << "B2: this= " << this << endl; }
};

class M: public B1, public B2 {
public:
  void Print() const {
    cout << "M: this= " << this << endl;
    Print1();
    Print2();
  }
};
```

```
int main() {
  M m;
  m.Print();
  B1* b1 = &m;
  B2* b2 = &m;
  cout << "B1: pointer= " << b1 << endl;
  cout << "B2: pointer= " << b2 << endl;
  return 0;
}
```

One can observe that the *b1* and *b2* values are different and they correspond to the addresses of the hidden objects of the *B1* and *B2* classes.

In the case when two classes from a hierarchy (one derived and one base class) exists more paths, it means that all the objects from the derived class will have more hidden sub-objects of the base class: one for each path between the derived class and the base class.

From the above reason, in the case when upcasting is used, the compiler will show an error at the step of converting the derived object address in the base object address.

Example:

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

class B {
public:
  virtual ~B() {}
  virtual char* f() { return "D1"; }
};

class D1 : public B {
public:
  char* f() { return "D1"; }
};

class D2 : public B {
public:
  char* f() { return "D2"; }
};

class M : public D1, public D2 {
public:
  char* f() { return D1::f(); }
};

int main() {
```

```
  B* b[3];
  b[0]= new D1;
  b[1] = new D2;
  b[2] = new M;     // !! Erorr: conversion
//not clear
  for(int i = 0; i < 3; i++)
    cout << b[i]->f() << endl;
  return 0;
}
```

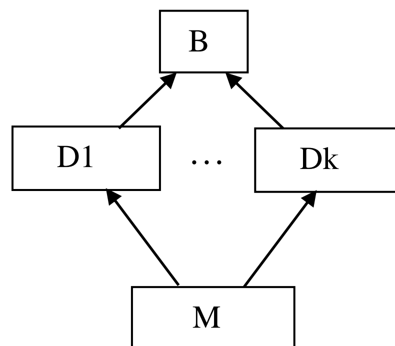
A solution to this problem is the destruction of the double sub-objects. This is done with the help of an extension of the derivation mechanism, called ***virtual derivation***. The base class from which is virtual derived a derived class is named ***virtual base class***.

Syntactically speaking, the virtual derivation is specified by the keyword `virtual`, which prefixes name of the base class. For example, for the classes *D1* and *D2*, their correct definition is:

```
class D1 : virtual public B {
public:
  char* f() { return "D1"; }
};

class D2 : virtual public B {
public:
  char* f() { return "D2"; }
};
```

The *B* class becomes a ***virtual base clas***s, and with this modification, the previous program works correctly.

For a virtual derived class from a base class, only a sub-subject of the base class will be stored by the derived classes. In order this mechanism to work, it must that all classes *D1*, *D2*, …, *Dk*, derived from a virtual base class **B**, and *D1*, *D2*, …, *Dk*, can be base classes for another derived class *M*, to virtual derived from base class B.



So, because the instance objects of the *M* class will have only one sub-object of the *B* class, it will be no confusion in the case of upcasting conversion.

Because that in the derived *M* class it will be only one sub-object of the virtual base class *B*, the constructors of the derived classes D1,D2,…,Dk, could not call the constructor of the base class B when these constructors are called inside the constructor of the M class. In this case, the constructor of the virtual base class will be called in each constructor of the classed derived from the virtual base class.

Example. Continuing the previous example, by adding one more class.

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

class B {
public:
  B(int) {}
  virtual char* f() const = 0;
  virtual ~B() {}
};

class D1 : virtual public B {
public:
  D1() : B(1) {}
  char* f() const { return "D1"; }
};

class D2 : virtual public B {
public:
  D2() : B(2) {}
  char* f() const { return "D2"; }
};

class M : public D1, public D2 {
public:
  M() : B(3) {}
  char* f() const {
    return D1::f();
  }
};

class X : public M {
public:
  // Virtual base class must always be initialized
  X() : B(4) {}
};

int main() {
```

```
  B* b[4];
  b[0]= new D1;
  b[1] = new D2;
  b[2] = new M;
  b[3] = new X;
  for(int i = 0; i < 4; i++)
    cout << b[i]->f() << endl;
  return 0;
}
```

One can observe from the previous example that the constructor of the *B* class has been called in constructors al all derived class from *B*, even in the constructor of the *X* class, which inherited *B* indirectly.

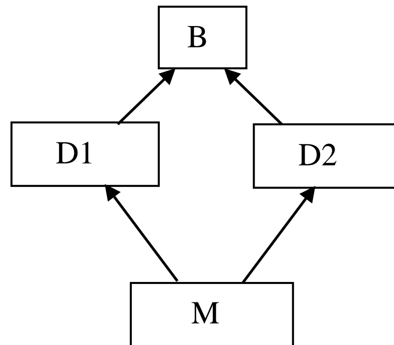
Remark. In the case when the virtual base class *B* has a default constructor, the default constructor of a derived class from a virtual derived class from *B* not need to have an explicit call of the default constructor of *B*, because this is automatic generated by the compiler. In this way it results a simplification of the definition of the class hierarchy.

Example. Rewriting previous hierarchy, where were used default constructors.

```
class B {
public:
  B(int = 0) {}
  virtual char* f() const = 0;
  virtual ~B() {}
};

class D1 : virtual public B {
public:
  D1() : B(1) {}    // B’s constr. Must be called
  char* f() const { return "D1"; }
};

class D2 : virtual public B {
public:
  D2() : B(2) {}    //B’s constr. Must be called
  char* f() const { return "D2"; }
};

class M : public D1, public D2 {
public:
  M() {}  // B’s constr. Is implicit called
  char* f() const {
    return D1::f();
  }
};
```

```
class X : public M {
public:
  X() {}  // B's constr. Is implicit called
};
```

For a better understanding of the order of the constructors' call, for the hierarchy from the next figure, will be presented the call order of constructors. For simplicity were denoted with *B1()* and *B2()* the constructor's call of the *B* class from the constructors of the classes *D1* and *D2* respectively, and with *B()* the constructor's call of *B* from the *M* class constructor.




```
class D1: public B
class D2: public B (there are 2 sub-objects of B)
```

- `B1(), D1(), B2(), Dd2()`

```
class D1: public B
class D2: virtual public B (there are 2 sub-objects of B)
```

- `Base(), B1(), D1(), D2()`

```
class D1: virtual public B
class D2: public B (there are also 2 sub-objects of B)
```

- `B(), D1(), B2(), D2()`

```
class D1: virtual public B
class D2: virtual public B (there is one sub-object of B)
```

- `B(), D1(), D2()`

For correct managing the instance objects, in derived cases from virtual base classes, supplementary pointers of *VFPTR* type will be used.

Example. For a certain version of the Microsoft Visual C++ compiler. the next program displays the following values: 4, 4, 44, 48, 48, 48, 48, 68.

```
class B {
```

```
public:
     int v[10];
     virtual ~B() {}
};

class D1: virtual public B { };

class D2: virtual public B { };

class D3: virtual public B { };

class D4: virtual public B { };

class M: public D1, public D2, public D3, public D4 {
public:
     int n;
};

int main() {
     B b;
     D1 d1;
     D2 d2;
     D3 d3;
     D4 d4;
     M m;
     cout << "sizeof(int)= " << sizeof(int) << endl;
     cout << "sizeof(void*)= " << sizeof(void*) << endl;
     cout << "sizeof(B)= " << sizeof(b) << endl;
     cout << "sizeof(D1)= " << sizeof(d1) << endl;
     cout << "sizeof(D2)= " << sizeof(d2) << endl;
     cout << "sizeof(D3)= " << sizeof(d3) << endl;
     cout << "sizeof(D4)= " << sizeof(d4) << endl;
     cout << "sizeof(M)= " << sizeof(m) << endl;
     return 0;
}
```

One can observe the fact that for the *B* class objects there is only one *VFPTR* additional pointer (normal operation, because it exists virtual function), for the classes *D1*, *D2*, *D3* and *D4* there are two additionally pointers, while for the objects of the class *M* there are 5 additionally pointers, but here is only one hidden sub-object of the *B* class.

Usually, beside of the vfptr pointer, associated to the current class, there exist for each class derived from the base class a vfptr pointer associated to the respective class, and, in addition, a pointer to the hidden object of the base class. For example, for the class hierarchy from the previous example, the structure of an object of the class *M* is presented in the following figure (usually this structure is dependent to the compiler).

Data members of the class D1
vfptr
Pointer to the hidden object B
Data members of the class D2
vfptr
Pointer to the hidden object B
Data members of the class M
Data members of the class B
vfptr