Intro

"***Polymorphism***" is the term used to describe the process of accessing different implementations of a function via the same name (handle).

- It is a far better mechanism than complicated switch statements
- Polymorphism is highly related to inheritance.
- We will discuss run-time polymorphic behavior (in contrast to simple function overriding).
- It is one of the most important aspects of object-oriented-programming.

Inheritance + virtual functions = polymorphism

- Polymorphism and inheritance go hand in hand.
- We use the special relations between classes of the same hierarchy in order to use the same handle (function name) to activate different implementations of the same function in different classes.
- The key element is to use a base class pointer.

```
#include <iostream.h>

class Base {
public:  virtual void print() {cout<<"calling base\n";}
};

class Derive : public Base {
public:  void print() {cout<<"calling derived\n";}
};

int main() {
   Base bse, *ptr;
   Derive der;
   ptr=&bse;
   ptr->print();
   ptr=&der;
   ptr->print();
   return 0;
}
```

output:
```
calling base
calling derived
```

Explanation:

- We use the keyword "***virtual***" to define a function that we want to implement in different ways in different classes in the hierarchy.
- We use a ***base class pointer*** in order to call the function. Remember that base class pointers can point to derived class objects as well (only "seeing" the base class inherited members).
- The first time, the base class pointer points to a base class object; the straightforward result is activating the **base-class function**.
- The second time we use the same pointer to point to a derived class object; this time, the result is activating the **derived-class function**, even though the base class pointer "sees" only the base class members!!!

Focusing on the second using base class pointer to derived class object:

- The base class pointer accesses the base class function in run-time. The keyword ***virtual*** tells the program to seek a function with the same name in the derived class.
- Still, the call is the same in both cases: `ptr->print();`
- Thus, the program actually chooses the right version of the function.

This is polymorphism – the ability of different objects to respond differently to the same call.

By using polymorphism we can now build into the class hierarchy the ability to deal differently with the same function call – depends on the object (class).

Virtual functions VS simple overriding – dynamic and static binding

Assume that we use a base class pointer to point at a derived class object.
By simply overriding a function, the function call will result in activating the base class function.
Using virtual function, the result will be activating the derived class function.

An example

A regular **function overriding** can be achieved simply by **omitting** the keyword "***virtual***" in the base class :

```
      void print() {cout<<"calling base\n";}
```

while leaving the derived class function definition as is.

```
void print() {cout<<"calling derived\n";}
```

This time, the output will be

```
calling base
calling base
```

In this case, when calling the function via a base class pointer, the pointer "sees" only the base class members, including the base class definition of the print() function. Without the virtual keyword, the program won't look for a different version of this function.
In order to activate the "correct" version of the function we have to use a derived class pointer/object specifically: `bse.print()`. This is ***static binding*** and it is resolved at compile-time.

The use of **virtual function** was described previously; the base class function definition was

```
Virtual void print() {cout<<"calling base\n";}
```

and the derived class function was defined as

```
void print() {cout<<"calling derived\n";}
```

the output was

```
calling base
calling derived
```

The Use of the ***virtual*** keyword enables the program to find the "correct" version of the function in run time. This is called ***dynamic binding***.
We can still use the explicit call `bse.print()` to invoke this function. Again, it's static binding.

Abstract classes & pure virtual functions

A pure virtual function is declared as follows:
virtual *<return_type>* *<function_name>* (*<parameters_list>*) =0;

for example:
```
virtual void printNum( int i )=0;
```

- A class that holds a pure virtual function is called an ***abstract class***.
- There is no "abstract" keyword. A pure virtual function is the "definition" of an abstract class.
- No objects can be instantiated from an abstract class; trying to do so is a compile error.
- A single pure virtual function is enough to make a class abstract.
- An abstract class can still have regular implemented functions (virtual or not). Like any other class, these functions are inheritable.
- Any class that is derived from an abstract class remains abstract until the pure virtual functions are implemented in it.

```
#include <iostream.h>

class Base {
public:  virtual void printNum( int i )=0;
};

class Derive : public Base {
public:  void printNum( int i ) { cout<<"print derived... "<<i<<endl; }
};

int main() {
// Base b;     //an error!!! cannot instantiate from abstract class!!!
   Derive d;
   Base *ptr=&d;
   ptr->printNum(7);
   return 0;
}
```

output:
```
print derived... 7
```

In many hierarchies, the first levels are abstract classes, while the upper levels (for more specifically defined and sophisticated classes) are "regular" classes.

For example, a hierarchy of vehicles with "Vehicle" as the base class and "Ferari" as a derived class way down the hierarchy, may have a function called "print_max_speed()". Since there is no general vehicle – it varies from submarines to space shuttles, cars to motorbikes – this function will remain pure virtual in the hierarchy, down to the "Ferari" class which will print "200 mph". Other classes, for example "Uboat", will print different speeds, like "40 knots".
The "Vehicle" class is an abstract class, while the "Uboat" and "Ferari" are not.

Virtual destructors

- Suppose we created an object from a class that is defined in specific hierarchy. What will happen if we use the "delete" operator on a base class pointer that points at it?
- By doing so, we will always invoke the base class destructor, which is not always wanted.
- The solution is to declare the base class destructor as a virtual function. This way, each time we use the "delete" operator, the "correct" destructor will be invoked. Like always, the base class parts of the derived class objects are also destroyed.

A complete example:

```
#include <iostream.h>

class Line {
public:
   Line( int x ) { length=x; cout<<"line created"<<endl; }
   virtual ~Line() { cout<<"line destroyed"<<endl; }     //virtual destructor
   virtual void printLength() { cout<<"length="<<length<<endl; }
   virtual void printArea()=0;                           //pure virtual function
   virtual void printVolume(){ cout<<"volume=0"<<endl; }
protected:
   int length;
};

class Rect : public Line {
public:
   Rect( int x, int y ) : Line(x) { width=y;  cout<<"rectangle created"<<endl; }
   ~Rect() { cout<<"rectangle destroyed"<<endl; }
   void printArea() { cout<<"area="<<length*width<<endl; }
protected:
   int width;
};

class Box : public Rect {
public:
   Box( int x, int y,int z ) : Rect(x,y){ height=z; cout<<"box created"<<endl; }
   ~Box() { cout<<"box destroyed"<<endl; }
   void printArea() { cout<<"area="<<(2*length*width+4*height*width)<<endl; }
   void printVolume() { cout<<"volume="<<length*width*height<<endl; }
protected:
   int height;

};

int main() {
   Line *ptr = new Box(2,3,4);
   cout<<"\nprinting box properties:"<<endl;
   ptr->printLength();
   ptr->printArea();
   ptr->printVolume();
   cout<<endl;
   delete ptr;
   return 0;
}
```

output:

```
line created
rectangle created
box created

printing box properties:
length=2
area=60
volume=24

box destroyed
rectangle destroyed
line destroyed
```

Remarks:

- The line `virtual ~Line() { cout<<"line destroyed"<<endl; }` specifies that when destroying the box object with the call `delete ptr` the proper destructors are initiated. Omitting the "virtual" keyword would result in output of `line destroyed` only.

- Not using the delete operator at all results in not invoking any constructor!!! The last printed line will read `volume=24`.

- We use the base class pointer (Line *ptr) on a derived class object (Box). By doing so, the proper print functions are called.

- The line `virtual void printArea()=0;` indicates that Line is an abstract class. It also means that in order to instantiate a Box or Rect objects, we will have to supply a printArea() implementation in these classes.

- Protected variables are accessible by members of the same hierarchy.

Choosing to create a Rect object by changing the main() function will have the following results:

```
int main() {
   Line *ptr = new Rect(2,3);
   cout<<"\nprinting rectangle properties:"<<endl;
   ptr->printLength();
   ptr->printArea();
   ptr->printVolume();
   cout<<endl;
   delete ptr;
   return 0;
}
```

output:

```
line created
rectangle created

printing rectangle properties:
length=2
area=6
volume=0

rectangle destroyed
line destroyed
```

Notice the following changes:
- We use the same base class pointer.
- This time, the printVolume() function that is called is the one that was inherited from the base class, and wasn't overridden in the Rect class.
- Again, proper destructors were called – first ~Rect(), then ~Line().

Second example:

```
#include <iostream.h>
#define PI 3.14

class CShape{
public:
   CShape (){};
   virtual ~CShape (){cout<<"Deleting shape"<<endl;};
   virtual void printArea ()=0;
   virtual void printPerimeter()=0;
   virtual  double getPerimeter()=0;
   virtual double getArea()=0;
};

class CCircle:public CShape{
protected:
   double radius;
   double getPerimeter(){return 2*PI*radius;};
   double getArea(){return PI*radius*radius;};
public:
   CCircle (int r){radius=r; };
   ~CCircle (){cout<<"Deleting circle"<<endl; };
   void printArea (){cout<<"Circle area is: "<<getArea()<<endl;}
   void printPerimeter(){cout<<"Circle perimeter is: "<<getPerimeter()<<endl;};
};

class CRectangle:public CShape{
protected:
   double width;
   double height;
   double getPerimeter(){return 2*(width+height);};
   double getArea(){return width*height;};
public:
   CRectangle (double iwidth, double iheight){width=iwidth; height=iheight;};
   ~CRectangle (){cout<<"Deleting rectangle"<<endl; };
   void printArea (){cout<<"Rectangle area is: "<<getArea()<<endl;}
   void printPerimeter()
   {cout<<"Rectangle perimeter is: "<<getPerimeter()<<endl;};
};

class CSquare:public CRectangle{
public:
   CSquare (int a): CRectangle (a,a){};
   ~CSquare (){cout<<"Deleting square"<<endl; };
   void printArea (){cout<<"Square area is: "<<getArea()<<endl;}
   void printPerimeter(){cout<<"Square perimeter is: "<<getPerimeter()<<endl;};
};

int main ()
{
   //CShape s; =>Error !
   //CShape* s = new CShape; =>Error !
   CShape* ShapeArea [3];
   ShapeArea[0]=new CRectangle (3,5);
   ShapeArea[1]=new CSquare (4);
   ShapeArea[2]=new CCircle (7);

   ShapeArea[0]->printArea();
   ShapeArea[1]->printArea();
   ShapeArea[2]->printArea();

   ShapeArea[0]->printPerimeter();
   ShapeArea[1]->printPerimeter();
   ShapeArea[2]->printPerimeter();

   for (int i=0; i<3;i++){delete ShapeArea[i];}

   return 0;
}
```

output:
```
Rectangle area is: 15
Square area is: 16
Circle area is: 153.86
Rectangle perimeter is: 16
Square perimeter is: 16
Circle perimeter is: 43.96

Deleting rectangle
Deleting shape

Deleting square
Deleting rectangle
Deleting shape

Deleting circle
Deleting shape
```

Another complete example in figure 10.2 from the “C++ how to program” book