MODULE 17
POLYMORPHISM

My Training Period: hours

Abilities

Able to understand and use:

- Polymorphism concept.
- Virtual function.
- Late and early binding.

17.1 Introduction

- Polymorphism is a technique that allows you to set a base object equal to one or more of its derived objects.
- The interesting thing about this technique is that, after the assignment, the base acts in different ways, depending on the traits of the derived object that is currently assigned to it. The base object that acts in many different ways, hence the name "**polymorphism**," which translates literally to "**many form**."
- Another way of looking at polymorphism: A base class defines a certain number of functions that are inherited by all its descendants. If you assign a variable of the derived type to one of its base, all the base's methods are guaranteed to be filled out with valid addresses of the pointers.
- The issue here is that the derived object, by the fact of its being a descendant object, must have valid addresses for all the methods used in its base's **virtual method table** (`VMT`). As a result, you can call one of these methods and watch as the derived functions get called.
- However, you cannot call one of the derived methods that do not also belong to the base. The base doesn't know about those methods, so the compiler won't let you call them. In other words, the base may be able to call some of the derive functions, but it is still a variable of the base type.
- A virtual method table, or `VMT`, is a **table** maintained in memory by the compiler; it contains a list of all **the pointers to the virtual methods hosted by an object**. If you have an object that is descended from, let say, `TObject`, the `VMT` for that object will contain all the virtual methods of that object, plus the virtual methods of `TObject`.
- If some of the methods in a base class are defined as `virtual`, each of the descendants can redefine the implementation of these methods. The key elements that define a typical case of polymorphism are a base class and the descendants that inherit a base class's methods. In particular, the fanciest type of polymorphism involves virtual methods that are inherited from a base class.

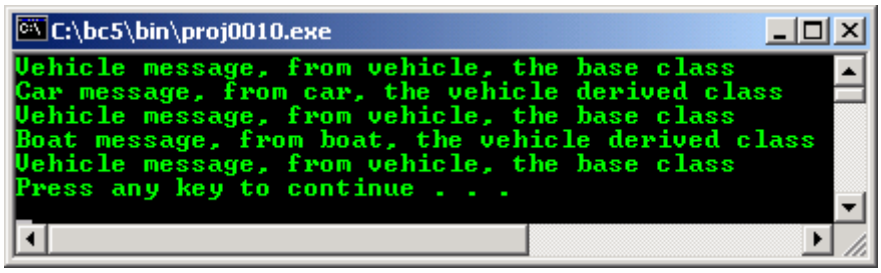
17.2 A Simple Program With Inheritance

- Examine the program example named `poly1.cpp`, the basic program that will be use for our discussion in this Module. The last program in this Module will illustrate the proper use of virtual functions.

```
1.  //Program poly1.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //---base class declaration
6.  //---and implementation part-----
7.  class   vehicle
8.  {
9.     int    wheels;
10.    float  weight;
11.    public:
12.    void  message(void)  //first message()
13.    {cout<<"Vehicle message, from vehicle, the base class\n";}
14. };
15.
16. //----derived class declaration and implementation part-----
17. class  car : public  vehicle
18. {
19.    int   passenger_load;
20.    public:
21.    void   message(void)    //second message()
22.    {cout<<"Car message, from car, the vehicle derived class\n";}
23. };
24.
```

```
25. class  truck : public  vehicle
26. {
27.    int  passenger_load;
28.    float   payload;
29.    public:
30.    int  passengers(void) {return  passenger_load;}
31. };
32.
33. class  boat : public  vehicle
34. {
35.    int  passenger_load;
36.    public:
37.    int  passengers(void) {return  passenger_load;}
38.    void  message(void)     //third message()
39.    {cout<<"Boat message, from boat, the vehicle derived class\n";}
40. };
41.
42. //------------the main program------------
43. int  main()
44. {
45.   vehicle  unicycle;
46.   car      sedan_car;
47.   truck    trailer;
48.   boat     sailboat;
49.
50.   unicycle.message();
51.   sedan_car.message();
52.   trailer.message();
53.   sailboat.message();
54.
55.      //base and derived object assignment
56.      unicycle = sedan_car;
57.      unicycle.message();
58.
59.   system("pause”);
60.   return 0;
61. }
```

61 Lines: Output:



- This program is greatly simplified in order to effectively show you the use of a **virtual function**. You will notice that many of the methods from the last Module have been completely dropped from this example for simplicity, and a new method has been added to the base class, the method named **`message()`** in line 12 as shown below.

```
void  message(void)  //first message()
```

- Throughout this Module we will be studying the operation of the method named **`message()`** in the base class and the derived classes. For that reason, there is another method named **`message()`** in the derived **`car`** class and **`boat`** in lines 21 and 38 respectively as shown below:

```
void   message(void)    //second message()
…
void  message(void)     //third message()
```

- You will also notice that there is no method named **`message()`** in the **`truck`** class. This has been done on purpose to illustrate the use of the virtual function/method. You will recall that the method named **`message()`** from the base class is available in the **`truck`** class because the method from the base class is inherited with the keyword **`public`** included in line 25 as shown below.

```
class  truck : public  vehicle
```

- The main program is as simple as the classes; one object of each of the classes is defined in lines 45 through 48 as shown below.

```
vehicle  unicycle;
car      sedan_car;
truck    trailer;
boat     sailboat;
```

- And the method named **`message()`** is called once for each object. The output of this program indicates that the method for each is called except for the object named **`trailer`**, which has no method named **`message()`**.
- The method named **`message()`** from the base class is called and the data output to the screen indicates that this did happen.
- Line 56 as shown below indicates how the derived object has been assigned to the base object,

```
unicycle = sedan_car;
```

- And then calls the base object again in line 57 as shown below.

```
unicycle.message();
```

- We are not concern with the data, so all the data is allowed to the default **`private`** type and none is inherited into the derived classes. Some of the data is left in the program example simply to make the classes look like classes.
- The data could be removed since it is not used. Compile and run this program to see if your compiler gives the same result of execution.

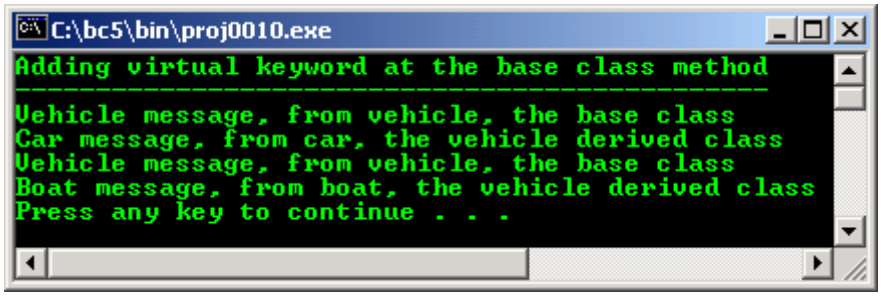
17.3 Adding The Keyword `virtual`

- Examine the next program example named `poly2.cpp`, you will notice that there is one small change in line 12. The keyword **`virtual`** has been added to the declaration of the method named **`message()`** in the base class.

```
1.  //Program poly2.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //---base class declaration
6.  //---and implementation part-----
7.  class   vehicle
8.  {
9.     int    wheels;
10.    float  weight;
11.    public:
12.    virtual void  message(void)
13.    //first message(), with virtual keyword
14.    {cout<<"Vehicle message, from vehicle, the base class\n";}
15. };
16.
17. //----derived class declaration and implementation part-----
18. class  car : public  vehicle
19. {
20.    int   passenger_load;
21.    public:
22.    void   message(void)    //second message()
23.    {cout<<"Car message, from car, the vehicle derived class\n";}
24. };
25.
26. class  truck : public  vehicle
27. {
28.    int  passenger_load;
29.    float   payload;
30.    public:
31.    int  passengers(void) {return  passenger_load;}
32. };
33.
34. class  boat : public  vehicle
35. {
36.    int  passenger_load;
37.    public:
38.    int  passengers(void) {return  passenger_load;}
39.    void  message(void)     //third message()
```

```
40.    {cout<<"Boat message, from boat, the vehicle derived class\n";}
41. };
42.
43. //------------the main program------------
44. int   main()
45. {
46.     vehicle   unicycle;
47.     car       sedan_car;
48.     truck     trailer;
49.     boat      sailboat;
50.
51.    cout<<"Adding virtual keyword at the base class method\n";
52.    cout<<"-----------------------------------------------\n";
53.    unicycle.message();
54.    sedan_car.message();
55.    trailer.message();
56.    sailboat.message();
57.
58.     //unicycle = sedan_car;
59.     //sedan_car.message();
60.
61.    system("pause");
62.    return  0;
63. }
```

63 Lines: Output:



- But this program operates no differently than the last program example. This is because we are using objects **directly** and virtual methods have nothing to do with objects, **only with pointers to objects** as we will see soon.
- There is an additional comment in line `59` and `60` as shown below:

```
//unicycle = sedan_car;
//sedan_car.message();
```

- Illustrating that since all four objects is of different classes, it is impossible to assign any object to any other object in this program with different result. We will soon see that some pointer assignments are permitted between objects of dissimilar classes.
- Compile and run this program example to see if your compiler results in the same output as shown.

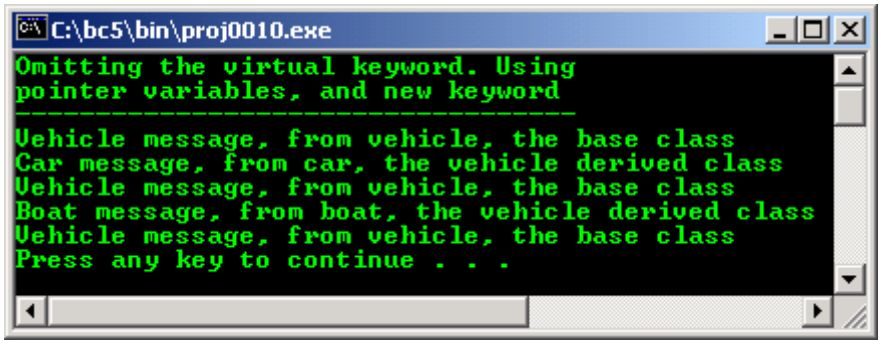
17.4 Using Object Pointers

- Examine the program example named `poly3.cpp` and you will find a repeat of the first program but with a different main program.

```
1.  //Program poly3.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //---base class declaration
6.  //---and implementation part-----
7.  class   vehicle
8.  {
9.     int    wheels;
10.    float  weight;
11.    public:
12.    void  message(void)
13.    //first message()
14.    {cout<<"Vehicle message, from vehicle, the base class\n";}
15. };
16.
17. //----derived class declaration and implementation part-----
```

```
18. class  car : public  vehicle
19. {
20.    int   passenger_load;
21.    public:
22.    void   message(void)    //second message()
23.    {cout<<"Car message, from car, the vehicle derived class\n";}
24. };
25.
26. class  truck : public  vehicle
27. {
28.    int  passenger_load;
29.    float   payload;
30.    public:
31.    int  passengers(void) {return  passenger_load;}
32. };
33.
34. class  boat : public  vehicle
35. {
36.    int  passenger_load;
37.    public:
38.    int  passengers(void) {return  passenger_load;}
39.    void  message(void)     //third message()
40.    {cout<<"Boat message, from boat, the vehicle derived class\n";}
41. };
42.
43. //------------the main program------------
44. int   main()
45. {
46.     vehicle *unicycle;
47.     car     *sedan_car;
48.     truck   *trailer;
49.     boat    *sailboat;
50.
51.     cout<<"Omitting the virtual keyword. Using\n";
52.     cout<<"pointer variables, and new keyword\n";
53.     cout<<"-----------------------------------\n";
54.
55.     unicycle = new vehicle;
56.     unicycle->message();
57.     sedan_car = new car;
58.     sedan_car->message();
59.     trailer = new truck;
60.     trailer->message();
61.     sailboat = new boat;
62.     sailboat->message();
63.
64.     unicycle = sedan_car;
65.     unicycle->message();
66.
67.
68.   system("pause");
69.     return  0;
70. }
```

70 Lines: Output:



- In this program the keyword **`virtual`** has been removed from the method declaration in the base class in line 12, and the main program defines pointers to the objects rather than defining the objects themselves in lines 46 through 49 as shown below:

```
vehicle *unicycle;
car     *sedan_car;
truck   *trailer;
```

```
boat    *sailboat;
```

- Since we only defined pointers to the objects, we find it necessary to allocate the objects before using them by using the **new** operator in lines `55, 57, 59` and `61` as shown below:

```
unicycle = new vehicle;
...
sedan_car = new car;
...
trailer = new truck;
...
sailboat = new boat;
```

- Upon running the program, we find that even though we are using pointers to the objects, we have done nothing different than what we did in the first program.
- The program operates in exactly the same manner as the first program example. This should not be surprising because a pointer to a method can be used to operate on an object in the same manner as an object can be directly manipulated.
- Be sure to compile and run this program before continuing on to the next program example. In this program you will notice that we failed to check the allocation to see that it did allocate the objects properly, and we also failed to de-allocate the objects prior to terminating the program.
- In such a simple program, it doesn't matter because the heap will be cleaned up automatically when we return to the operating system.
- In real program development you have to implement this allocation checking and the de allocation. As shown in the previous Module, if we do not de allocate, there will be garbage left.

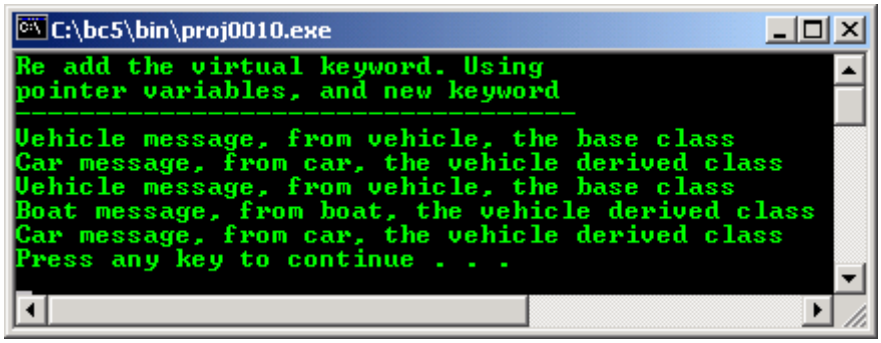
17.5 A Pointer And A Virtual Function

- The program example named `poly4.cpp` is identical to the last program except for the addition of the keyword **`virtual`** to line `12` once again.

```
1.  //Program poly4.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //---base class declaration
6.  //---and implementation part-----
7.  class   vehicle
8.  {
9.     int    wheels;
10.    float  weight;
11.    public:
12.    virtual void  message(void)
13.    //first message(), with virtual keyword
14.    {cout<<"Vehicle message, from vehicle, the base class\n";}
15. };
16.
17. //----derived class declaration and implementation part-----
18. class  car : public  vehicle
19. {
20.    int   passenger_load;
21.    public:
22.    void   message(void)    //second message()
23.    {cout<<"Car message, from car, the vehicle derived class\n";}
24. };
25.
26. class  truck : public  vehicle
27. {
28.    int  passenger_load;
29.    float   payload;
30.    public:
31.    int  passengers(void) {return  passenger_load;}
32. };
33.
34. class  boat : public  vehicle
35. {
36.    int  passenger_load;
37.    public:
38.    int  passengers(void) {return  passenger_load;}
39.    void  message(void)     //third message()
40.    {cout<<"Boat message, from boat, the vehicle derived class\n";}
41. };
42.
43. //------------the main program------------
```

```
44. int   main()    //main program
45. {
46.     vehicle *unicycle;
47.     car     *sedan_car;
48.     truck   *trailer;
49.     boat    *sailboat;
50.
51.     cout<<"Re add the virtual keyword. Using\n";
52.     cout<<"pointer variables, and new keyword\n";
53.     cout<<"-----------------------------------\n";
54.
55.     unicycle = new vehicle;
56.     unicycle->message();
57.        sedan_car = new car;
58.        sedan_car->message();
59.     trailer = new truck;
60.     trailer->message();
61.        sailboat = new boat;
62.        sailboat->message();
63.
64.     unicycle = sedan_car;
65.     unicycle->message();
66.
67.     system("pause");
68.     return  0;
69. }
```

69 Lines: Output:



- By including the keyword **virtual**, it is still identical to the last program. Once again we are simply using pointers to each of the objects, and in every case the pointer is of the same type as the object to which it points. You will begin to see some changes in the next program example.
- Please compile and run this program. The four previous programs were meant just to show to you in what virtual functions do not do. The next two will show you what virtual functions do.

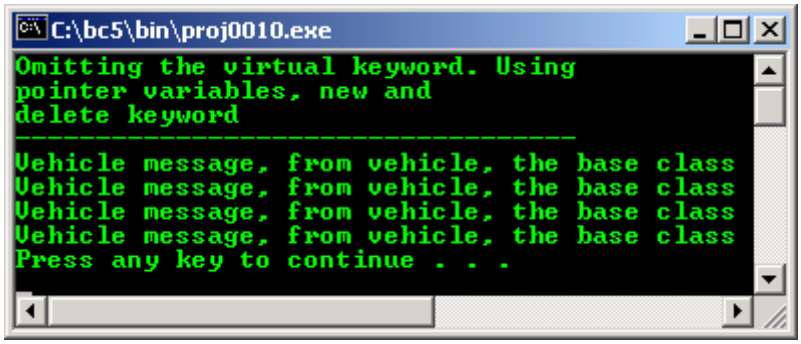
17.6 A Single Pointer To The Parent Class

- Examine the program example named `poly5.cpp` where we almost use a virtual function. We are almost ready to use a virtual method.

```
1.  //Program poly5.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //---base class declaration
6.  //---and implementation part-----
7.  class   vehicle
8.  {
9.     int    wheels;
10.    float  weight;
11.    public:
12.    void  message(void)
13.    //first message()
14.    {cout<<"Vehicle message, from vehicle, the base class\n";}
15. };
16.
17. //----derived class declaration and implementation part-----
18. class  car : public  vehicle
19. {
20.    int   passenger_load;
21.    public:
```

```
22.    void   message(void)    //second message()
23.    {cout<<"Car message, from car, the vehicle derived class\n";}
24. };
25.
26. class  truck : public  vehicle
27. {
28.    int  passenger_load;
29.    float   payload;
30.    public:
31.    int  passengers(void) {return  passenger_load;}
32. };
33.
34. class  boat : public  vehicle
35. {
36.    int  passenger_load;
37.    public:
38.    int  passengers(void) {return  passenger_load;}
39.    void  message(void)     //third message()
40.    {cout<<"Boat message, from boat, the vehicle derived class\n";}
41. };
42.
43. //------------the main program------------
44. int   main()
45. {
46.
47.     cout<<"Omitting the virtual keyword. Using\n";
48.     cout<<"pointer variables, new and\n";
49.     cout<<"delete keyword\n";
50.     cout<<"-----------------------------------\n";
51.
52.     vehicle   *unicycle;
53.     unicycle = new vehicle;
54.     unicycle->message();
55.      delete unicycle;
56.
57.     unicycle = new car;
58.     unicycle->message();
59.       delete unicycle;
60.
61.    unicycle = new truck;
62.    unicycle->message();
63.       delete unicycle;
64.
65.    unicycle = new boat;
66.    unicycle->message();
67.      delete unicycle;
68.
69.     //unicycle = sedan_car;
70.     //unicycle->message();
71.
72.    system("pause");
73.    return  0;
74. }
```

74 Lines: Output:



- The keyword **`virtual`** omitted again in line 12 and with a totally different main program. In this program, we only define a single pointer to a class and the pointer is pointing to the base class of the class hierarchy. We will use the single pointer to refer to each of the four classes and observe what the output of the method named **`message()`** is.

- If we referred to a vehicle (in the real world, not necessarily in this program), we could be referring to a car, a truck, a motorcycle, or any other kinds of transportation, because we are referring to a very general form of an object.
- If however, we were to refer to a car, we are excluding trucks, motorcycles, and all other kinds of transportation, because we are referring to a car specifically. The more general term of vehicle can therefore refer to a many kinds of vehicles, but the more specific term of car can only refer to a single kind of vehicle, namely a car.
- We can apply the same thought process in C++ and say that if we have a pointer to a **vehicle**, we can use that pointer to refer to any of the more specific objects whereas if we have a pointer to a **car**, we cannot use that pointer to reference any of the other classes including the **vehicle** class because the pointer to the **car** class is too specific and restricted to be used on any other classes.

17.7 C++ Pointer Rule

- A pointer declared as pointing to a base class can be used to point to an object of a derived class of that base class, but a pointer to a derived class cannot be used to point to an object of the base class or to any of the other derived classes of the base class.
- In our program therefore, we are allowed to declare a pointer to the **vehicle** class which is the base class, and use that pointer to refer to objects of the base class or any of the derived classes.
- This is exactly what we do in the main program. We define a single pointer which points to the **vehicle** class and use it to point to objects of each of the classes in the same order as in the last four programs. In each case, we allocate the object, send a message to the method named **message()** and de-allocate the object before going on to the next class.
- You will notice that when we send the four messages, we are sending the message to the same method, named **message()** which is a part of the **vehicle** base class. This is because the pointer has a class associated with it. Even though the pointer is actually pointing to four different classes in this program, the program acts as if the pointer is always pointing to an object of the base class because the pointer is of the type of the base class.

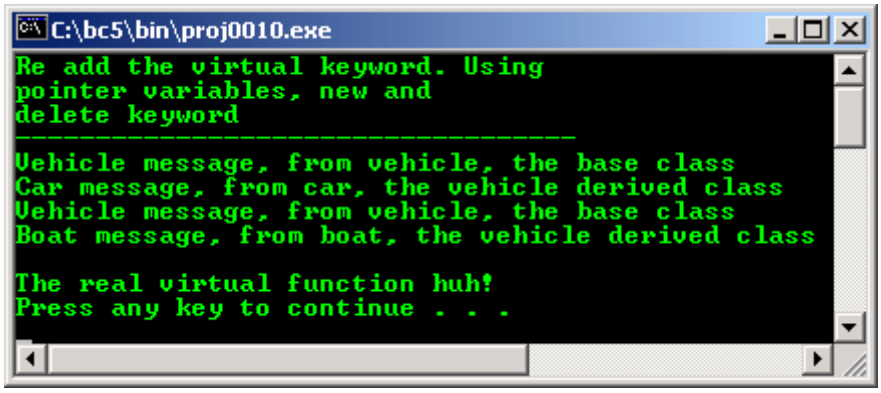
17.8 An Actual Virtual Function

- We finally come to a program example with a virtual function that operates as a virtual function and exhibits **dynamic binding** or **polymorphism** as it is called.

```
1.  //Program poly6.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //---base class declaration
6.  //---and implementation part-----
7.  class   vehicle
8.  {
9.     int    wheels;
10.    float  weight;
11.    public:
12.    virtual void  message(void)
13.    //first message() method, with virtual keyword
14.    {cout<<"Vehicle message, from vehicle, the base class\n";}
15. };
16.
17. //----derived class declaration and implementation part-----
18. class  car : public  vehicle
19. {
20.    int   passenger_load;
21.    public:
22.    void   message(void)    //second message()
23.    {cout<<"Car message, from car, the vehicle derived class\n";}
24. };
25.
26. class  truck : public  vehicle
27. {
28.    int  passenger_load;
29.    float   payload;
30.    public:
31.    int  passengers(void) {return  passenger_load;}
32. };
33.
34. class  boat : public  vehicle
35. {
36.    int  passenger_load;
```

```
37.    public:
38.    int  passengers(void) {return  passenger_load;}
39.    void  message(void)     //third message()
40.    {cout<<"Boat message, from boat, the vehicle derived class\n";}
41. };
42.
43. //------------the main program------------
44. int   main()
45. {
46.
47.     cout<<"Re add the virtual keyword. Using\n";
48.     cout<<"pointer variables, new and\n";
49.     cout<<"delete keyword\n";
50.     cout<<"-----------------------------------\n";
51.
52.     vehicle  *unicycle;
53.
54.    unicycle = new vehicle;
55.    unicycle->message();
56.       delete unicycle;
57.    unicycle = new car;
58.    unicycle->message();
59.           delete unicycle;
60.    unicycle = new truck;
61.    unicycle->message();
62.           delete unicycle;
63.
64.    unicycle = new boat;
65.    unicycle->message();
66.           delete unicycle;
67.
68.   cout<<"\nThe real virtual function huh!\n";
69.
70.   system("pause");
71.   return  0;
72. }
```

72 Lines: Output:



- This program `poly6.cpp` is identical to the last program example except that the keyword **`virtual`** is added to line `12` to make the method named **`message()`** a virtual function. You will notice that the keyword **`virtual`** only appears in the base class, all classes that derive this class will have the corresponding method **automatically** declared **`virtual`** by the system.
- In this program, we use the single pointer to the base class and allocate, use, then delete an object of each of the four available classes using the identical code we used in the last program. However, because of the addition of the keyword **`virtual`** in line `12`, this program acts entirely different from the last program example.
- Since the method named **`message()`** is declared to be a **`virtual`** function in its declaration in the base class, anytime we refer to this function with a pointer to the base class, we actually execute the function associated with one of the derived classes. But this is true only if there is a function available in the derived class and if the pointer is actually pointing to that derived class.
- When the program is executed, the output reflects the same output we saw in the other cases when we were actually calling the methods in the derived classes, but now we are using a pointer of the **base class type** to make the calls.
- You will notice that in lines `55`, `58`, `61`, and `65`, even though the code is identical in each line, the system is making the decision of which method to actually call based on the type of the pointer when each message is sent. The decision of which method to call **is not made during the time when the**

code is compiled but when the code is executed. This is **dynamic binding** and can be very useful in some programming situations.

- In fact, there are only three different calls made because the class named **`truck`** does not have a method named **`message(),`** so the system simply uses the method from the base class to satisfy the message passed.
- For this reason, a virtual function **must** have an implementation available in the base class which will be used if one is not available in one or more of the derived classes. Note that the message is actually sent to a pointer to the object.
- Notice that the structure of the virtual function in the base class and each of the derived classes is identical. The return type and the number and types of the parameters must be identical for all functions, since a single statement can be used to call any of them.
- If the keyword **`virtual`** is used, the system will use **late binding** (some call it dynamic Binding) which is done at run time, but if the keyword is not included, **early binding** (some call it static binding) will be used. What these words actually mean is that with late binding, the compiler does not know which method will actually respond to the message because the type of the pointer is not known at compile time. With early binding, however, the compiler decides at compile time what method will respond to the message sent to the pointer.
- In real world, the example for the late binding is when the application program calling the `dll` (**dynamic link library**) file(s) during the program execution. The `dll` files don't have the `main()` function and it can be called (shared) by many programs.
- Compile and run this program example.

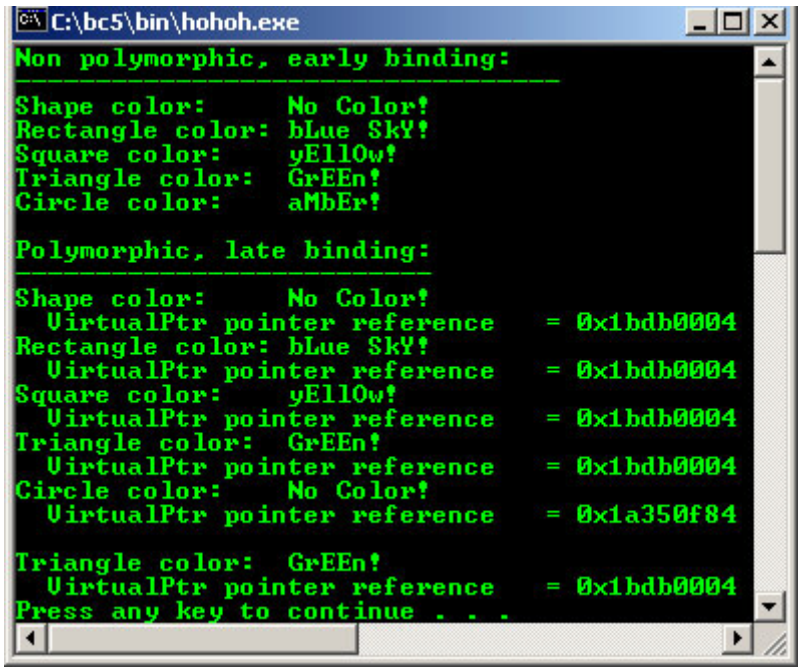
Program Example And Experiment

```
//polymorphic functions, virtual keyword
//program example...
#include <iostream.h>
#include <stdlib.h>

//------class declaration and implementation--------
//base class
class Shape
{
	//protected member variables should be available
	//for derived classes...
	protected:
	char* Color;

	public:
	//constructor, set the object's data
	Shape(){Color = "No Color!";}
	~Shape(){};
	//virtual base member function...
	//return the object's data
	virtual char* GetColor(){return Color;}
};

//derived class...
class Rectangle:public Shape
{
	//notice the same variable name, it is OK...
	char* Color;

	public:
	Rectangle(){Color = "bLue SkY!";}
	~Rectangle(){}
	//derived class member function
	//should also be virtual...
	char* GetColor(){return Color;}
};

class Square:public Shape
{
	char* Color;

	public:
	Square(){ Color = "yEllOw!";}
	~Square(){}
	char* GetColor(){return Color;}
};

class Triangle:public Shape
{
```

```
        char* Color;

        public:
        Triangle(){Color = "GrEEn!";}
        ~Triangle(){}
        char* GetColor(){return Color;}
};

class Circle:public Shape
{
        char* Color;

        public:
        Circle(){Color = "aMbEr!";}
        ~Circle(){}
        //let set different function name but
        //same functionality...
        char* GetMyColor(){return Color;}
};

//-----------main program-----------
int main()
{
        //instantiate objects of class type...
        Shape ObjOne;
        Rectangle ObjTwo;
        Square ObjThree;
        Triangle ObjFour;
        Circle ObjFive;

        cout<<"Non polymorphic, early binding:"<<endl;
        cout<<"-----------------------------------"<<endl;
        cout<<"Shape color:     "<<ObjOne.GetColor()<<endl;
        cout<<"Rectangle color: "<<ObjTwo.GetColor()<<endl;
        cout<<"Square color:    "<<ObjThree.GetColor()<<endl;
        cout<<"Triangle color:  "<<ObjFour.GetColor()<<endl;
        //notice the different function name as previous function...
        cout<<"Circle color:    "<<ObjFive.GetMyColor()<<endl;

        cout<<"\nPolymorphic, late binding:"<<endl;
        cout<<"--------------------------"<<endl;

        //pointer variable of type Shape class...
        Shape *VirtualPtr;

        //object allocation of type Shape size...
        VirtualPtr = new Shape;
        cout<<"Shape color:     "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        //de-allocate, clean up...
        delete VirtualPtr;

        VirtualPtr = new Rectangle;
        cout<<"Rectangle color: "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        VirtualPtr = new Square;
        cout<<"Square color:    "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        VirtualPtr = new Triangle;
        cout<<"Triangle color:  "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        //no GetColor() in this derived class, so use the GetColor
        //from the base class...
        VirtualPtr = new Circle;
        cout<<"Circle color:    "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<&VirtualPtr<<"\n\n";
        delete VirtualPtr;

        //retest..
        VirtualPtr = new Triangle;
        cout<<"Triangle color:  "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;
```

```
        system("pause");
        return 0;
}
```

Output:

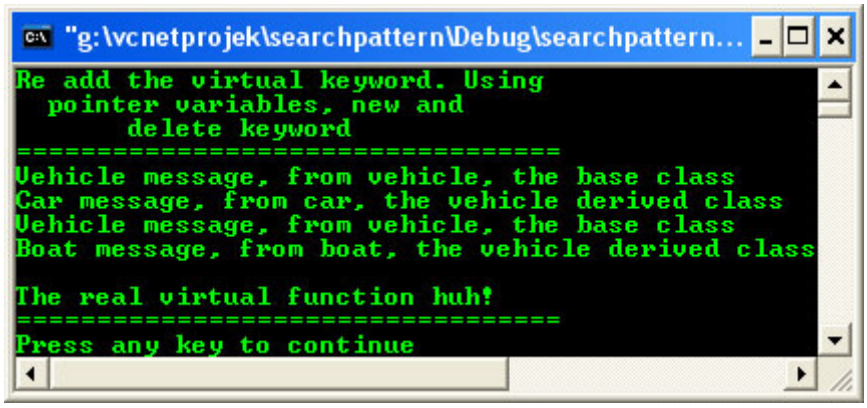


- From the output, notice the memory address (virtual pointer) similarity and one of them has different address.
- Program example compiled using VC++/VC++ .Net.

```
//Program poly6.cpp
#include  <iostream>
using namespace std;

//---base class declaration
//---and implementation part-----
class   vehicle
{
   int    wheels;
   float  weight;
   public:
   virtual void  message(void)
   //first message() method, with virtual keyword
   {cout<<"Vehicle message, from vehicle, the base class\n";}
};

//----derived class declaration and implementation part-----
class  car : public  vehicle
{
   int   passenger_load;
   public:
   void   message(void)    //second message()
   {cout<<"Car message, from car, the vehicle derived class\n";}
};

class  truck : public  vehicle
{
   int  passenger_load;
   float   payload;
   public:
   int  passengers(void) {return  passenger_load;}
};

class  boat : public  vehicle
{
   int  passenger_load;
   public:
   int  passengers(void) {return  passenger_load;}
   void  message(void)     //third message()
```

```
   {cout<<"Boat message, from boat, the vehicle derived class\n";}
};

//------------the main program------------
int   main()
{
    cout<<"Re add the virtual keyword. Using\n";
    cout<<"  pointer variables, new and\n";
    cout<<"       delete keyword\n";
    cout<<"==================================\n";

     vehicle  *unicycle;

   unicycle = new vehicle;
   unicycle->message();
      delete unicycle;
   unicycle = new car;
   unicycle->message();
        delete unicycle;
   unicycle = new truck;
   unicycle->message();
        delete unicycle;

   unicycle = new boat;
   unicycle->message();
        delete unicycle;

  cout<<"\nThe real virtual function huh!\n";
  cout<<"==================================\n";

  return 0;
}
```

Output:



- Previous example compiled using **g++**.

```
///////-polymorph.cpp-/////////
///////-polymorphic functions, virtual function-////////
///////-FEDORA 3, g++ x.x.x-///////
#include <iostream>
using namespace std;

//------class declaration and implementation--------
//base class
class Shape
{
        //protected member variables should be available
        //for derived classes...
        protected:
        char* Color;

        public:
        //constructor, set the object's data
        Shape(){Color = "No Color!";}
        ~Shape(){};
        //virtual base member function...
        //return the object's data
        virtual char* GetColor(){return Color;}
```

```
};

//derived class...
class Rectangle:public Shape
{
        //notice the same variable name, it is OK...
        char* Color;

        public:
        Rectangle(){Color = "bLue SkY!";}
        ~Rectangle(){}
        //derived class member function
        //should also be virtual...
        char* GetColor(){return Color;}
};

class Square:public Shape
{
        char* Color;

        public:
        Square(){Color = "yEllOw!";}
        ~Square(){}
        char* GetColor(){return Color;}
};

class Triangle:public Shape
{
        char* Color;

        public:
        Triangle(){Color = "GrEEn!";}
        ~Triangle(){}
        char* GetColor(){return Color;}
};

class Circle:public Shape
{
        char* Color;

        public:
        Circle(){Color = "aMbEr!";}
        ~Circle(){}
        //let set different function name but
        //same functionality...
        char* GetMyColor(){return Color;}
};

//-----------main program-----------
int main()
{
        //instantiate objects of class type...
        Shape ObjOne;
        Rectangle ObjTwo;
        Square ObjThree;
        Triangle ObjFour;
        Circle ObjFive;

        cout<<"Non polymorphic, early binding:"<<endl;
        cout<<"---------------------------------"<<endl;
        cout<<"Shape color:     "<<ObjOne.GetColor()<<".  ";
        cout<<" The address-->"<<&ObjOne<<endl;
        cout<<"Rectangle color: "<<ObjTwo.GetColor()<<". ";
        cout<<" The address-->"<<&ObjTwo<<endl;
        cout<<"Square color:    "<<ObjThree.GetColor()<<". ";
        cout<<" The address-->"<<&ObjThree<<endl;
        cout<<"Triangle color:  "<<ObjFour.GetColor()<<". ";
        cout<<" The address-->"<<&ObjFour<<endl;
        //notice the different function name as previous function...
        cout<<"Circle color:    "<<ObjFive.GetMyColor()<<". ";
        cout<<"The address-->"<<&ObjFive<<endl;

        cout<<"\nPolymorphic, late binding:"<<endl;
        cout<<"--------------------------"<<endl;

        //pointer variable of type Shape class...
        Shape *VirtualPtr;

        //object allocation of type Shape size...
        VirtualPtr = new Shape;
```

```
        cout<<"Shape color:     "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        //deallocate, clean up...
        delete VirtualPtr;

        VirtualPtr = new Rectangle;
        cout<<"Rectangle color: "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        VirtualPtr = new Square;
        cout<<"Square color:    "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        VirtualPtr = new Triangle;
        cout<<"Triangle color:  "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        //no GetColor() in this derived class, so use the GetColor
        //from the base class...
        VirtualPtr = new Circle;
        cout<<"Circle color:    "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<&VirtualPtr<<"\n\n";
        delete VirtualPtr;

        //retest...
        VirtualPtr = new Triangle;
        cout<<"Triangle color:  "<<VirtualPtr->GetColor()<<endl;
        cout<<"  VirtualPtr pointer reference   = "<<VirtualPtr<<endl;
        delete VirtualPtr;

        return 0;
}
```

[bodo@bakawali ~]$ g++ polymorph.cpp -o polymorph
[bodo@bakawali ~]$ ./polymorph

```
Non polymorphic, early binding:
---------------------------------
Shape color:     No Color!.   The address-->0xbffffa80
Rectangle color: bLue SkY!.  The address-->0xbffffa70
Square color:    yEllOw!.  The address-->0xbffffa60
Triangle color:  GrEEn!.  The address-->0xbffffa50
Circle color:    aMbEr!. The address-->0xbffffa40

Polymorphic, late binding:
--------------------------
Shape color:     No Color!
  VirtualPtr pointer reference   = 0x804b008
Rectangle color: bLue SkY!
  VirtualPtr pointer reference   = 0x804b008
Square color:    yEllOw!
  VirtualPtr pointer reference   = 0x804b008
Triangle color:  GrEEn!
  VirtualPtr pointer reference   = 0x804b008
Circle color:    No Color!
  VirtualPtr pointer reference   = 0xbffffa3c

Triangle color:  GrEEn!
 VirtualPtr pointer reference   = 0x804b008
```

Further reading and digging:

1. Check the best selling C/C++, Object Oriented and pattern analysis books at Amazon.com.
2. Subjects, topics or books related to the following items:

    a. Object Oriented Analysis.
    b. Object Oriented Design.
    c. Unified Modeling Language (UML).