MODULE 12
CLASS - ENCAPSULATION I

My Training Period: hours

Notes:
Starting from this Module, you have to be careful for the source codes that **span more than one line**. When you copy and paste to the text or compiler editor, make it in one line! This Module is a transition from C to C++ and topics of C++ such as Functions, Arrays, Pointers and Structure that have been discussed in C (Tutorial #1 and #2) will not be repeated. They are reusable!
This Module and that follows can be a very good fundamental for Object Oriented programming though it is an old story :o).

Abilities

- To understand the basic principle of encapsulation.
- To understand the principle of data hiding.
- To understand and use class, object, object instance and message.
- To understand and use keyword `public` and `private`.
- To understand and use constructor and destructor.
- To understand and use `inline` function.
- To understand and use object packaging.

12.1 Introduction

- This Module is the beginning of the definition of objects oriented programming of C++. Basically, encapsulation is **the process of forming objects**. It is container, which can only be accessed through certain entry points in controlled manner.
- An encapsulated object is often called an **abstract data type** (ADT). Without encapsulation, which involves the use of one or more **classes**, it is difficult to define the object oriented programming.
- We need encapsulation because we are human, and humans make errors. When we properly encapsulate some code, we actually build protection for the contained code from accidental corruption due to the errors that we are all prone to make.
- We also tend to isolate errors to small portions of code to make them easier to find and fix. Furthermore, programming becomes more efficient, productive and faster program development cycle by dividing and creating smaller modules or program portions, then combine in a systematic processes.
- You will find a lot of readily available classes in Java, Visual Basic®, Microsoft Foundation Classes (MFC) of Visual C++ and other visual programming languages.
- In visual programming languages, you have to learn how to use the classes, which file to be included in your program etc. For non technical programmer, it is much easier to learn programming by using visual programming languages isn't it? You decide!

12.2 Starting With `struct`

- As the beginning please refer to program `start.cpp`. This program will be the starting point for our discussion of encapsulation.
- In this program, a very simple structure is defined in lines `5` through `8` which contain a single `int` type variable within the structure.

```
struct   item //struct data type
   {
        int keep_data;
   };
```

- Three variables are declared in line `12`, each of which contains a single `int` type variable and each of the three variables are available for use anywhere within the `main()` function.

```
item   John_cat, Joe_cat, Big_cat;
```

- Each variable can be assigned, incremented, read, modified, or have any number of operations performed on it and a few of the operations are illustrated in lines `15` through `17`. Notice the use of the dot operator to access the structure element.

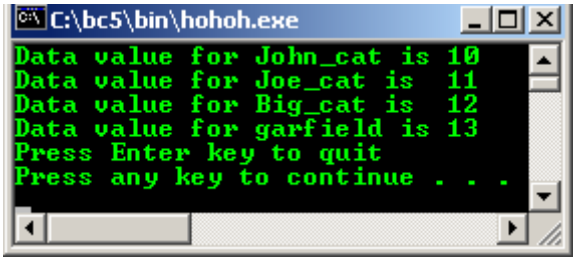
```
John_cat.keep_data = 10;      //assigning data
```

```
Joe_cat.keep_data = 11;
Big_cat.keep_data = 12;
```

- An isolated normal local variable named `garfield` is also declared and used in the same section of code for comparison of the normal variable.
- Study this program example carefully, then compile and run.

```
1.  //Program start.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.   struct   item //struct data type
6.     {
7.       int  keep_data;
8.      };
9.
10.  void main()
11.   {
12.    item   John_cat, Joe_cat, Big_cat;
13.    int   garfield;   //normal variable
14.
15.    John_cat.keep_data = 10;      //assigning data
16.    Joe_cat.keep_data = 11;
17.    Big_cat.keep_data = 12;
18.    garfield = 13;
19.
20.   //Displaying data
21.   cout<<"Data value for John_cat is "<<John_cat.keep_data<<"\n";
22.   cout<<"Data value for Joe_cat is  "<<Joe_cat.keep_data <<"\n";
23.   cout<<"Data value for Big_cat is  "<<Big_cat.keep_data<<"\n";
24.   cout<<"Data value for garfield is "<<garfield<<"\n";
25.   cout<<"Press Enter key to quit\n";
26.   system("pause"); //just for screen snapshot
27.   }
```

27 lines:Output:



12.3 Changing To `class`

- Next, please refer to example program `class.cpp`. This program is identical to the last one except for a few program portions.
- The first difference is that we have a **class** instead of a **structure** beginning in line 7.

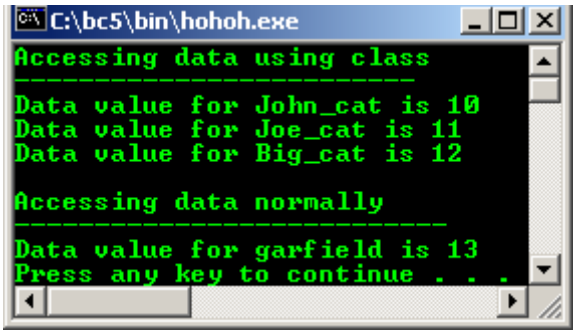
```
class item
```

- The **only** difference between a class and a structure is that a class begins with a `private` section by default whereas a structure begins with a `public` section. The keyword **`class`** is used to declare a class as illustrated.
- The class named `item` is composed of the single variable named `keep_data` and two functions, one named `set()` and the other named `get_value()`.
- A more complete definition of a class is: a group of variables (data), and one or more functions that can operate on that data.
- In programming language terms, **attributes**, **behaviors** or **properties** of the **object** used for the member variables.

```
1.  //Program class.cpp using class instead of struct
2.  #include <iostream.h>
3.  #include <stdlib.h>
4.
5.  //--------Class declaration part------------
6.
```

```
7.  class item
8.   {
9.       int  keep_data;    //private by default, it is public in struct
10.      public:       //public part
11.      void  set(int enter_value);
12.      int  get_value(void);
13.  };
14.
15. //--------Class implementation part---------
16.
17.  void  item::set(int enter_value)
18.    {
19.      keep_data = enter_value;
20.    }
21.  int  item::get_value(void)
22.    {
23.      return  keep_data;
24.   }
25.
26. //-------main program------------
27. void main()
28.   {
29.     item    John_cat,  Joe_cat,  Big_cat;
30.      //three objects instantiated
31.     int   garfield;    //normal variable
32.
33.     John_cat.set(10);   //assigning data
34.     Joe_cat.set(11);
35.     Big_cat.set(12);
36.     garfield = 13;
37.   //John_cat.keep_data = 100;
38.   //Joe_cat.keep_data = 110;
39.   //This is illegal cause keep_data now, is private by default
40.
41. cout<<"Accessing data using class\n";
42. cout<<"-------------------------\n";
43. cout<<"Data value for John_cat is "<<John_cat.get_value()<<"\n";
44. cout<<"Data value for Joe_cat is "<<Joe_cat.get_value()<<"\n";
45. cout<<"Data value for Big_cat is "<<Big_cat.get_value()<<"\n";
46. cout<<"\nAccessing data normally\n";
47. cout<<"--------------------------\n";
48. cout<<"Data value for garfield is "<<garfield<<"\n";
49.
50. system("pause");
51.   }
```

51 Lines:Output:



12.3.1 Private Section

- All data at the beginning of a class defaults to `private`. This means, the data at the beginning of the class cannot be accessed from outside of the class; it is hidden from any outside access.
- Therefore, the variable named `keep_data` which is part of the object named `John_cat` defined in line 37 and 38, is not available for use anywhere in the `main()` program. That is why we have to comment out the following codes:

```
//John_cat.keep_data = 100;
//Joe_cat.keep_data = 110;
```

- It is as if we have built a wall around the variables to protect them from accidental corruption by outside programming influences.

- The concept is graphically shown in figure 12.1, `item class` with its wall built around the data to protect it.
- You will notice the small peep holes (through the arrow) we have opened up to allow the user to gain access to the functions `set()` and `get_value()`. The peep holes were opened by declaring the functions in the public section of the class.

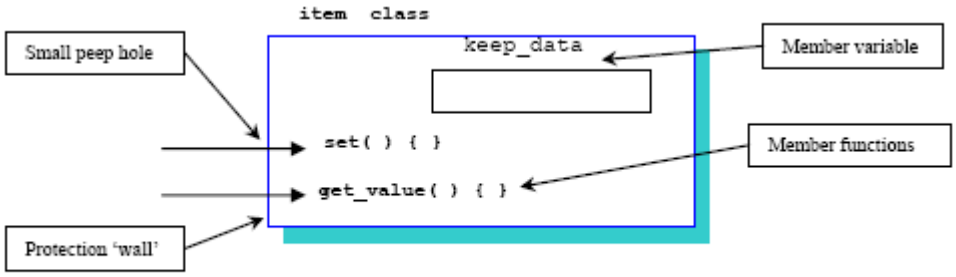


Figure 12.1: Graphically `item` class representation

12.3.2 Public Section

- A new keyword `public`, introduced in line `10` which states that anything following this keyword can be accessed from outside of this class as shown below:

```
public:   //public part
```

- Because the two functions are declared following the keyword `public`, they are both public and available for use by any calling program that is **within the scope of this object**.
- This essentially opens two small peepholes in the solid wall of protection that we built around the class and the private `keep_data` variable is not available to the **calling program**.
- Thus, we can only use the variable by calling one of the two functions defined within the public part of the class. These are called **member functions** because they are members of the class.
- Since we have two functions, we need to define them by saying what each function will actually do. This is done in lines `17` through `24` where they are each define in the normal way, except that the class name is prepended onto the function name and separated from it by a **double colon** (**::**), called **scope operator** as shown below:

```
void item::set(int enter_value)
{
     keep_data = enter_value;
}

int item::get_value(void)
{
     return keep_data;
}
```

- These two **function definitions are called the implementation of the functions**. The class name is required because we **can use** the same function name in other classes and the compiler will know with which class to associate each function implementation.
- Notice that, the private data contained within the class is available within the implementation of the member functions of the class for modification or reading in the normal manner.
- You can do anything with the private data within the function implementations which are a part of that class; also the private data of other classes is hidden and not available within the member functions of this class.
- This is the reason we must prepend the **class name** to the **function names** of this class when defining them. Figure 12.2 depicts the data space following the program execution.
- It is legal to declare **variables** and **functions** in the `private` part, and additional variables and functions in the `public` part also.
- In most practical situations, variables only declared in the `private` section and functions only declared in the `public` part of a class definition. Occasionally, variables or functions are declared in the other part. This sometimes leads to a very practical solution to a particular problem, but in general, the entities are used only in the places mentioned for consistency and good programming style.

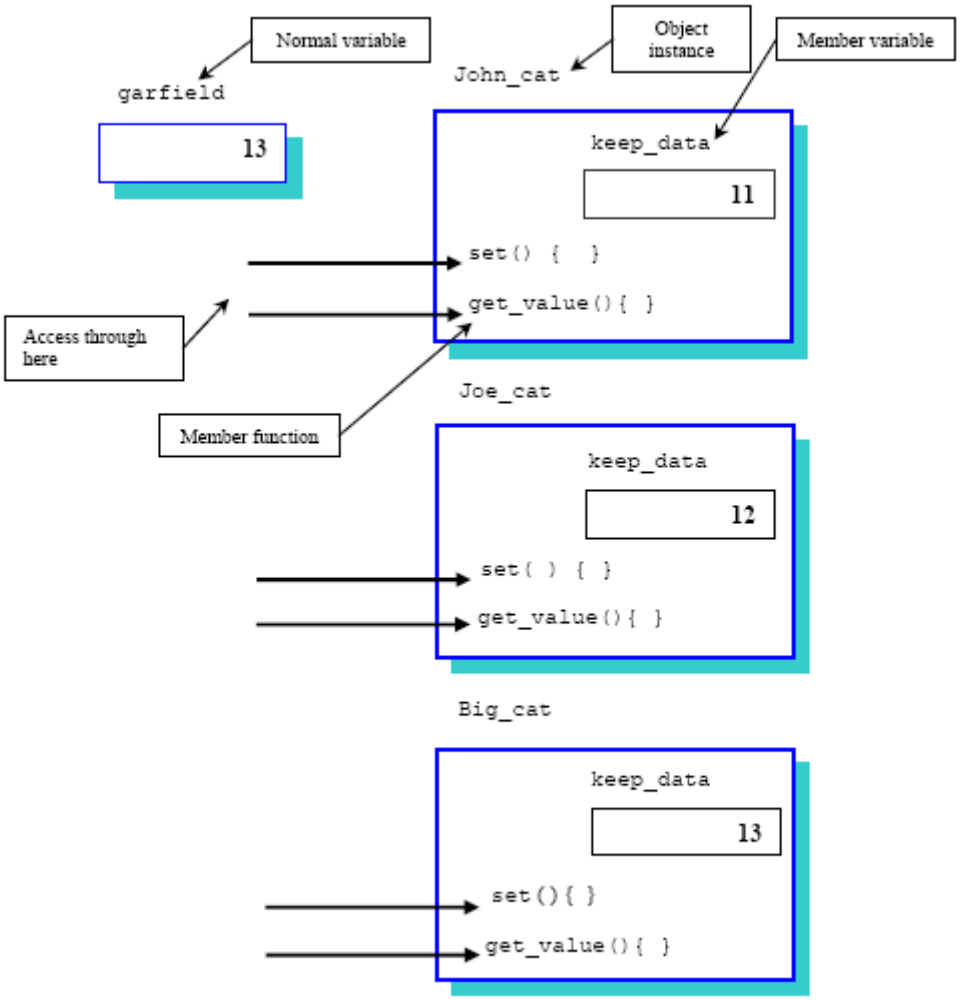


Figure 12.2: Graphically representation of the class `item` data space after the program execution

- A variable with **class** scope is available anywhere within the scope of a class, including the implementation code, and nowhere else. Hence, the variable named `keep_data` has a class scope.

12.3.3 Some Terminologies

- The following a list of terminologies that you need to understand their meaning in object oriented programming.

Term	Description
`class`	Is a group of data and methods (functions). A class is very much like a structure type as used in ANSI-C, it is just a type used to create a variable which can be manipulated through method in a program.
`object`	Is an instance of a class, which is similar to a variable, defined as an instance of a type. An object is what you actually use in a program since it contains values and can be changed.
`method`	Is a function contained within the class. You will find the functions used within a class often referred to as methods in programming literature.
`message`	Is similar to function call. In object oriented programming, we send messages instead of calling functions. For the time being, you can think of them as identical. Later you will see that they are in fact slightly different. In programming terms, event or action of the object normally used to describe a consequence of sending message.

Table 12.1: Some terms definition.

- We have defined that, objects have **attributes** and by sending message, the object can do something, that is **action** or something can be done, so there may be an **event**.

- Now for program named `class.cpp`, we can say that we have a class, composed of one variable and two methods. The methods operate on the variable contained in the class when they receive messages to do so.
- Lines `11` and `12` of this program are actually the prototypes for the two methods, and are our first example of a prototype usage within a class as shown below:

```
void  set(int enter_value);
int   get_value(void);
```

- You will notice line `11` which says that the method named `set()` requires one parameter of type `int` and returns nothing, hence the return type is `void`. The method named `get_value()` however, according to line `12` has no input parameters but returns an `int` type value to the caller.

12.3.4 Sending A Message Or Function Call?

- After the definition in lines `2` through `24`, we finally come to the program where we actually use the class. In line `29` we instantiate three objects of the class `item` and name the objects `John_cat`, `Joe_cat` and `Big_cat`.
- Each object contains a single data point which we can set through the use of the method `set()` or read through the use of the method `get_value()`, but we **cannot directly** set or read the value of the data point because it is hidden within the block wall around the class as if it is in a container.
- In line `32`, we **send a message to the object** named `John_cat` instructing it to set its internal value to `10`, and even though this looks like a function call, it is properly called **sending a message to a method**. It is shown below:

```
John_cat.set(10);   //assigning data
```

- Remember that the object named `John_cat` has a method associated with it called `set()` that sets its internal value to the actual parameter included within the message.
- You will notice that the form is very much like the means of accessing the elements of a structure. You mention the **name of the object** with a **dot** connecting it to the **name of the method**.

```
Object_name.method_name()
```

e.g. `John_cat.set(10);`

- This means, perform operation `set`, with argument `10` on the instance of the object `John_cat`.
- In a similar manner, we send a message to each of the other two objects, `Joe_cat` and `Big_cat`, to set their values to those indicated.
- Lines `37` and `39` have been commented out because the operations are **illegal**. The variable named `keep_data` is private by default and therefore not available to the code outside of the object itself.
- Also, the data contained within the object named `John_cat` is not available within the methods of `Joe_cat` or `Big_cat` because they are different objects.
- The other method defined for each object is used in lines `43` through `45` to illustrate their usage. In each case, another **message is sent to each object** and the returned result is output to the standard output, screen, via the stream library `cout`
- There is another variable named `garfield` declared and used throughout this example program that illustrates, a normal variable can be intermixed with the objects and used in the normal manner.
- Compile and run this program. Try removing the comments from lines `37` and `38`, and then see what kind of error messages your compiler issues.

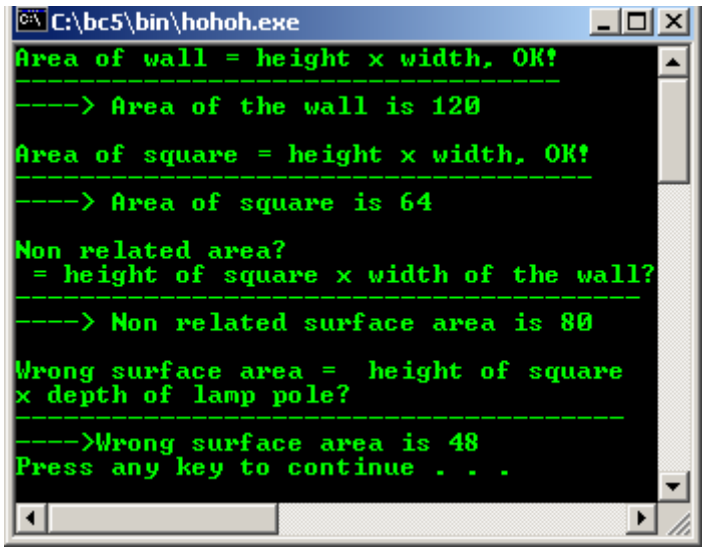
12.4 Real Object And The Problem

- Examine the program named `robject.cpp` carefully, a program with a few serious problems that will be overcome in the next program example by using the principles of encapsulation.

```
1.  //Program robject.cpp
2.  #include <iostream.h>
3.  #include <stdlib.h>
4.
```

```
5.  //--------function prototype------------
6.  int  area(int rectangle_height, int rectangle_width);
7.
8.  struct  rectangle
9.  {
10.    int  height;  //public
11.    int  width;   //public
12. };
13.
14. struct  pole
15. {
16.    int  length;   //public
17.    int  depth;    //public
18. };
19.
20. //----------rectangle area-------------
21. int surface_area(int rectangle_height, int rectangle_width)
22. {
23.    return   (rectangle_height * rectangle_width);
24. }
25.
26. //-----------main program--------------
27. void main ( )
28. {
29.     rectangle        wall,  square;
30.     pole       lamp_pole;
31.
32.    wall.height = 12;     //assigning data
33.    wall.width  = 10;
34.    square.height  =  square.width  =  8;
35.
36.    lamp_pole.length  =  50;
37.    lamp_pole.depth  =  6;
38.
39.   cout<<"Area of wall = height x width, OK!"<< "\n";
40.   cout<<"-------------------------------------"<< "\n";
41.   cout<<"----> Area of the wall is "<<surface_area(wall.height,
42.                                                    wall.width)<< "\n\n";
43.   cout<<"Area of square = height x width, OK!"<< "\n";
44.   cout<<"-------------------------------------"<< "\n";
45.   cout<<"----> Area of square is
46.                       "<<surface_area(square.height,square.width)<<"\n\n";
47.   cout<<"Non related area?"<<"\n = height of square x width of the wall?"<<
48.                                                                    "\n";
49.   cout<<"-------------------------------------"<< "\n";
50.   cout<<"----> Non related surface area is
51.                         "<<surface_area(square.height,wall.width)<<"\n\n";
52.   cout<<"Wrong surface area =  height of square"<<"\nx depth of lamp
53.                                                          pole?"<<"\n";
54.   cout<<"-------------------------------------"<< "\n";
55.   cout<<"---->Wrong surface area is
56.                       "<<surface_area(square.height,lamp_pole.depth)<<"\n";
57.
58.   system("pause");
59. }
```

59 Lines:Output:



- We have two structures declared, one being a `rectangle` and the other is a `pole`. The `depth` of the lamp pole is the depth it is buried in the ground, the overall length of the pole is therefore the sum of the `height` and `depth`.

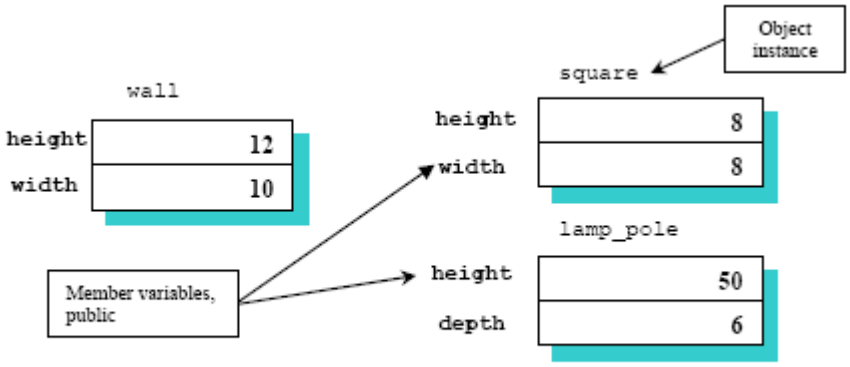


Figure 12.3: Graphically representation of the data space after execution of the program.

- Figure 12.3 try to describe the data space after the program execution. It may be a bit confused at the meaning of the result found in line `50` where we multiply the `height` of the `square` with `width` of the `wall`, because the data can be access publicly.
- Another one, although it is legal, the result has no meaning because the product of the `height` of the `square` and the `depth` of the `lamp_pole` has absolutely no meaning in any physical system we can think up in reality because they don't have relation and the result is useless.
- The **error is obvious** in a program as simple as this, but in a large program production it is very easy for such problems to be inadvertently introduced into the code by a team of programmers and the errors can be very difficult to find.
- If we have a program that defined all of the things we can do with a `square's` data and another program that defined everything we could do with `lamp_pole`'s data, and if the data could be kept **mutually exclusive**, we could prevent these silly things from happening.
- If these entities must interact, they cannot be put into separate programs, but they can be put into **separate classes** to achieve the desired goal. Compile and run the program.

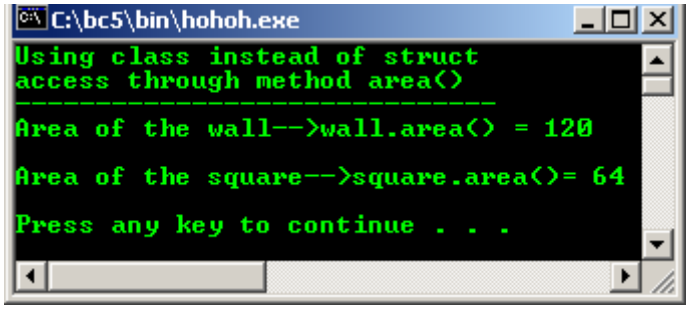
12.5 Objects Data Protection – `class` Solution

- Examine the program named `classobj.cpp` carefully, as an example of object's data protection in a very simple program.

```
1.  //Program classobj.cpp - using class instead of struct
2.  #include <iostream.h>
3.  #include <stdlib.h>
4.
```

```
5.  //---------a simple class declaration part-----------
6.  class  rectangle
7.   {
8.     //private by default, member variables
9.     int height;
10.    int width;
11.    public:
12.    //public, with two methods
13.    int area(void);
14.    void initialize(int, int);
15. };
16.
17. //-----------class implementation part--------------
18. int rectangle::area(void)
19. {
20.   return (height * width);
21. }
22.
23. void  rectangle::initialize(int initial_height, int initial_width)
24. {
25.    height = initial_height;
26.    width = initial_width;
27. }
28.
29. //normal structure - compare the usage with class
30. struct  pole
31.   {
32.      int  length;  //public
33.      int  depth;   //public
34.   };
35.
36. //-------------main program--------------------------
37. void main ( )
38. {
39.   rectangle wall, square;
40.   pole             lamp_pole;
41.
42.   //wall.height = 12;
43.   //wall.width  = 10;
44.   //square.height  =  square.width  =  8;
45.   //these 3 lines invalid now, private, access only through methods
46.
47.   wall.initialize(12,10);    //access data through method
48.   square.initialize(8,8);
49.   lamp_pole.length  =  50;  //normal struct data access
50.   lamp_pole.depth  =  6;
51.   cout<<"Using class instead of struct\n";
52.   cout<<"access through method area()\n";
53.   cout<<"-----------------------------\n";
54.   cout<<"Area of the wall-->wall.area() = "<<wall.area()<< "\n\n";
55.   cout<<"Area of the square-->square.area()= "<<square.area()<<"\n\n";
56.   //cout<<"---->Non related surface area is
57.   //               "<<surface_area(square.height,wall.width)<<"\n\n";
58.   //cout<<"---->Wrong area is
59.   //             "<surface_area(square.height,lamp_pole.depth)<<"\n";
60.   //-----illegal directly access the private data
61.
62.  system("pause");
63.  }
```

63 Lines:Output:



- In this program, the `rectangle` is changed to a `class` with the same two variables which are now **private** by default, and two methods which can **manipulate the private data**. One method is used to

initialize the values of the objects instance and the other method returns the area of the object. The two methods are defined in lines 17 through 27 in the manner describe earlier in this module.

- The pole is left as a structure to illustrate that the two can be used together and that C++ is truly an extension of ANSI-C.
- In line 39, we define two objects, once again named wall and square, but this time we cannot assign values directly to their individual components because they are private member variable of the class. The declaration is shown below:

```
rectangle  wall, square;
```

- Figure 12.4 is a graphical illustration of the two objects available for use within the calling program. Lines 42 through 44 are commented out for that reason and the messages are sent to the objects in lines 47 and 48 to tell them to initialize themselves to the values input as parameters.

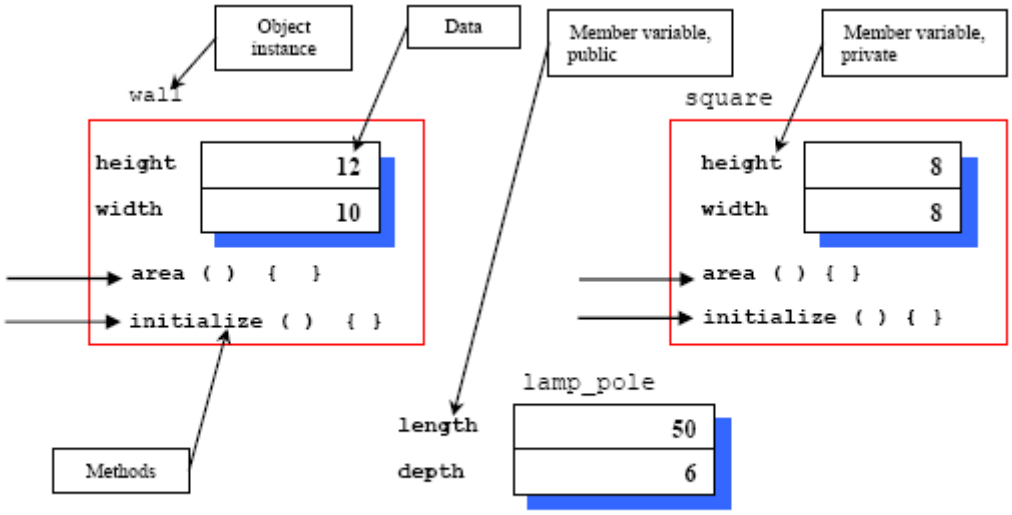


Figure 12.4: wall, square and lamp_pole objects and their data space

- The lamp_pole is initialized in the same manner as in the previous program. Using the class in this way prevents us from making the **silly calculations** we did in the last program, because we can only calculate the **area of an object** by using the **data stored within that object**.
- The compiler is now being used to prevent the erroneous calculations, so lines 56 through 59 have been commented out.
- Even though the square and the wall are both objects of class rectangle, their private data is hidden from each other such that neither can purposefully or accidentally change the other's data.
- This is the **abstract data type**, a model with a set of **private variables for data storage** and a **set of operations** that can be performed on that stored data.
- The **only** operations that can be performed on the data are those defined by the **methods**, which prevents many kinds of erroneous operations.
- Encapsulation and data hiding bind the data and methods, tightly together and limit the scope and visibility of each.
- An object is separated from the rest of the code and carefully developed in complete isolation from it. Only then, it is integrated into the rest of the code with a few of very simple **interfaces**.
- There are two aspects of this technique that really count when you develop software:

    1. First, you can get all of the data you really need through the interface.
    2. Secondly, you cannot get any protected data that you do not need.

- You are prevented from getting into the protected area and accidentally corrupting some data stored within it. You also prevented from using the wrong data because the functions available demand a serial or restricted access to the data.
- This is a very weak example because it is very easy for a knowledgeable programmer to break the encapsulation, but we will avoid this in next examples.
- You can see that object oriented programming is allowing the programmer to partition his programs into smaller portion with their own functionalities, hiding some information, accessing data in controlled manner.

- The drawback is this technique will cost you something in efficiency because every access to the elements of the object will require **time** and **inefficiency** sending messages.
- But, a program made up of **objects** that **closely match the application** in real world are much easier to understand and developed than a program that does not. In a real project however, it could be a great savings if one person developed all of the details of the `rectangle`, programmed it, and made it available to you to simplify the use.
- That is why a lot of classes have been developed, such as, Microsoft Visual C++ has MFC (Microsoft Foundation Class). We will explore this process in next examples.

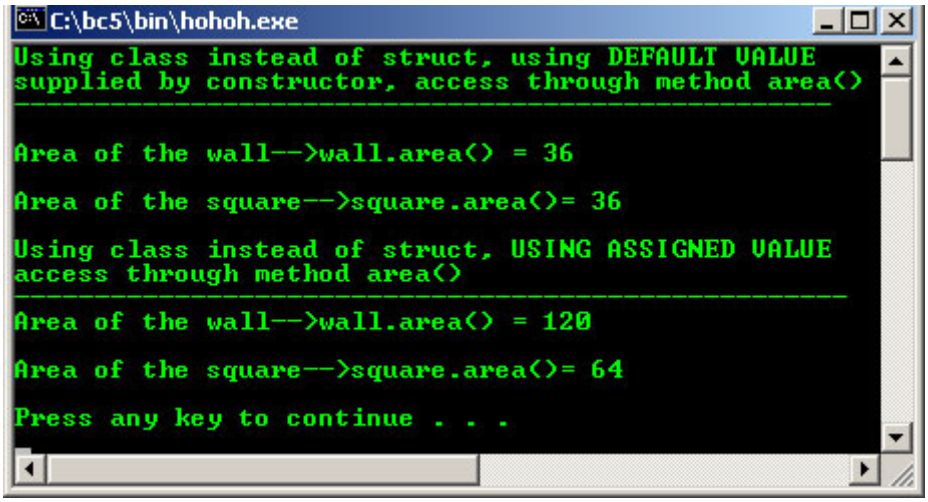
12.6 Constructors And Destructors - Initialization

- Examine program `consdest.cpp` carefully. It introduces **constructors** and **destructors** for initializing and destroying the class member variables respectively. It is very important to do the initialization mainly for pointer variables as you have learned in module 8.
- This program example is identical to the last one, except, a constructor and destructor have been added.

```
1.  //Program consdest.cpp - using class instead of struct
2.  //with constructor and destructor
3.  #include <iostream.h>
4.  #include <stdlib.h>
5.
6.  //----------a simple class declaration part-----------
7.  class  rectangle
8.  {
9.    //private by default, member variables
10.   int height;
11.   int width;
12.   //public
13.   public:
14.   rectangle(void);            //-----constructor--------
15.   int area(void);
16.   void initialize(int, int);
17.   ~rectangle(void);             //destructor
18. };
19.
20. //--------------Implementation part------------
21. rectangle::rectangle(void)
22. //constructor implementation
23. {
24.     height = 6;
25.     width = 6;
26. }
27.
28. int rectangle::area(void)
29. {
30.    return (height * width);
31. }
32.
33. void  rectangle::initialize(int initial_height, int initial_width)
34. {
35.    height = initial_height;
36.    width = initial_width;
37. }
38.
39. //----destructor implementation-----
40. rectangle::~rectangle(void)
41. {
42.    height = 0;
43.    width = 0;
44. }
45.
46. //normal structure - compare with class usage
47. struct  pole
48. {
49.    int  length;
50.    int  depth;
51. };
52.
53. //---------------main program-------------------------
54. void main ( )
55. {
56.   rectangle   wall, square;
57.   pole       lamp_pole;
58.
59.
60.  cout<<"Using class instead of struct, using DEFAULT VALUE\n";
```

```
61.  cout<<"supplied by constructor, access through method area()\n";
62.  cout<<"-------------------------------------------------\n\n";
63.  cout<<"Area of the wall-->wall.area() = "<<wall.area()<< "\n\n";
64.  cout<<"Area of the square-->square.area() = "<<square.area()<<"\n\n";
65.  //    wall.height = 12;
66.  //    wall.width  = 10;
67.  //    square.height  =  square.width  =  8;
68.  //These 3 lines, invalid now, private access only through methods
69.
70.   wall.initialize(12,10);     //override the constructor values
71.   square.initialize(8,8);
72.   lamp_pole.length  =  50;
73.   lamp_pole.depth  =  6;
74.
75.  cout<<"Using class instead of struct, USING ASSIGNED VALUE\n";
76.  cout<<"access through method area()\n";
77.  cout<<"-------------------------------------------------\n";
78.  cout<<"Area of the wall-->wall.area() = "<<wall.area()<<"\n\n";
79.  cout<<"Area of the square-->square.area()= "<<square.area()<<"\n\n";
80.  //cout<<"----> Non related surface area is
81.  //            "<<surface_area(square.height,wall.width)<<"\n\n";
82.  //cout<<"---->Wrong area is
83.  //               "<surface_area(square.height,lamp_pole.depth)<<"\n";
84.
85.  system("pause");
86. }
```

86 Lines:Output:



- The constructor always has the **same name as the class itself** and is declared in line `14`:

```
rectangle(void)   //--constructor declaration--
...
```

- The constructor is **called automatically** by the `C++` system when the object is declared and prevents the use of an **uninitialized variable**. The following is the code segment for constructor:
- When the object named `wall` is instantiated in line `56`, the constructor is called automatically by the system. The constructor sets the values of `height` and `width` each to `6` in lines `24` and `25`, in the object named `wall`.

```
rectangle::rectangle(void)
//constructor implementation
{
    height = 6;
    width = 6;
}
```

- This is printed out for reference in lines `63` and `64`.

```
cout<<"Area of the wall-->wall.area() = "<<wall.area()<< "\n\n";
cout<<"Area of the square-->square.area() = "<<square.area()<<"\n\n";
```

- Likewise, when the object `square` is defined in line `56`, the values of the `height` and the `width` of the `square` are each initialized to `6` when the constructor is called automatically.
- A constructor is defined as having the **same name as the class itself**. In this case both are named `rectangle`. The constructor **cannot have a return type** associated with it because of the definition of the `C++`. It actually has a predefined return type, a **pointer to the object itself**.
- Even though both objects are assigned values by the constructor, they are initialized in lines `70` and `71` to the new values as shown below and processing continues.

```
wall.initialize(12, 10);  //override the constructor values
square.initialize(8, 8);
```

- Since we have a constructor that does the initialization, we should probably rename the method named `initialize()` something else such as `reinitialize()`.
- The **destructor** is very similar to the constructor except that it is called automatically when each of the **objects goes out of scope**. You will recall that automatic variables have a limited lifetime because they cease to exist when the enclosing block in which they were declared is exited.
- When an object is about to be automatically de-allocated, its destructor, if one exists, is called automatically.
- A destructor is characterized as having the **same name as the class** but with a **tilde** (~) prepend to the class name. A destructor also has **no return type**.
- A destructor is **declared** in line `17`:

```
~rectangle(void);  //--destructor declaration--
      ...
```

- And **defined** in lines `40` through `44`.

```
//----destructor implementation-----
rectangle::~rectangle(void)
{
   height = 0;
   width  = 0;
}
```

- In this case, the destructor only assigns zeros to the variables prior to their being de allocated.
- The destructor is only included for illustration of how it is used. If some blocks of memory were dynamically allocated within an object, the destructor should contain code to de allocate them prior to losing the pointers to them. This would return their memory to the **free store** for later use.
- Most compilers implement the calling destructor by default.

12.7 Object Packaging

- Examine the program named `wall1.cpp` carefully. This is an example of how do not to package an object for universal use.
- This packaging is actually fine not just for a very small program, but is meant to illustrate to you how to split your program up into smaller, more manageable programs when you are developing a large program as individual or a team of programmers.
- This program is very similar to the last one, with the `pole` structure dropped and the class named `wall`. The class is declared in lines `6` through `20`:

```
//-------a simple class, declaration part--------
class wall
{
   int length;
   int width;
   //private by default
   public:
   wall(void);
   //constructor declaration
   void   set(int new_length, int new_width);
   //method
   int   get_area(void){return (length * width);}
   //method
   ~wall(void);
   //destructor declaration
};
```

- The implementation of the class is given in lines 23 through 40 as shown in the following code segment:

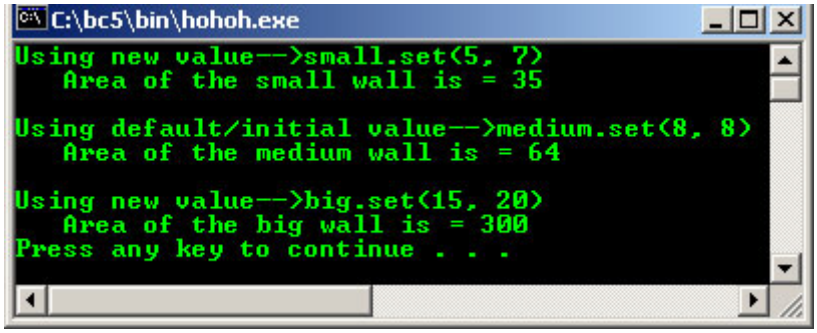
```
//----------implementation part-------------
wall::wall(void)
{  length = 8;
   width = 8;
}
//This method will set a wall size to the two inputs
//parameters by default or initial value,

void wall::set(int  new_length, int  new_width)
{
  length = new_length;
  width = new_width;
}

wall::~wall(void)
//destructor implementation
{   length = 0;
    width = 0;
}
```

- Study this program, compile and run.

```
1.  //Program wall1.cpp
2.  #include  <iostream.h>
3.  #include  <stdlib.h>
4.
5.  //-------a simple class, declaration part--------
6.  class  wall
7.  {
8.     int  length;
9.     int  width;
10.   //private by default
11.    public:
12.    wall(void);
13.   //constructor declaration
14.    void   set(int new_length, int new_width);
15.   //method
16.    int   get_area(void){return (length * width);}
17.   //destructor method
18.    ~wall(void);
19.    //destructor declaration
20. };
21.
22. //----------implementation part-------------
23. wall::wall(void)
24. {  length = 8;
25.    width = 8;
26. }
27. //This method will set a wall size to the two input
28. //parameters by default or initial value,
29.
30. void  wall::set(int  new_length, int  new_width)
31. {
32.   length = new_length;
33.   width = new_width;
34. }
35.
36. wall::~wall(void)
37. //destructor implementation
38. {   length = 0;
39.    width = 0;
40. }
41.
42. //----------------main program---------------------
43. void  main()
44. {
45.   wall   small, medium, big;
46.   //three objects instantiated of type class wall
47.
48.   small.set(5, 7);
49.   //new length and width for small wall
50.   big.set(15, 20);
51.   //new length and width for big wall
52.   //the medium wall uses the default
53.   //values supplied by constructor (8,8)
```

```
54.
55.   cout<<"Using new value-->small.set(5, 7)\n";
56.   cout<<"   Area of the small wall is = "<<small.get_area()<<"\n\n";
57.   cout<<"Using default/initial value-->medium.set(8, 8)\n";
58.   cout<<"   Area of the medium wall is = "<<medium.get_area()<<"\n\n";
59.   cout<<"Using new value-->big.set(15, 20)\n";
60.   cout<<"   Area of the big wall is = "<<big.get_area()<<"\n";
61.
62.   system("pause");
63. }
```

63 Lines:Output:



12.8 Inline Implementation

- The method in line `16` as shown below contains **the implementation for the method as a part of the declaration** because it is very simple function.

```
int get_area(void){return (length * width);}
```

- When the **implementation is included in the declaration**, it will be assembled **inline** whenever this function is called leading to much faster code execution.
- This is because there is no function call overhead when making a call to the method. In some cases this will lead to code that is both smaller and faster.
- Inline code implementation in `C++` accomplishes the same efficiency that the **macro** accomplishes in `C`, and it is better.

12.9 The Class Header File – User Defined HeaderFfile

- Examine the `wall.h` program carefully. You will see that it is only the **class definition**. No details are given of how the various methods are implemented except of course for the inline method named `get_area()`.
- This gives the **complete definition** of how to use the class with no **implementation** details. You will notice that it contains lines `6` through `20` of the previous program `wall1.cpp`.
- This is called **header file** and cannot be compiled or run. Create this file, under your `main()` program folder. You have learnt this step regarding the user defined function in module 4.

```
1.  //Program wall.h, the header file
2.
3.  //--------class declaration part---------------
4.  class wall
5.  {
6.    int   length;
7.    int   width;
8.     public:
9.    wall(void);
10.   //constructor method
11.   void set(int new_length, int new_width);
12.   //method
13.   int  get_area(void) {return (length * width);}
14.   //destructor method
15.   ~wall(void);
16.   //destructor
17. };
18.
19. //this header file cannot be compiled or run
```

19 Lines

12.9.1 The Class Implementation Program

- Examine the program `wall.cpp` carefully. This is the **implementation of the methods** declared in the class header file (`wall.h`). Notice that the class header file is included into this file in line `3` which contains the prototype for the methods and the definitions of the variables to be manipulated.

```
1.  //Program wall.cpp, this is implementation file
2.
3.  #include  "wall.h"
4.
5.  wall::wall(void)
6.  //constructor implementation
7.  {
8.     length = 8;
9.     width = 8;
10. }
11.
12. //This method implementation will set a
13. //wall size to the two input parameters
14.
15. void wall::set(int new_length, int new_width)
16. {
17.    length = new_length;
18.    width = new_width;
19. }
20.
21. wall::~wall(void)
22. //destructor implementation
23. {
24.    length = 0;
25.    width = 0;
26. }
27.
28. //This implementation program should be compiled
29. //without error, generating object file but can't be
30. //run because there is no main() entry point.
```

30 Lines

- The code from lines `23` through `41` of `wall1.cpp` is contained in this program which is the **implementation of the methods** declared in the class named `wall`.
- This file should be compiled but it cannot be run because there is **no main entry point** which is required for all `ANSI-C` or `C++` programs. Make sure there is no error and the `wall.h` header file also must be included in your project.
- When it is compiled, the **object code** will be stored in the current directory and available for use by other programs. It should be noted here that the **result of the compilation** is usually referred to as **an object** as used in object oriented programming.
- The separation of the **definition** and the **implementation** is a major step forward in software engineering. The definition file or **interface** is all the user needs in order to use this class effectively in a program.
- User needs no knowledge of the **actual implementation** of the methods. If he had the implementation available, he may study the code and modify it to make the overall program slightly more efficient, but this would lead to **non-portable** software and possible bugs' later if the implementer changed the implementation without changing the **interface**.
- The **purpose** of object oriented programming is to hide the implementation in such a way that the implementation can not affect anything outside of its own small and well defined boundary or interface. You should compile this implementation program without error and we will use the result with the next program example. The compilation generate object file. No need to run.

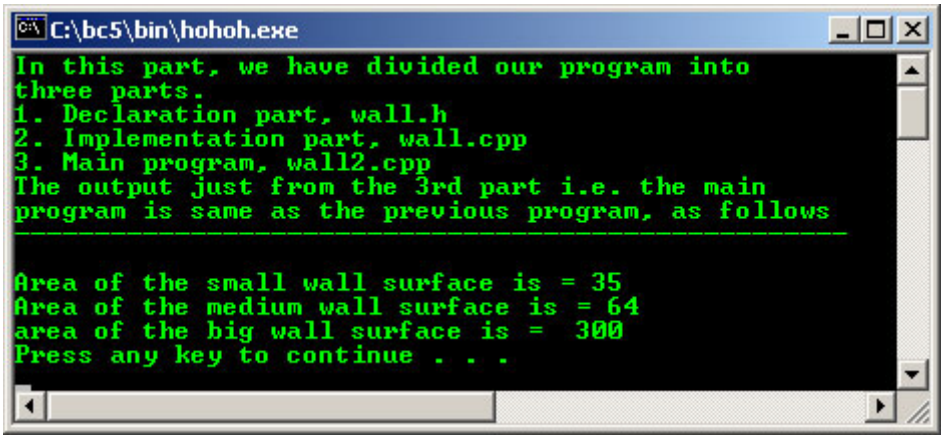
12.9.2 Using The `wall` Object, `main()` Program

- Examine the `wall2.cpp` program and you will find that the class we defined previously is used within this file. In fact, these last three programs (and three files!) taken together are identical to the program named `wall1.cpp` studied earlier.

```
1.  //Program wall2.cpp here are the main program,
2.  //the actual program that programmer create
3.
```

```
4.  #include   <iostream.h>
5.  #include   <stdlib.h>
6.  #include   "wall.h"
7.  //user defined header file containing
8.  //class declaration
9.
10. main()
11. {
12.    wall   small, medium, large;
13.   //three objects instantiated of class wall
14.
15.    small.set(5, 7);
16.    large.set(15, 20);
17.   //the medium wall uses the values
18.   //supplied by the constructor
19.
20.    cout<<"In this part, we have divided our program into\n";
21.    cout<<"three parts.\n"<<"1. Declaration part, wall.h\n";
22.    cout<<"2. Implementation part, wall.cpp\n"<<"3. Main program, wall2.cpp\n";
23.    cout<<"The output just from the 3rd part i.e. the main \n";
24.    cout<<"program is same as the previous program, as follows\n";
25.    cout<<"-------------------------------------------------\n\n";
26.    cout<<"Area of the small wall surface is = "<<small.get_area()<<"\n";
27.    cout<<"Area of the medium wall surface is = "<<medium.get_area()<<"\n";
28.    cout<<"area of the big wall surface is =  "<<large.get_area()<<"\n";
29.    system("pause");
30. }
```

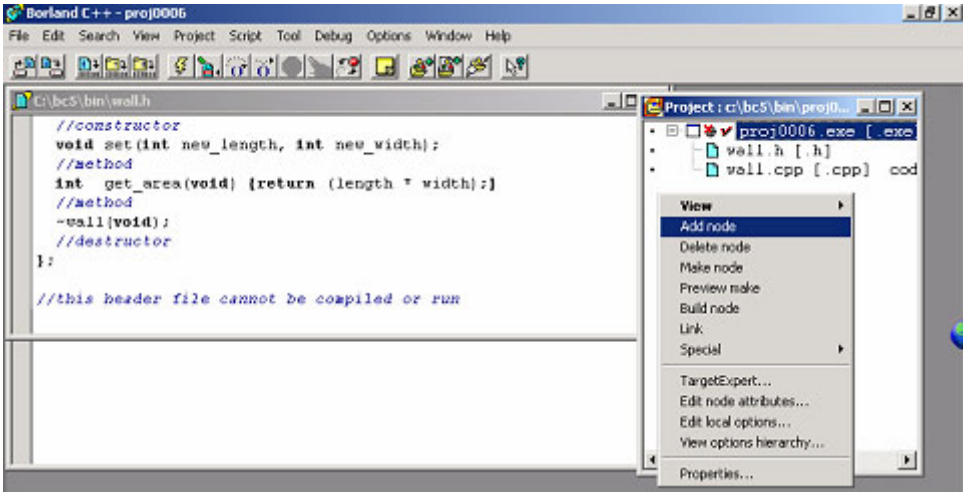
30 Lines:Output:

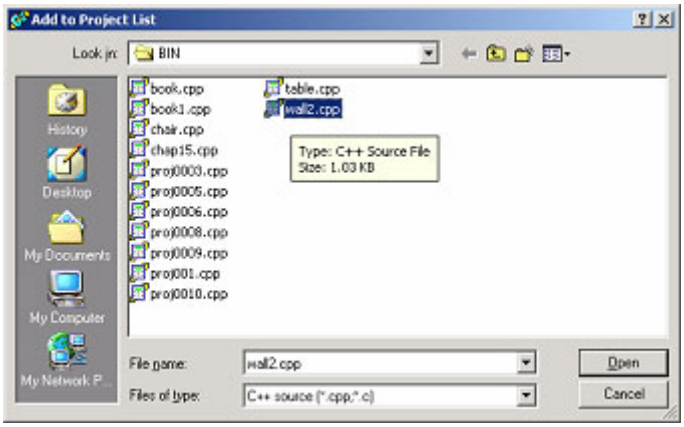


- The `wall.h` file is included here, in line 6, since the definition of the `wall` class is needed to declare three objects and use their methods. There is big difference in `wall1.cpp` and `wall2.cpp` as we will see shortly.
- We are not merely **calling functions** and changing the terminology a little to say we are **sending messages**. There is an inherent difference in the two operations.
- Since the data for each object is tightly bound up within the object, there is no way to get to the data except through the methods and we send a message to the object telling it to perform some operation based on its internally stored data.
- However, whenever we call a function, we **take along the data** for it to work with, as parameters since it doesn't contain its own data.
- Compile and run this program, but when you come to the **link** step, you will be required to link this program along with the result of the compilation when you compiled the class named `wall` (`wall.cpp`). The object file (compiled form) and the header file must be linked together.
- To make sure there is no error, follow these steps (Borland C++, Microsoft Visual C++ or other visual IDE):

    1. Create a project.
    2. Create header file `wall.h` and save it.
    3. Create `wall.cpp` file, compile it without error, generating object file.
    4. Create the main program `wall2.cpp`, compile and run this file.
    5. All three files must be in the same project.
    6. Regarding the use of the `"box.h"` or <`box.h`>, please refer to module 4.

- For Borland C++, the details steps are: First, select the project folder, right click and select `Add node`.
- When the **`Add to the Project List`** Dialog box appears, select the desired file(s) and click **`Open`** button. Finally compile the `main()` program.





12.10 Data Hiding

- The three programs examples we have just studied illustrate a concept of **data hiding**.
- Since the only information the user of the class really need is the **class header**, the details of the **implementation** can be kept hidden from him.
- Since he doesn't know exactly what the implementer did, he must follow only the **definition** given in the header file. As mentioned earlier, accidental corruption of data is also prevented.
- Another reason for hiding the implementation is economic. The company that supplied you with your `C++` compiler gave you many library functions but did not supply the source code of the library functions, only the **interface** to each function.
- You know how to use the functions but you do not have the details of implementation, nor do you need them. Likewise, a class library vendors can develop, which supplies users with libraries of high quality, completely developed, and tested **classes**, for a licensing fee.
- Since the user only needs the **interface** defined, he can be supplied with the interface and the object (compiled) code for the class and can use it in any way he desired. The vendors' source code is protected from accidental or intentional compromise and he can maintain complete control over it.
- But keep in mind that may be this is not the case for the open source community :o).

12.11 Abstract Data Type

- An abstract data type is a group of data, each of which can store **a range of data values** and **a set of methods** or **functions** that operate on that data. Since the data are protected from any outside influence, it is protected and said to be encapsulated.
- Also, since the data is somehow related, it is a very coherent group of data that may be highly interactive with each other, but with little interaction outside the scope of its class.
- The methods, on the other hand, are coupled to the outside world through the **interface**, but there are a limited number of contacts with or a weak coupling with the outside.

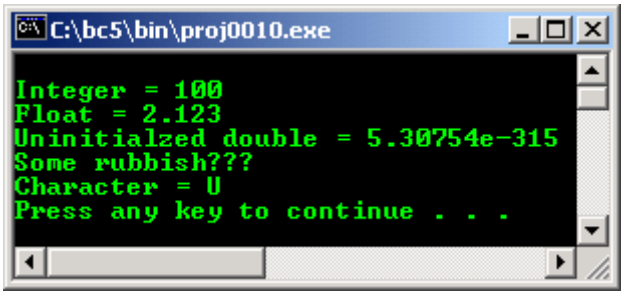
- Because of the tight coherency and the loose coupling, **ease of maintenance** of the software should be greatly enhanced.
- The variables could have been hidden completely out of sight in another file, but because the designers of C++ wished to make the execution of the completed application as efficient as possible, the variables were left in the **class definition** where they can be seen but not used.

12.12 The union in C++

- In C++ `union` can contain access specifiers (`public`, `protected`, `private`), member data, and member functions, including constructors and destructors.
- It cannot contain virtual functions or static data members. It cannot be used as a base class, nor can it have base classes. Default access of members in a union is public. In C++, the `union` keyword is unnecessary.
- Same as in C, a union type variable can hold one value of any type declared in the union by using the member-selection operator (**.**) to access a member of a union.
- You can declare and initialize a union in the same statement by using enclosed curly braces. The expression is evaluated and assigned to the **first field** of the union.
- Program example:

```
#include <iostream.h>
#include <stdlib.h>

union Num
{
   int      ValueI;
   float    ValueF;
   double   ValueD;
   char    ValueC;
};

void main()
{
    //Optional union keyword
    //ValueI = 100
    Num TestVal = {100};

    cout<<"\nInteger = "<<TestVal.ValueI<<endl;
    TestVal.ValueF = 2.123L;
    cout<<"Float = "<<TestVal.ValueF<<endl;
    cout<<"Uninitialzed double = "<<TestVal.ValueD<<endl;
    cout<<"Some rubbish???"<<endl;
    TestVal.ValueC = 'U';
    cout<<"Character = "<<TestVal.ValueC<<endl;
    system("pause");
}
```

Output:

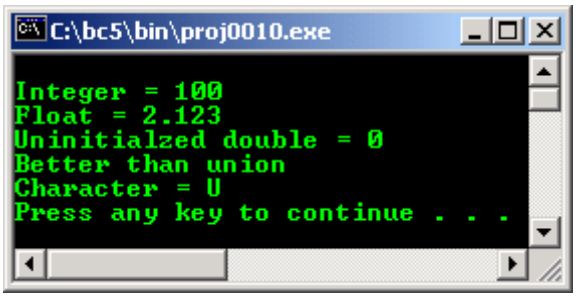


12.13 The struct in C++

- The `struct` is still useable in C++ and operates just like it does in ANSI-C with one addition. You can include methods in a structure that operate on data in the same manner as in a class, but methods and data are **automatically defaulted to be public** at the beginning of a structure.
- Of course you can make any of the data or methods private section within the structure. The structure should be used **only** for constructs that are truly structures. If you are building even the simplest objects, you are advised to use classes to define them.
- Program example as used in previous `union` example.

```
#include <iostream.h>
#include <stdlib.h>

struct Num
{
   int      ValueI;
   float    ValueF;
   double   ValueD;
   char    ValueC;
};

void main()
{
    struct Num TestVal = {100};

    cout<<"\nInteger = "<<TestVal.ValueI<<endl;
    TestVal.ValueF = 2.123L;
    cout<<"Float = "<<TestVal.ValueF<<endl;
    cout<<"Uninitialzed double = "<<TestVal.ValueD<<endl;
    cout<<"Better than union"<<endl;
    TestVal.ValueC = 'U';
    cout<<"Character = "<<TestVal.ValueC<<endl;
    system("pause");
}
```

Output:



Program Examples and Experiments

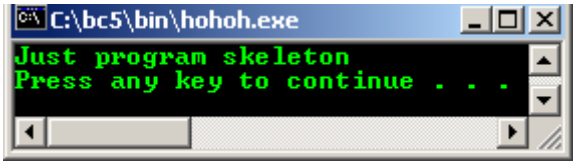
Example#1

- The following example just to show you the simple program development process using class. We are very familiar with line as a basic object used in drawing. So we would like to create line object.
- Think simple; first of all just create the program skeleton as shown below. Compile and run, make sure there is no error.

```
//Creating simple class, STEP #1
#include  <iostream.h>
#include  <stdlib.h>

//--------class, declaration part------
class line
{

   public:
   line(void);
   ~line(void);
};

//------class implementation part-------
line::line(void)
{

}

line::~line(void)
{

}

//---------main program-------------
int main()
{
  line   LineOne;
```

```
    cout<<"Just program skeleton\n";
    system("pause");
    return 0;
}
```

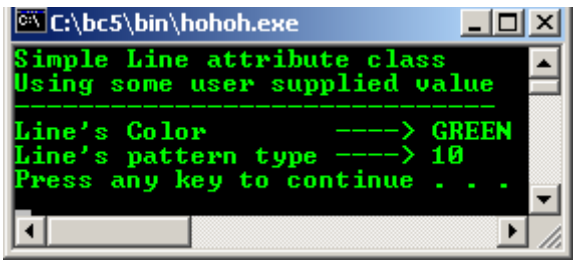
Output:



- Next step is to add simple functionalities. We add one line attribute, line's color and line pattern type, with simple implementation returning the color value and user selected pattern type respectively.
- Compile and run this program without error.

```
//Creating simple class, STEP #2
#include  <iostream.h>
#include  <stdlib.h>

//--------class, declaration part------
class line
{
   char*  color;
   int  pattern;
   public:
   line(void);
   char*   LineColor(char*){ return  color = "GREEN";};
   int    LinePattern(int pattern){return pattern;};
   ~line(void);
};

//----class implementation part-----
line::line(void)
{
   //constructor's value…
   pattern = 12;
}

line::~line(void)
{
   color = NULL;
   pattern = 0;
}

//---------------main program----------------------
void main()
{
  line   LineOne;
  int x = 10;

  cout<<"Simple Line attribute class\n";
  cout<<"Using some user supplied value\n";
  cout<<"------------------------------"<<"\n";
  cout<<"Line's Color        ----> "<<LineOne.LineColor("")<<"\n";
  cout<<"Line's pattern type ----> "<<LineOne.LinePattern(x)<<"\n";
  system("pause");
}
```

Output:



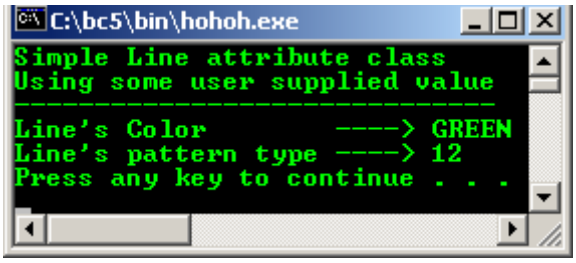
- Let do some tweaking to this very simple class, notice the difference output and the program execution flow. Firstly, Change the following red line of code,

```
int LinePattern(int pattern){return pattern;};
```

- To the following code, recompile and rerun the program, with no argument supplied, constructor value (12) will be used.

```
int LinePattern(int){return pattern;};
```

- The output:



- Next, change again the same line of code:

```
int LinePattern(int pattern){return pattern;};
```

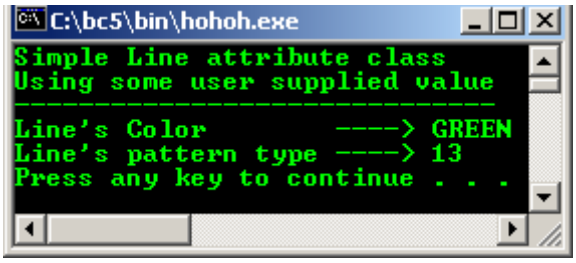
- To the following code:

```
int LinePattern(int){return pattern = 13;};
```

- And delete or comment out the following line of code, recompile and rerun the program, so there is no default constructor value.

```
pattern = 12;
```

- The output is:

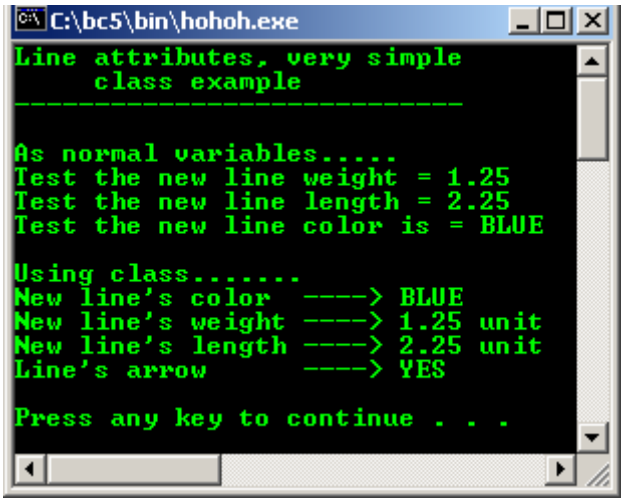


- Finally we add other line attributes to complete our line object using class.

```
//Creating simple class, STEP #3, complete
#include  <iostream.h>
#include  <stdlib.h>

//--------class, declaration part------
class line
{
       char*  color;
       float  weight;
       float  length;
       char * arrow;

       public:
       line(void);
       char* LineColor(char* color){return color;};
       float LineWeight(float weight){return weight;};
       float LineLength(float length){return length;};
       char *LineArrow(char* arrow){return arrow = "YES";};
       ~line(void);
};
```

```
//------implementation part-------
line::line(void)
{
	//constructors or initial values…
	weight = 0.25;
	length = 10;
}

line::~line(void)
{
	color = NULL;
	weight = 0;
	length = 0;
	arrow = NULL;
}

//---------------main program----------------
void main()
{
	line   LineOne;

	float x = 1.25, y = 2.25;
	char newcolor[10] = "BLUE", *colorptr;

	cout<<"Line attributes, very simple\n";
	cout<<"     class example\n";
	cout<<"----------------------------"<<"\n";

	colorptr = newcolor;
	//just for testing the new attribute values...
	cout<<"\nAs normal variables....."<<endl;
	cout<<"Test the new line weight = "<<x<<endl;
	cout<<"Test the new line length = "<<y<<endl;
	cout<<"Test the new line color is = "<<colorptr<<endl;

	cout<<"\nUsing class......."<<endl;
	cout<<"New line's color  ----> "<<LineOne.LineColor(colorptr)<<"\n";
	cout<<"New line's weight ----> "<<LineOne.LineWeight(x)<<" unit"<<"\n";
	cout<<"New line's length ----> "<<LineOne.LineLength(y)<<" unit""\n";

	cout<<"Line's arrow      ----> "<<LineOne.LineArrow(" ")<<"\n\n";
	system("pause");
}
```

Output:

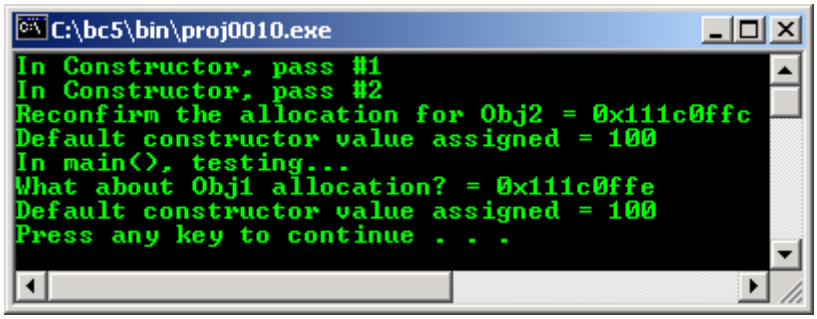


- Let do some experiment against the constructor and destructor. In this program example we declare a member variable `TestVar` and a member function `DisplayValue()` as a tester.
- Compile and execute this program.

```
//testing the constructor and destructor
#include <iostream.h>
#include <stdlib.h>

//----class declaration part--------
class TestConsDest
```

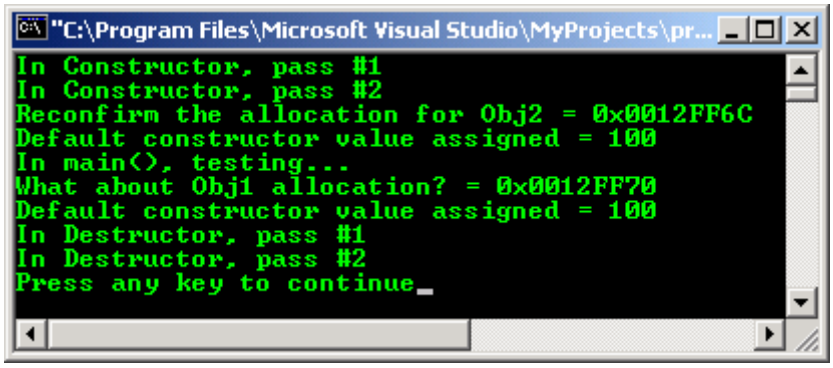
```
{
	//member variable...
	public:
	int TestVar;

	//member functions, constructor and destructor...
	public:
	TestConsDest();
	int DisplayValue();
	~TestConsDest();
};

//----------class implementation part------------
//constructor...
TestConsDest::TestConsDest()
{
	//test how the constructor was invoked...
	//static-retain the previous value…
	static int x=1;
	cout<<"In Constructor, pass #"<<x<<endl;
	x++;
}

//simple function returning a value...
int TestConsDest::DisplayValue()
{
	return TestVar = 100;
}

//destructor...
TestConsDest::~TestConsDest()
{
	//test how destructor was invoked...
	static int y=1;
	cout<<"In Destructor, pass #"<<y<<endl;
	y++;
	//explicitly...
	TestVar = 0;
}

//----------------main program-----------------------
int main()
{
	//instantiate two objects...
	//constructor should be invoked...
	//with proper memory allocation...
	TestConsDest Obj1, Obj2;

	cout<<"Reconfirm the allocation for Obj2 = "<<&Obj2<<endl;
	cout<<"Default constructor value assigned = "<<Obj1.DisplayValue()<<endl;
	cout<<"In main(), testing..."<<endl;
	cout<<"What about Obj1 allocation? = "<<&Obj1<<endl;
	cout<<"Default constructor value assigned = "<<Obj1.DisplayValue()<<endl;
	system("pause");
	return 0;
}
```

Output:



- Notice that the constructor has been invoked properly for the objects `Obj1` and `Obj2` memory allocation. It has been reconfirmed by checking the memory address of the objects.

- Unfortunately from the output, there is no cleanup work done. If the destructor called automatically when the program execution exit the closing brace of the `main()` program, it is OK. If it is not, then, there should be **memory leak** somewhere :o).
- Fortunately, when this program example compiled and run using Microsoft Visual C++ 6.0®, the destructor properly called for cleanup. The output is shown below:



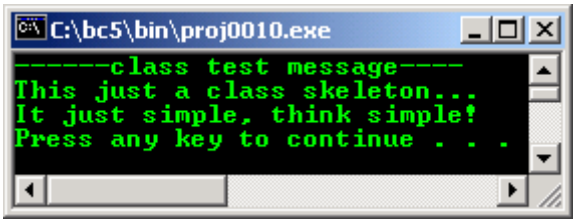
- It seems that the destructor called when the execution exit the closing brace because the second time sending message to the `DisplayValue()`, the value of `TestVar` = `100` still been displayed although we have set the value to 0 in the destructor.

Final remarks think simple!!!

```
//class skeleton program
//as summary...
#include <iostream.h>
#include <stdlib.h>

//----The base class declaration---
class BaseClass
{
//---member variables declaration---
//declare all your variables here
//with optional access restrictions
public:
	int p;

//this keyword will be explained later...
protected:
	float q;

private:
	char r;

//---member functions or methods---
//define your functions here
//also with optional access restrictions
public:
	//constructor
	BaseClass();
	//destructor
	~BaseClass();

private:
	int Funct1();

protected:
	void Funct2();
};

//---class implementation part------
//define your functions or method here
BaseClass::BaseClass()
{ }

int BaseClass::Funct1()
{return 0;}

//constructor implementation
void BaseClass::Funct2()
{}
```

```
//destructor implementation
BaseClass::~BaseClass()
{}

//the main program
//start instantiate objects here...
int main(void)
{
       cout<<"------class test message----"<<endl;
       cout<<"This just a class skeleton..."<<endl;
       cout<<"It just simple, think simple!"<<endl;

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

Output:

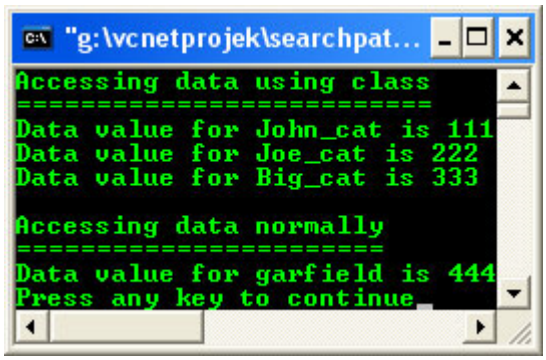


- Program example compiled using VC++/VC++ .Net.

```
//Program class.cpp using class instead of struct
#include <iostream>
using namespace std;

//--------Class declaration part------------
class item
{
    //private by default, it is public in struct
     int  keep_data;
    //public part
     public:
    void  set(int enter_value);
    int  get_value(void);
};

//--------Class implementation part---------
void item::set(int enter_value)
{
  keep_data = enter_value;
}

int item::get_value(void)
{
  return  keep_data;
}

//-------main program------------
void main()
{
    //three objects instantiated
     item    John_cat, Joe_cat, Big_cat;
    //normal variable
     int   garfield;

     //assigning data
     John_cat.set(111);
     Joe_cat.set(222);
    Big_cat.set(333);
    garfield = 444;
  //John_cat.keep_data = 100;
  //Joe_cat.keep_data = 110;
  //These are illegal because keep_data now, is private by default

cout<<"Accessing data using class\n";
cout<<"=========================\n";
cout<<"Data value for John_cat is "<<John_cat.get_value()<<"\n";
cout<<"Data value for Joe_cat is "<<Joe_cat.get_value()<<"\n";
cout<<"Data value for Big_cat is "<<Big_cat.get_value()<<"\n";
```

```
cout<<"\nAccessing data normally\n";
cout<<"=======================\n";
cout<<"Data value for garfield is "<<garfield<<"\n";
}
```

Output:



- A program example compiled using **g++**.

```
////////-simpleclass.cpp-////////
///////FEDORA 3, g++ x.x.x////////
/////Creating a simple class/////
#include <iostream>
using namespace std;

//--------class, declaration part------
class line
{
        char*  color;
        float  weight;
        float  length;
        char * arrow;

        public:
        line(void);
        char* LineColor(char* color){return color;};
        float LineWeight(float weight){return weight;};
        float LineLength(float length){return length;};
        char *LineArrow(char* arrow){return arrow = "YES";};
        ~line(void);
};

//------implementation part-------
line::line(void)
{
        //constructors or initial values.
        weight = 0.25;
        length = 10;
}

line::~line(void)
{
        color = NULL;
        weight = 0;
        length = 0;
        arrow = NULL;
}

//---------------main program-----------------
int main()
{
        line   LineOne;

        float x = 1.25, y = 2.25;
        char newcolor[10] = "BLUE", *colorptr;

        cout<<"Line attributes, very simple\n";
        cout<<"     class example\n";
        cout<<"----------------------------"<<"\n";

        colorptr = newcolor;
        //just for testing the new attribute values...
        cout<<"\nAs normal variables....."<<endl;
```

```
        cout<<"Test the new line weight = "<<x<<endl;
        cout<<"Test the new line length = "<<y<<endl;
        cout<<"Test the new line color is = "<<colorptr<<endl;

        cout<<"\nUsing class......."<<endl;
        cout<<"New line's color  ----> "<<LineOne.LineColor(colorptr)<<"\n";
        cout<<"New line's weight ----> "<<LineOne.LineWeight(x)<<" unit"<<"\n";
        cout<<"New line's length ----> "<<LineOne.LineLength(y)<<" unit""\n";
        cout<<"Line's arrow      ----> "<<LineOne.LineArrow(" ")<<"\n\n";
        return 0;
}
```

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

```
Line attributes, very simple
     class example
----------------------------

As normal variables.....
Test the new line weight = 1.25
Test the new line length = 2.25
Test the new line color is = BLUE

Using class.......
New line's color  ----> BLUE
New line's weight ----> 1.25 unit
New line's length ----> 2.25 unit
Line's arrow      ----> YES
```

----------o0o----------

---www.tenouk.com---

Further reading and digging:

1. Check the best selling C/C++ books at Amazon.com.
2. C++ standards reference: ISO/IEC 14882:1998 on the programming language C++.
3. What C++ has done to this world? C++ Applications - Bjarne Stroustrup site of C++. But, read HERE for some interesting Bjarne's interview :o).