

## MODULE 13

### CLASS - ENCAPSULATION II

My Training Period:      hours

#### Abilities

Able to understand and use:

- Class and arrays.
- Pointer within class.
- Pointer of the objects.
- `static` member variable.
- Pointer of object to another object: list and linked list examples.
- Class and strings.
- Nesting the classes.
- `new` and `delete` operators.
- `this` pointer.
- Operators overloading.
- Functions overloading.
- Default methods.

#### 13.1 Introduction

- This Module will illustrate how to use some of the C/C++ features with classes and objects. **Pointers** to an object as well as pointers within an object also will be illustrated.
- **Arrays** embedded within an object, and an array of objects will also be experimented.
- Since objects are simply another C++ data construct, all of these things are possible. Make sure you have pre knowledge of the [array](#) and [pointer](#) to continue on this Module.
- Here, method and function terms may be used interchangeably, as simplicity they provide the same meaning.

#### 13.2 Object Array

- Examine the program named `obarray.cpp` carefully. This program is nearly identical to the program named `wall1.cpp` in the previous Module, until we come to line 47 where an array of 4 wall objects named `group` are defined (together with 3 normal variable) as shown below:

```
wall small, medium, large, group[4];
```

- Be reminded that the proper way to use these constructs is to separate them into three programs: `wall.h`, `wall.cpp` (compiled form) and `wall2.cpp` as in the previous Module.
- We do not follow the rule here just to simplify our learning.

```
1.  //program obarray.cpp
2.  //object and an array
3.  #include <iostream.h>
4.  #include <stdlib.h>
5.
6.  //-----class declaration-----
7.  class wall
8.  {
9.  int length;
10. int width;
11. static int extra_data;
12. //declaration of the extra_data static type
13. public:
14. wall(void);
15. void set(int new_length, int new_width);
16. int get_area(void);
17. int get_extra(void) { return extra_data++; } //inline function
18. };
19.
20. //-----class implementation-----
21. int wall::extra_data; //Definition of extra_data
22.
23. //constructor, assigning initial values
24. wall::wall(void)
```

```

25.     {
26.         length = 8;
27.         width = 8;
28.         extra_data = 1;
29.     }
30.
31.     //This method will set a wall size to the two input parameters
32.     void wall::set(int new_length, int new_width)
33.     {
34.         length = new_length;
35.         width = new_width;
36.     }
37.
38.     //This method will calculate and return the area of a wall instance
39.     int wall::get_area(void)
40.     {
41.         return (length * width);
42.     }
43.
44.     //-----main program-----
45.     void main()
46.     {
47.         wall    small, medium, large, group[4];
48.         //7 objects are instantiated, including an array
49.
50.         small.set(5, 7);           //assigning values
51.         large.set(15, 20);
52.
53.         for(int index=1; index<4; index++)           //group[0] uses default
54.             group[index].set(index + 10, 10);
55.
56.         cout<<"Sending message-->small.get_area()\n";
57.         cout<<"Area of the small wall is "<<small.get_area()<<"\n\n";
58.         cout<<"Sending message-->medium.get_area()\n";
59.         cout<<"Area of the medium wall is "<<medium.get_area()<<"\n\n";
60.         cout<<"Sending message-->large.get_area()\n";
61.         cout<<"Area of the large wall is "<<large.get_area()<<"\n\n";
62.
63.         cout<<"New length/width group[index].set(index + 10, 10)\n";
64.         for(int index=0; index<4; index++)
65.         {
66.             cout<<"Sending message using an array
67.             -->group"<<"["<<index<<"].get_area()\n";
68.             cout<<"An array of wall area "<<index<<" is
69.             "<<group[index].get_area()<<"\n\n";
70.         }
71.
72.         cout<<"extra_data = 1, extra_data++\n";
73.         cout<<"Sending message using-->small.get_extra() or \n";
74.         cout<<"array, group[0].get_extra()\n";
75.
76.         cout<<"Extra data value is "<<small.get_extra()<<"\n";
77.         cout<<"New Extra data value is "<<medium.get_extra()<<"\n";
78.         cout<<"New Extra data value is "<<large.get_extra()<<"\n";
79.         cout<<"New Extra data value is "<<group[0].get_extra()<<"\n";
80.         cout<<"New Extra data value is "<<group[3].get_extra()<<"\n";
81.
82.         system("pause");
83.     }

```

**83 lines: Output:**

```

C:\bc5\bin\hohoh.exe
Sending message-->small.get_area()
Area of the small wall is 35

Sending message-->medium.get_area()
Area of the medium wall is 64

Sending message-->large.get_area()
Area of the large wall is 300

New length/width group[index].set(index + 10, 10)
Sending message using an array-->group[0].get_area()
An array of wall area 0 is64

Sending message using an array-->group[1].get_area()
An array of wall area 1 is110

Sending message using an array-->group[2].get_area()
An array of wall area 2 is120

Sending message using an array-->group[3].get_area()
An array of wall area 3 is130

extra_data = 1, extra_data++
Sending message using-->small.get_extra() or
array, group[0].get_extra()
Extra data value is 1
New Extra data value is 2
New Extra data value is 3
New Extra data value is 4
New Extra data value is 5
Press any key to continue . . .

```

- Each four wall objects will be initialized to the values defined within the constructor since the constructor will be executed for each wall as they are defined.
- In order to define an array of objects, a **constructor** for that object with **no parameters** must be available.
- Line 53 defines a for loop that begins with 1 instead of the normal starting index 0 for an array leaving the first object, named group[0], to use the default values stored when the constructor was called (length = 8, width = 8 and extra\_data = 1).

```

for(int index=1; index<4; index++)    //group[0] uses default value
    group[index].set(index + 10, 10);

```

- Notice that sending a message to one of the array objects, uses the same construct as used for any object. The name of the array followed by its index in square brackets is used to send a message to one of the objects in the array as,

```

group[index].set(index + 10, 10)

```

- The general form:

```

The_object.the_method

```

- This is illustrated again in line 68-69 as shown below.

```

cout<<"An array of wall area "<<index<<" is "<<group[index].get_area()<<"\n\n";

```

- The other method get\_extra() is called in the cout statement in lines 76 to 80 where the area of the four walls in the group array are displayed. The following is the code segment:

```

cout<<"Extra data value is "<<small.get_extra()<<"\n";
cout<<"New Extra data value is "<<medium.get_extra()<<"\n";
cout<<"New Extra data value is "<<large.get_extra()<<"\n";
cout<<"New Extra data value is "<<group[0].get_extra()<<"\n";
cout<<"New Extra data value is "<<group[3].get_extra()<<"\n";

```

### 13.3 static Variable

- An extra variable as shown below was included for illustration, named `extra_data` in line 11. Since the keyword **static** is used as variable modifier, it is an **external variable** and means **only one copy** of this variable will ever exist.

```
static int extra_data;
```

- All seven objects defined in line 47 of this class, **share** a single copy of this variable which is global to the objects. The variable is actually only declared here which says it will exist somewhere, but it is not yet defined.
- A **declaration** says the variable will exist and gives it a name, but the **definition** actually defines a place to store it somewhere in the computers memory.
- By definition, a `static` variable can be declared in a class header but it cannot be defined there, so it is usually defined in the implementation program. In this case it is defined in line 21 as shown below and can then be used throughout the class.

```
int wall::extra_data; //Definition of extra_data
```

- Figure 13.1 is a graphical representation of some of the variables. Note that the objects named `large`, `group[0]`, `group[1]`, and `group[2]` are not shown but they also share the variable named `extra_data`.
- Each object **has its own personal** length and width because they are not declared static.

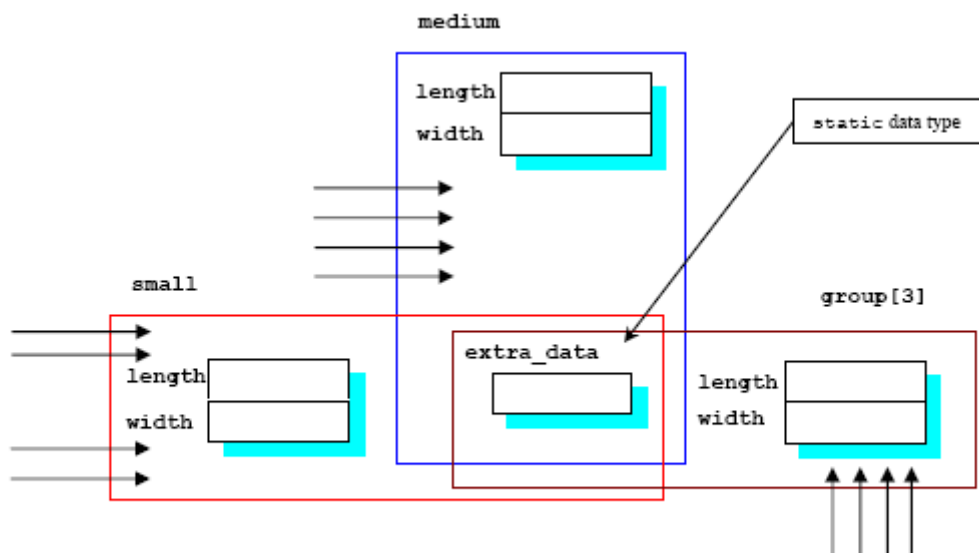


Figure 13.1: Graphical representation some of the variables.

- Line 28 of the constructor sets this single global variable to 1 each time an object is declared. Only one assignment is necessary as follows.

```
extra_data = 1;
```

- To illustrate that there is only one variable shared by all objects of this class, the method to read its value also increments it. Each time it is read in lines 76 through 80, it is incremented and the output of the execution proves that there is only a single variable shared by all objects of this class.
- `static` is used when only one copy of the variable is needed in the program. This can be considered as an optimization. Understand this program, especially the `static` variable, then compile and run it.

#### 13.4 A String Within An Object

- Examine the program named `obstring.cpp` for our first example of an object with an embedded string. Actually, the object does not have an embedded string; it has an **embedded pointer**, but the two work so closely together.

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

```

5.  //-----class declaration part-----
6.  class wall
7.  {
8.  int    length;
9.  int    width;
10. char   *line_of_text;    //pointer variable
11. public:
12. wall(char *input_line); //constructor declaration
13. void  set(int new_length, int new_width);
14. int   get_area(void);
15. };
16.
17. //-----class implementation part-----
18. wall::wall(char *input_line)    //constructor implementation
19. {
20. length = 8;
21. width  = 8;
22. line_of_text = input_line;
23. }
24.
25. //This method will set a wall size to the two input parameters
26. void wall::set(int new_length, int new_width)
27. {
28. length = new_length;
29. width  = new_width;
30. }
31.
32. //This method will calculate and return
33. //the area of a wall instance
34. int wall::get_area(void)
35. {
36. cout<<line_of_text<< "= ";
37. return (length * width);
38. }
39.
40. //-----main program-----
41. void main()
42. {
43. //objects are instantiated with a string
44. //constant as an actual parameters
45. wall  small("of small size "),
46.       medium("of medium size "),
47.       large("of large size ");
48.
49. small.set(5, 7);
50. large.set(15, 20);
51.
52. cout<<"    Embedded string used as an object\n";
53. cout<<"    -----\n\n";
54. cout<<"Area of wall surface ";
55. cout<<small.get_area()<<"\n";
56. cout<<"Area of wall surface ";
57. cout<<medium.get_area()<<"\n";
58. //use default value of constructor
59. cout<<"Area of wall surface ";
60. cout<<large.get_area()<<"\n";
61.
62. system("pause");
63. }

```

63 lines: Output:

```

C:\bc5\bin\hohoh.exe
Embedded string used as an object
-----
Area of wall surface of small size = 35
Area of wall surface of medium size = 64
Area of wall surface of large size = 300
Press any key to continue . . .

```

- You will notice that line 10 contains a pointer to a char named `line_of_text`. The constructor contains an input parameter which is a pointer to a string which will be copied to the string named `line_of_text` within the constructor.

```
char    *line_of_text;    //pointer variable
```

- We could have defined the variable `line_of_text` as an actual array in the class, then use `strcpy()` to copy the string into the object.
- It should be pointed out that we are not limited to passing single parameters to a constructor. Any number of parameters can be passed, as will be illustrated later.
- You will notice that the three walls are defined this time, we supply a string constant as an actual parameter with each declaration which is used by the constructor to assign the string pointer some data to point to.
- When we call `get_area()` in lines 55, 57 and 60 as shown below, we get the message displayed and the area returned. It would be prudent to put this operation in separate methods since there is no apparent connection between printing the message and calculating the area, but it was written this way to illustrate that it can be done.

```
cout<<small.get_area()<<"\n";
...
cout<<medium.get_area()<<"\n";
//use default value of constructor
...
cout<<large.get_area()<<"\n";
```

- After you understand this program, compile and run.

### 13.5 An Object With An Internal Pointer

- Next, study program `obinptr.cpp` carefully. This is our first example, the program with an embedded pointer which will be used for dynamic data allocation discussion.
- Internal pointers refer to the pointer variables within the class itself.

```
1. //Program opinptr.cpp
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. //-----class declaration part-----
6. class wall
7. {
8.     int    length;
9.     int    width;
10.    int     *point;
11.    //declaration of the pointer variable
12.    public:
13.        wall(void);
14.        //constructor declaration
15.        void set(int new_length, int new_width, int stored_value);
16.        int  get_area(void) { return (length * width); }
17.        //Inline function
18.        int  get_value(void) { return *point; }
19.        //Inline function
20.        ~wall();
21.        //destructor
22. };
23.
24. //-----class implementation part-----
25. wall::wall(void)
26. //constructor implementation
27. {
28.     length = 8;
29.     width = 8;
30.     point = new int;    //new keyword
31.     *point = 112;
32. }
33.
34. //This method will set a wall size to the input parameters
35. void wall::set(int new_length, int new_width, int stored_value)
36. {
37.     length = new_length;
38.     width = new_width;
39.     *point = stored_value;
40. }
41.
42. wall::~~wall(void)
43. //destructor
```

```

44. {
45.     length = 0;
46.     width = 0;
47.     delete point;    //delete keyword
48. }
49.
50. //-----main program-----
51. void main()
52. {
53.     wall small, medium, large; //objects instance
54.
55.     small.set(5, 7, 177);
56.     large.set(15, 20, 999);
57.
58.     cout<<"Area of the small wall surface is "<<small.get_area()<<"\n";
59.     cout<<"Area of the medium wall surface is "<<medium.get_area()<<"\n";
60.     cout<<"Area of the large wall surface is "<<large.get_area()<<"\n\n";
61.     cout<<"Third variable in class wall, pointer *point\n";
62.     cout<<"-----\n";
63.     cout<<"Stored value of the small wall surface is "<<small.get_value()<<"\n";
64.     cout<<"Stored value of the medium wall surface is
65.             "<<medium.get_value()<<"\n";
66.     cout<<"Stored value of the large wall surface is "<<large.get_value()<<"\n";
67.
68.     system("pause");
69. }

```

69 lines: Output:

- In line 10 we declare a pointer named `point` to an integer variable, but it is only a pointer, there is no storage associated with it. The constructor therefore allocates storage for an integer type variable on the heap (free memory space) for use with this pointer in line 30 as shown below.
- The compiler automatically determines the proper size of the object.

```

int    *point;
...
point = new int; //new operator

```

- Three objects instantiated in line 53 each contain a **pointer** which points into the heap to **three different locations** as shown below. Each object has its own dynamically storage allocated variable for its own private use.

```

wall small, medium, large; //objects instance

```

- Moreover each has a value of 112 stored in it dynamically because line 31 stores that value in each of the three locations, once for each call to the constructor as shown below:

```

*point = 112;

```

- In a real program production, it would be mandatory to test that the value of the return pointer is not NULL to assure that the data actually did get allocated.
- The method named `set ( )` has three parameters associated with it and the third parameter is used to set the value of the new dynamically allocated variable. There are two messages passed, one to the `small` wall and one to the `large` wall. As before, the `medium` wall is left with its default values.
- The three areas are displayed followed by the three stored values in the dynamically allocated variables, and we finally have a program that requires a destructor in order to do a clean up.

- If we simply leave the scope of the objects as we do when leave the `main ( )` program, we will leave the three dynamically allocated variables on the heap with nothing pointing to them.
- They will be inaccessible and will therefore represent wasted storage on the heap. For that reason, as shown below, the destructor is used to delete the variable which the pointer named `point` is referencing, as each object goes out of existence as in line 47.

```
delete point; //delete operator
```

- In this case, lines 45 and 46 as shown below assign zero to variables that will be automatically deleted. Even though these lines of code really do no good, they are legal statements.

```
length = 0;
width = 0;
```

- Actually, in this particular case, the variables will be automatically reclaimed when we return to the operating system because all program cleanups are done for us at that time.
- This is an example of good programming practice that of cleaning up when you no longer need the dynamically allocated variables to free up the heap.
- Compile and run this program.

### 13.6 Dynamically Allocated Object

- Examine the program named `obdyna.cpp` carefully for our first look at dynamically allocated object. Notice that the pointer is in the main program instead of within the class as in previous example.

```
1. //Program obdyna.cpp
2. //dynamically allocated object
3.
4. #include <iostream.h>
5. #include <stdlib.h>
6.
7. //-----class declaration part-----
8. class wall
9. {
10. int length;
11. int width;
12. //two member variables
13. public:
14. wall(void);
15. //constructor declaration
16. void set(int new_length, int new_width);
17. int get_area(void);
18. //two methods
19. };
20.
21. //-----class implementation part-----
22. wall::wall(void)
23. //constructor implementation
24. {
25. length = 8;
26. width = 8;
27. }
28.
29. //This method will set a wall size to the input parameters
30. void wall::set(int new_length, int new_width)
31. {
32. length = new_length;
33. width = new_width;
34. }
35.
36. //This method will calculate and return the area of a wall instance
37. int wall::get_area(void)
38. {
39. return (length * width);
40. }
41.
42. //-----main program-----
43. void main()
44. {
45. wall small, medium, large;
46. //objects are instantiated of type class wall
47. wall *point;
```



```

48. //a pointer to a class wall
49.
50. small.set(5, 7);
51. large.set(15, 20);
52.
53. point = new wall; //new operator
54. //use the defaults value supplied by the constructor
55.
56. cout<<"Use small.set(5, 7)\n";
57. cout<<"-----\n";
58. cout<<"Area of the small wall surface is "<<small.get_area()<<"\n\n";
59. cout<<"Use default/constructor value medium.set(8, 8)\n";
60. cout<<"-----\n";
61. cout<<"Area of the medium wall surface is "<<medium.get_area()<<"\n\n";
62. cout<<"Use large.set(15, 20)\n";
63. cout<<"-----\n";
64. cout<<"Area of the large wall surface is "<<large.get_area()<<"\n\n";
65. cout<<"Use default/constructor value, point->get_area()\n";
66. cout<<"-----\n";
67. cout<<"New surface area of wall "<<point->get_area()<<"\n\n";
68. cout<<"Use new value, point->set(12, 12)\n";
69. cout<<"-----\n";
70. point->set(12, 12);
71. cout<<"New surface area of wall "<<point->get_area()<<"\n";
72. delete point; //delete operator
73.
74. system("pause");
75. }

```

75 lines: Output:

```

C:\bc5\bin\hohoh.exe
Use small.set(5, 7)
Area of the small wall surface is 35
Use default/constructor value medium.set(8, 8)
Area of the medium wall surface is 64
Use large.set(15, 20)
Area of the large wall surface is 300
Use default/constructor value, point->get_area()
New surface area of wall 64
Use new value, point->set(12, 12)
New surface area of wall 144
Press any key to continue . . .

```

- In line 47 as shown below, we defined a pointer to an object of type wall named point and it is only a pointer with nothing to point to.

```

wall *point;

```

- The following code means, we dynamically allocated object storage for it in line 53, with the object being instantiated on the heap just like any other dynamically allocated variable using the new keyword.

```

point = new wall; //new keyword

```

- When the object is created in line 53, the constructor is called automatically to assign values to the two internal storage variables. Note that the constructor is not called when the pointer is defined since there is nothing to initialize. It is **called when the object is allocated**.
- Reference to the components of the object are handled in much the same way that structure references are made, through the use of the **pointer operator** as illustrated in lines 67, 70 and 71 as shown below.

```

cout<<"New surface area of wall "<<point->get_area()<<"\n\n";

```

```
...
point->set(12, 12);
cout<<"New surface area of wall "<<point->get_area()<<"\n";
```

- Alternatively, you also can use the pointer dereferencing method without the arrow such as (refer to [Module 8](#)).

```
(*point).set(12, 12);
```

- Same as:

```
point->set(12, 12)
```

- Finally, the object is deleted in line 72 as shown below and the program terminates. If there were a destructor for this class, it would be called automatically as part of the delete statement to clean up the object prior to deletion.

```
delete point; //delete keyword
```

- Notice that the use of objects is not much different from the use of structure. Compile and run this program after you have studied it thoroughly.

### 13.7 new and delete operators

- These operators provide better dynamic memory allocation compared to `malloc()` and `free()` function calls used in C. Consider the following code:

```
TypeName *typeNamePtr;
```

- In ANSI C (ISO/IEC C), to dynamically create an object of type `TypeName`, you would write like this:

```
typeNamePtr = malloc(sizeof(TypeName));
```

- This requires a function call to `malloc()`. In C++ you simply write:

```
typeNamePtr = new TypeName;
```

- The `new` operator automatically creates a class's object of the proper size, calls the constructor for the object (if any) and returns a pointer of the correct type. If there is unavailable space, 0 pointer is returned.
- Next, to free the space for the allocated class's object, after has been used, the `delete` operator is used as follows:

```
delete typeNamePtr;
```

- An initializer also is permitted for newly created object, for example:

```
double *Ptr = new double (4.2345);
```

- This initializes a newly created double object to 4.2345. Example for an array:

```
ObjectPtr = new int[10][10];
```

- Then can be deleted by using:

```
delete[] ObjectPtr;
```

- `new` and `delete` automatically invokes the constructor and destructor classes respectively.

### 13.8 An Object With A Pointer To Another Object

- The program named `oblis.cpp` contains an object with internal reference to another object of its own class.

- This is the standard structure used for a singly linked list and we will keep the use of it very simple in this program.

```

1. //Program oblist.cpp
2. //object and list
3.
4. #include <iostream.h>
5. #include <stdlib.h>
6.
7. //-----class declaration part-----
8. class wall
9. {
10.     int length;
11.     int width;
12.     wall *another_wall;    //pointer variable
13. public:
14.     wall(void);
15.     //constructor declaration
16.     void set(int new_length, int new_width);
17.     int get_area(void);
18.     void point_at_next(wall *where_to_point);
19.     wall *get_next(void);
20. };
21.
22. //-----class implementation part-----
23. wall::wall(void)
24. //constructor implementation
25. {
26.     length = 8;
27.     width = 8;
28.     another_wall = NULL;
29. }
30.
31. //This method will set a wall size to the input parameters
32. void wall::set(int new_length, int new_width)
33. {
34.     length = new_length;
35.     width = new_width;
36. }
37.
38. //This method will calculate and return the area of a wall instance
39. int wall::get_area(void)
40. {
41.     return (length * width);
42. }
43.
44. //this method causes the pointer to point to the input parameter
45. void wall::point_at_next(wall *where_to_point)
46. {
47.     another_wall = where_to_point;
48. }
49.
50. //this method returns the wall the current one points to
51. wall *wall::get_next(void)
52. {
53.     return another_wall;
54. }
55.
56. //-----main program-----
57. void main()
58. {
59.     wall small, medium, large;
60.     //objects are instantiated, of type class wall
61.     wall *wall_pointer;
62.     //wall *point;
63.     //a pointer to a wall
64.
65.     small.set(5, 7);
66.     large.set(15, 20);
67.
68.     //point = new wall;
69.     //use the defaults value supplied by the constructor
70.     cout<<"Using small.set(5, 7):\n";
71.     cout<<"-----\n";
72.     cout<<"Area of the small wall surface is "<<small.get_area()<<"\n\n";
73.     cout<<"Using default/constructor value\n";
74.     cout<<"-----\n";
75.     cout<<"Area of the medium wall surface is "<<medium.get_area()<<"\n\n";
76.     cout<<"Using large.set(15, 20):\n";

```

```

77.     cout<<"-----\n";
78.     cout<<"Area of the large wall surface is "<<large.get_area()<<"\n\n";
79.
80.     small.point_at_next(&medium);
81.     medium.point_at_next(&large);
82.
83.     wall_pointer = &small;
84.     wall_pointer = wall_pointer->get_next();
85.
86.     cout<<"The wall's pointer pointed to has area "<<wall_pointer
87.                                     ->get_area()<<"\n";
88.
89.     system("pause");
90. }

```

90 lines: Output:

- The constructor contains the statement in line 28 as shown below, which assigns the pointer named `another_box` the value of `NULL` to initialize the pointer. Don't allow any pointer to point off into space, but initialize all pointers to point to something.

```
another_wall = NULL;
```

- By assigning the pointer within the constructor, you guarantee that every object of this class will automatically have its pointer initialized.
- Two additional methods are declared in lines 18 and 19 as in the following code with the one in line 19 having a construct we have not yet mentioned in this Module.
- This method returns a pointer to an object of the wall class. As you are aware, you can return a pointer to a struct in standard C, and this is a parallel construct in C++.

```
void    point_at_next(wall *where_to_point);
wall    *get_next(void);
```

- As shown below, the implementation in lines 51 through 54 returns the pointer stored as a member variable within the object. We will see how this is used when we get to the actual program.

```

wall *wall::get_next(void)
{
    return another_wall;
}

```

- An extra pointer named `wall_pointer` is defined in the main program for later use and in line 80 we make the embedded pointer within the small wall point to the medium wall. Line 81 makes the embedded pointer within the medium wall point to the large wall as shown below:

```

small.point_at_next(&medium);
medium.point_at_next(&large);

```

- We have effectively generated a linked list with three elements. In line 83 we make the extra pointer point to the small wall. Continuing in line 84 we use it to refer to the small wall and update it to the value contained in the small wall which is the address of the medium wall.

- The code segment is shown below:

```

wall_pointer = &small;
wall_pointer = wall_pointer->get_next();

```

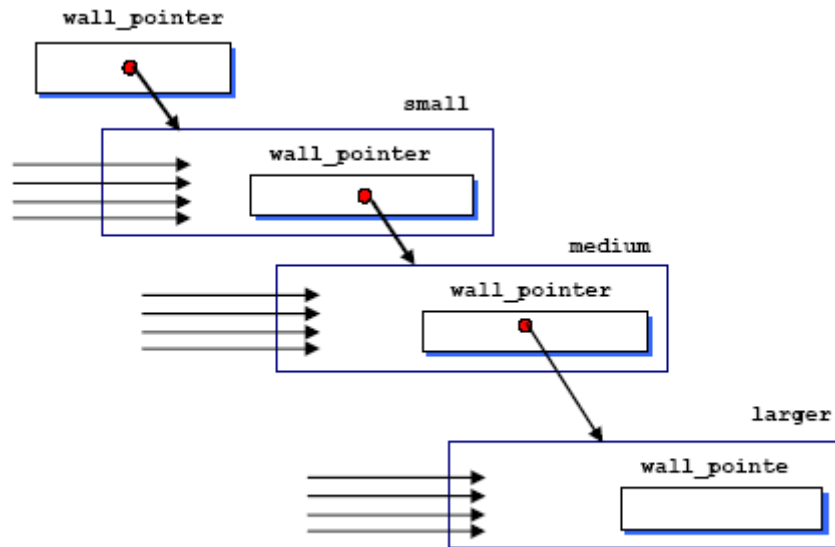


Figure 13.2: Graphical representation of the data space after the program execution

- We have therefore traversed from one element of the list to another by sending a message to one of the objects. If line 84 were repeated exactly as shown, it would cause the extra pointer to refer to the large wall, and we would have traversed the entire linked list which is only composed of three elements.
- Figure 13.2 is a graphical representation of a portion of each object data space following the program execution.
- Compile and run this program in preparation for our next program example.
- Keep in mind that a better solution for list can be found in Tutorial #4, the Standard Template Library, using **list Containers**.

### 13.9 this keyword

- The pointer **this** is defined within any object as **being a pointer to the object** in which it is contained. It is a pointer and explicitly defined as:

```

class_name
ame
*this;

```

- And is initialized to point to the object for which the member function is invoked. This pointer is **most useful when working with pointers** and especially with a **linked list** when you need to reference a pointer to the object you are inserting into the list.
- The pointer **this** is available for this purpose and can be used in any object. Actually the proper way to refer to any variable within a list is through the use of the predefined pointer **this**, by writing:

```

this->variable_name

```

- But the compiler assumes the pointer is used, and we can simplify every reference by omitting the pointer.
- If using explicitly, each object can determine its own address by using **this** keyword.
- **this** pointer can be used to prevent an object from being assigned to itself.
- The following example demonstrates the use of the **this** pointer explicitly to enable a member function of class **ThiPoint** to display the private data **c** of a **ThiPoint** object.

```

//using the this pointer explicitly
//to refer to object members
#include <iostream.h>

```

```

#include <stdlib.h>

class ThiPoint
{
    int c;
public:
    ThiPoint(int);
    void display();
};

ThiPoint::ThiPoint(int a){ c = a;} //just a constructor

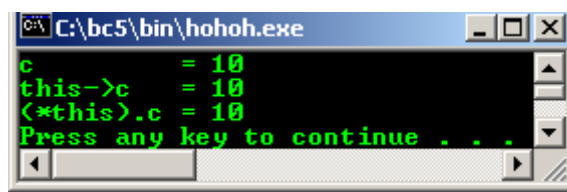
void ThiPoint::display()
{
    cout<<"c          = "<<c<<endl;
    cout<<"this->c    = "<<this->c<<endl;
    cout<<"(*this).c = "<<(*this).c<<endl;
    //use parentheses for (*this).c because
    //dot has higher precedence than *
}

void main(void)
{
    ThiPoint b(10);

    b.display();
    system("pause");
}

```

**Output:**



### 13.10 A Simple Link List Of Objects

- Examine the program named oblink.cpp carefully and this program is a complete example of a linked list written in a simple object oriented notation.
- This program is very similar to the last one, oblist.cpp until we get to the main() program.

```

1. //Program oblink.cpp,
2. //object link list
3. #include <iostream.h>
4. #include <stdlib.h>
5.
6. //-----class declaration part-----
7. class wall
8. {
9.     int length;
10.    int width;
11.    wall *another_wall;    //pointer variable
12. public:
13.     wall(void);
14.     //constructor declaration
15.     void set(int new_length, int new_width);
16.     int get_area(void);
17.     void point_at_next(wall *where_to_point);
18.     wall *get_next(void);
19. };
20.
21. //-----class implementation part-----
22. wall::wall(void)
23. //constructor implementation
24. {
25.     length = 8;
26.     width = 8;
27.     another_wall = NULL;
28. }
29.
30. //This method will set a wall size to the input parameters
31. void wall::set(int new_length, int new_width)

```

```

32. {
33.     length = new_length;
34.     width = new_width;
35. }
36.
37. //This method will calculate and return the area of a wall instance
38. int wall::get_area(void)
39. {
40.     return (length * width);
41. }
42.
43. //this method causes the pointer to point to the input parameter
44. void wall::point_at_next(wall *where_to_point)
45. {
46.     another_wall = where_to_point;
47. }
48.
49. //this method returns the wall the current one points to
50. wall *wall::get_next(void)
51. {
52.     return another_wall;
53. }
54.
55. //-----main program-----
56. void main()
57. {
58.     wall *start = NULL;           //always point to the start of the list
59.     wall *temp = NULL;           //working pointer, initialize with NULL
60.     wall *wall_pointer;           //use for object wall instances
61.
62.     //Generate the list
63.     for(int index = 0; index < 8; index++)
64.     {
65.         wall_pointer = new wall;   //new object instances
66.         wall_pointer->set(index+1, index+3);
67.
68.         if(start == NULL)
69.             start = wall_pointer;   //first element in list
70.         else
71.             temp->point_at_next(wall_pointer); //next element, link list
72.
73.         temp = wall_pointer;
74.         //print the list
75.         cout<<"Starting with wall_pointer
76.             ->set("<<(index+1)<<","<<(index+3)<<")<<"\n";
77.         cout<<"      New Wall's surface area is " <<temp->get_area() << "\n";
78.     }
79.
80.     //clean up
81.     temp = start;
82.     do {
83.         temp = temp->get_next();
84.         delete start;
85.         start = temp;
86.     } while (temp != NULL);
87.
88.     system("pause");
89. }

```

**89 lines: Output:**

```

C:\bc5\bin\hohoh.exe
Starting with wall_pointer->set(1,3)
    New Wall's surface area is 3
Starting with wall_pointer->set(2,4)
    New Wall's surface area is 8
Starting with wall_pointer->set(3,5)
    New Wall's surface area is 15
Starting with wall_pointer->set(4,6)
    New Wall's surface area is 24
Starting with wall_pointer->set(5,7)
    New Wall's surface area is 35
Starting with wall_pointer->set(6,8)
    New Wall's surface area is 48
Starting with wall_pointer->set(7,9)
    New Wall's surface area is 63
Starting with wall_pointer->set(8,10)
    New Wall's surface area is 80
Press any key to continue . . .

```

- You will recall that in the last program the only way we had to set or use the embedded pointer was through the use of two methods named `point_at_next()` and `get_next()` which are listed in lines 17 and 18 of this program.

```

void    point_at_next(wall *where_to_point);
wall    *get_next(void);

```

- We will use these to build up our linked list then traverse and print the list. Finally, we will **delete** the entire list to free the space on the heap.
- In lines 58 through 60 as in the following code segment, we define three pointers for use in the program as shown below. The pointer named `start` will always point to the beginning of the list, but `temp` will move down through the list as we creating it.

```

wall *start = NULL;           //always point to the start of the list
wall *temp = NULL;            //working pointer, initialize with NULL
wall *wall_pointer;           //use for object wall instances

```

- The pointer named `wall_pointer` will be used for the instantiated of each object. We execute the loop in lines 63 through 78 as shown below, to generate the list where line 65 dynamically allocates a new object of the `wall` class and line 66 fills it with some data for illustration.

```

for(int index = 0; index < 8; index++)
{
    wall_pointer = new wall;    //new object instances
    wall_pointer->set(index+1, index+3);
    ...
    ...
    ...
}

```

- If this is the first element in the list, the `start` pointer is set to point to this element, but if the elements already exist, the last element in the list is assigned to point to the new element. In either case, the `temp` pointer is assigned to point to the last element of the list, in preparation for adding another element if any.
- In line 73, the pointer named `temp` is caused to point to the first element at the beginning and it is used to increment its way through the list by updating itself in line 71 during each pass through the loop. When `temp` has the value of `NULL`, which it gets from the list, we are finished traversing the list.
- Finally, we delete the entire list by starting at the beginning and deleting one element each time we pass through the loop in lines 81 through 86. The code segment is shown below:

```

temp = start;
do{
    temp = temp->get_next();
    delete start;
    start = temp;
} while (temp != NULL);

```

- This program generates a linked list of ten elements, each element being an object of class `wall`.



- Compile and run this program example.
- A better solution for list can be found in Part III of this tutorial, the Standard Template Library, using **list Containers**.

### 13.11 Nesting Objects

- Examine the program named `obnest.cpp` for an example of nesting classes which results in nested objects.
- This program example contains a class named `box` which contains an object of another class embedded within it in line 25, the `mail_info` class as shown below. It is depicted graphically in figure 13.3.
- This object is available for use only within the class implementation of `box` because that is where it is defined, it is local scope.

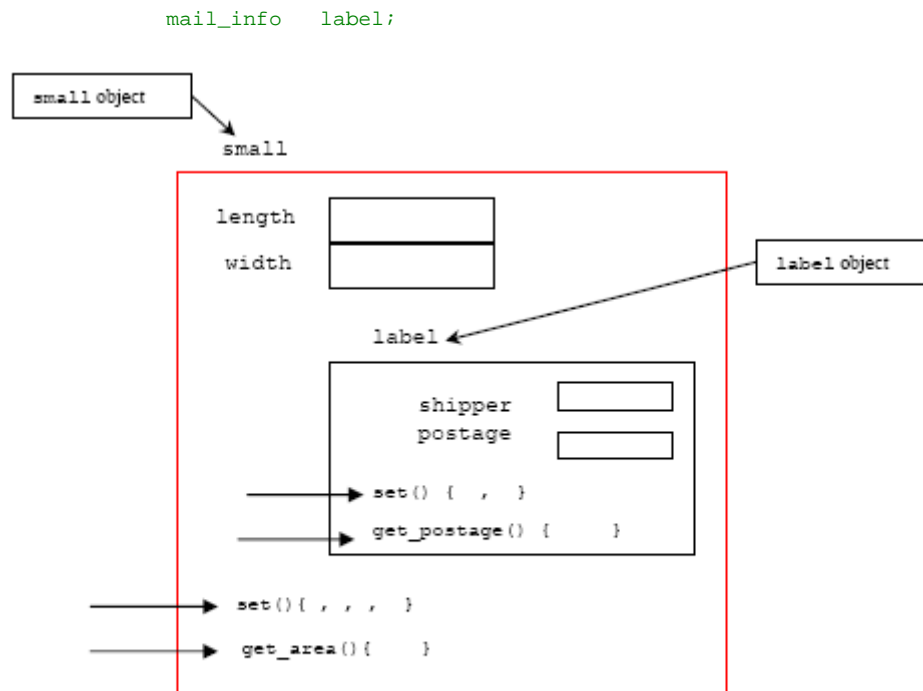


Figure 13.3: Graphical description of the class nesting

- The `main()` program has objects of class `box` defined but no objects of class `mail_info`, so the `mail_info` class cannot be referred to in the `main()` program.
- In this case, the `mail_info` class object is meant to be used internally to the `box` class and one example is given in line 32 as shown below where a message is sent to the `label.set()` method to initialize the variables.

```
label.set(ship, post);
```

- Additional methods could be used as needed, but these are given as an illustration of how they can be called.

```
1. //Program obnest.cpp
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. //-----first class declaration-----
6. class mail_info
7. {
8.     int    shipper;
9.     int    postage;
10. public:
11.     void set(int input_class, int input_postage)
12.     {
13.         shipper = input_class;
14.         postage = input_postage;
15.     }
16.     int get_postage(void)
```

```

17.     { return postage;}
18.     };
19.
20.     //-----Second class declaration-----
21.     class box
22.     {
23.     int   length;
24.     int   width;
25.     mail_info label;
26.
27.     public:
28.     void set(int l, int w, int ship, int post)
29.     {
30.     length = l;
31.     width = w;
32.     label.set(ship, post);
33.     //Accessing the first class, mail_info set() method
34.     }
35.
36.     int get_area(void) { return (length * width);}
37.     };
38.
39.     //-----main program-----
40.     void main()
41.     {
42.     box      small, medium, large;    //object instances
43.
44.     small.set(2,4,1,35);
45.     medium.set(5,6,2,72);
46.     large.set(8,10,4,98);
47.
48.     cout<<"Normal class-->small.get_area()\n";
49.     cout<<"-----\n";
50.     cout<<"Area of small box surface is "<<small.get_area()<< "\n\n";
51.     cout<<"Normal class-->medium.get_area()\n";
52.     cout<<"-----\n";
53.     cout<<"Area of medium box surface is "<<medium.get_area() << "\n\n";
54.     cout<<"Normal class-->large.get_area()\n";
55.     cout<<"-----\n";
56.     cout<<"Area of large box surface is "<<large.get_area()<<"\n\n";
57.
58.     system("pause");
59.     }

```

59 lines: Output:

```

C:\bc5\bin\hohoh.exe
Normal class-->small.get_area()
-----
Area of small box surface is 8
Normal class-->medium.get_area()
-----
Area of medium box surface is 30
Normal class-->large.get_area()
-----
Area of large box surface is 80
Press any key to continue . . .

```

- The fact is that there are never any objects of the mail\_info class declared directly in the main() program, they are inherently declared when the enclosing objects of class box are declared.
- The objects of the mail\_info class could be declared and used in the main() program if needed, but they are not in this program example. In order to be complete, the box class should have one or more methods to use the information stored in the object of the mail\_info class.
- Compile and run this program.

### 13.12 Operator Overloading

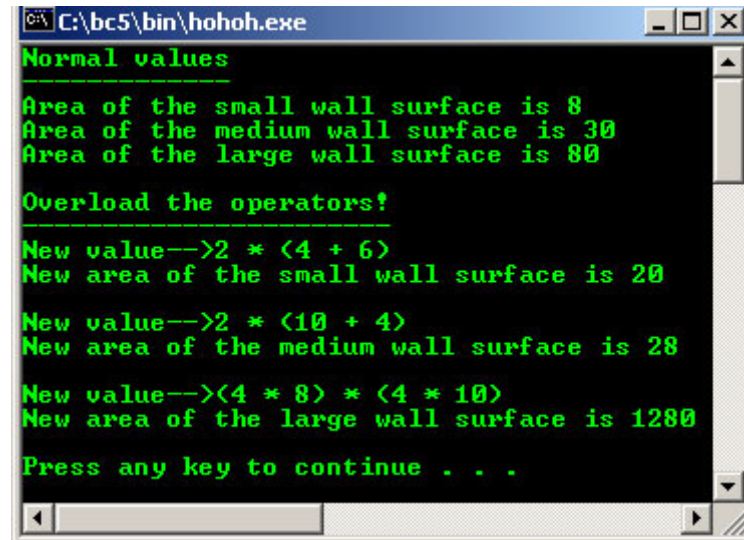
- Examine the program named opverlod.cpp carefully; this program contains examples of operators overloading. This allows you to define a class of objects and **redefine** the use of the normal operators.

```

1. //Program opverlod.cpp, operator overloading
2. #include <iostream.h>
3. #include <stdlib.h>
4.
5. //-----class declaration part-----
6. class wall
7. {
8.     public:
9.         int length;
10.        int width;
11.
12.    public:
13.        void set(int l,int w) {length = l; width = w;}
14.        int get_area(void) {return length * width;}
15.        //operator overloading
16.        friend wall operator + (wall aa, wall bb); //add two walls
17.        friend wall operator + (int aa, wall bb); //add a constant to a wall
18.        friend wall operator + (int aa, wall bb);
19.        //multiply a wall by a constant
20. };
21.
22. //-----class implementation part-----
23. wall operator + (wall aa, wall bb)
24. //add two walls widths together
25. {
26.     wall temp;
27.     temp.length = aa.length;
28.     temp.width = aa.width + bb.width;
29.     return temp;
30. }
31.
32. wall operator + (int aa, wall bb)
33. //add a constant to wall
34. {
35.     wall temp;
36.     temp.length = bb.length;
37.     temp.width = aa + bb.width;
38.     return temp;
39. }
40.
41. wall operator * (int aa, wall bb)
42. //multiply wall by a constant
43. {
44.     wall temp;
45.     temp.length = aa * bb.length;
46.     temp.width = aa * bb.width;
47.     return temp;
48. }
49.
50. void main()
51. {
52.     wall small, medium, large; //object instances
53.     wall temp;
54.
55.     small.set(2,4);
56.     medium.set(5,6);
57.     large.set(8,10);
58.
59.     cout<<"Normal values\n";
60.     cout<<"-----\n";
61.     cout<<"Area of the small wall surface is "<<small.get_area()<<"\n";
62.     cout<<"Area of the medium wall surface is "<<medium.get_area()<<"\n";
63.     cout<<"Area of the large wall surface is "<<large.get_area()<<"\n\n";
64.     cout<<"Overload the operators!"<<"\n";
65.     cout<<"-----"<<endl;
66.     temp = small + medium;
67.     cout<<"New value-->2 * (4 + 6)\n";
68.     cout<<"New area of the small wall surface is "<<temp.get_area()<<"\n\n";
69.     cout<<"New value-->2 * (10 + 4) \n";
70.     temp = 10 + small;
71.     cout<<"New area of the medium wall surface is "<<temp.get_area()<<"\n\n";
72.     cout<<"New value-->(4 * 8) * (4 * 10)\n";
73.     temp = 4 * large;
74.     cout<<"New area of the large wall surface is "<<temp.get_area()<<"\n\n";
75.
76.     system("pause");
77. }

```

77 lines: Output:



```
C:\bc5\bin\hohoh.exe
Normal values
Area of the small wall surface is 8
Area of the medium wall surface is 30
Area of the large wall surface is 80

Overload the operators!
New value-->2 * (4 + 6)
New area of the small wall surface is 20
New value-->2 * (10 + 4)
New area of the medium wall surface is 28
New value-->(4 * 8) * (4 * 10)
New area of the large wall surface is 1280
Press any key to continue . . .
```

- The end result is that objects of the new class can be used in as natural as the predefined types. In fact, they seem to be a part of the language rather than your own add-on.
- In this case we overload the + operator and the \* operator, with the declarations in lines 16 through 18 as shown below, and the definitions in lines 23 through 47. The methods are declared as friend functions, so we can use the double parameter function as listed.
- If we do not use the friend construct, the function would be a part of one of the objects and that object would be the object to which the message was sent.

```
friend wall operator + (wall aa, wall bb);    //add two walls
friend wall operator + (int aa, wall bb);    //add a constant to a wall
friend wall operator + (int aa, wall bb);
```

- By including the friend construct allows us to separate this method from the object and call the method with **infix notation**. Using this technique, it can be written as:

```
object1 + object2
```

- Rather than:

```
object1.operator + (object2)
```

- Also, without the friend construct we could not use an overloading with an int type variable for the first parameter because we can not send a message to an integer type variable such as

```
int.operator + (object)
```

- Two of the three operators overloading use an int for the first parameter so it is necessary to declare them as friend functions.
- There is no upper limit to the number of overloading for any given operator. Any number of overloading can be used provided the parameters are different for each particular overloading.
- As shown below, the header in line 23, illustrates the first overloading where the + operator is overloaded by giving the return type followed by the keyword operator and the operator we wish to overload.

```
wall operator + (wall aa, wall bb)
```

- The two formal parameters and their types are then listed in the parentheses and the normal function operations are given in the implementation of the functions in lines 25 through 30 as shown below.
- Notice that the implementation of the **friend** functions is not actually a part of the class because the class name is not prepended onto the method name in line 22.

```
{
    wall temp;
```

```

        temp.length = aa.length;
        temp.width = aa.width + bb.width;
        return temp;
    }

```

- There is nothing unusual about this implementation; it should be easily understood by you at this point. For purposes of illustration, some simple mathematical operations are performed in the method implementation, but any desired operations can be done.
- The biggest difference occurs in line 65 as shown below where this method is called by using the **infix notation** instead of the usual message sending format.

```
temp = small + medium;
```

---

### Infix Expression:

Any expression in the standard form such as " $4*2-6/3$ " is an Infix (In order) expression.

### Postfix Expression:

The Postfix (Post order) form of the above expression is " $42*63/-$ ".

---

- Since the variables `small` and `medium` are objects of the `wall` class, the system will search for a way to use the `+` operator on two objects of class `wall` and will find it in the overloaded operator `+` method we have just discussed.
- In line 70 as shown below, we ask the system to add an `int` type constant to a `small` object of class `wall`, so the system finds the other overloading of the `+` operator beginning in line 31 through 38 to perform this operation.

```
temp = 10 + small;
```

- In line 72 as shown below, we ask the system to use the `*` operator to do something to an `int` constant and a large object of class `wall`, which it satisfies by finding the method in lines 40 through 47.

```
temp = 4 * large;
```

- Note that it would be illegal to attempt to use the `*` operator the other way around, namely `large * 4` since we did not define a method to use the two types in that **order**. Another overloading could be given with reversed types, and we could then use the reverse order in a program.
- You will notice that when using operator overloading, we are also using **function name** overloading since some of the function names are same. When we use operator overloading in this manner, we actually make our programs look like the class is a natural part of the language since it is integrated into the language so well.
- Each new part we study has its pitfalls which must be warned against and the part of operator overloading seems to have the record for pitfalls since it is so prone to misuse and has several problems.
- The overloading of operators is **only** available for classes; you cannot redefine the operators for the predefined basic data types.
- The preprocessing symbols `#` and `##` cannot be overloaded.
- The `=`, `[ ]`, `( )`, and `→` operators can be overloaded only as non-static member functions. These operators cannot be overloaded for `enum` types. Any attempt to overload a global version of these operators results in a compile-time error.
- The keyword **operator** followed by the **operator symbol** is called the operator function name; it is used like a normal function name when defining the new (overloaded) action for the operator.
- The operator function cannot alter the number of arguments or the precedence and associativity rules applying to normal operator use.
- Table 13.1 and 13.2 list the operators that can be overloaded and cannot be overloaded respectively.
- Which method is used is implementation dependent, so you should use them in such a way that it doesn't matter which is used. Compile and run the program `opverlod.cpp` before continuing on the next program example.

**Operators**

+	-	*	/	%
^	&		~	!
=	<	>	+=	-=
*=	/=	%=	^=	&=
=	<<	>>	>>=	<<=
==	!=	<=	>=	&&
	++	--	->*	,
→	[ ]	( )	new	delete

Table 13.1: Operators that can be overloaded

Operators			
.	.*	?:	::

Table 13.2: Operators that cannot be overloaded

### 13.13 Function Overloading

- Examine the program named `funovlod.cpp`. This is an example of function name overloading within a class. In this program, the constructor is overloaded as well as one of the methods to illustrate what can be done.

```

1.    //Program funovlod.cpp, function overloading
2.    #include <iostream.h>
3.    #include <stdlib.h>
4.
5.    //-----class declaration part-----
6.    class many_names
7.    {
8.    int   length;
9.    int   width;
10.   public:
11.   many_names(void);
12.   many_names(int len);
13.   many_names(int len, int wid);
14.   //constructors with different number and type
15.   //of parameter list - overloaded functions
16.   void display(void);
17.   void display(int one);
18.   void display(int one, int two);
19.   void display(float number);
20.   //methods with different number and type
21.   //of parameter list - overloaded functions
22.   };
23.
24.   //-----implementation part-----
25.   many_names::many_names(void)    //void
26.   {
27.   length = 8;
28.   width = 8;
29.   }
30.
31.   many_names::many_names(int len) //one parameter
32.   {
33.   length = len;
34.   width = 8;
35.   }
36.
37.   many_names::many_names(int len, int wid)    //two parameter
38.   {
39.   length = len;
40.   width = wid;
41.   }
42.
43.   void many_names::display(void)    //void for display
44.   {
45.   cout<<"From void display function, Area = "<<length * width<<"\n";
46.   }

```

```

47.
48. void many_names::display(int one) // 1 parameter
49. {
50. cout<<"From int display function, Area = "<<length * width<<"\n";
51. }
52.
53. void many_names::display(int one, int two) //2 parameters
54. {
55. cout<<"From two int display function, Area = "<<length * width<<"\n";
56. }
57.
58. void many_names::display(float number) //1 parameter
59. {
60. cout<<"From float display function, Area = "<<length * width<<"\n";
61. }
62.
63. //-----main program-----
64. main()
65. {
66. many_names small, medium(10), large(12,15);
67. int gross = 144;
68. float pi = 3.1415, payroll = 12.50;
69.
70. cout<<"Guess, which function that they invoked???\n";
71. cout<<"-----\n";
72. cout<<"-->small.display()\n";
73. small.display();
74. cout<<"\n-->small.display(100)\n";
75. small.display(100);
76. cout<<"\n-->small.display(gross,100)\n";
77. small.display(gross,100);
78. cout<<"\n-->small.display(payroll)\n";
79. small.display(payroll);
80. cout<<"\n-->medium.display()\n";
81. medium.display();
82. cout<<"\n-->large.display(pi)\n";
83. large.display(pi);
84.
85. system("pause");
86. }

```

86 lines: Output:

```

C:\bc5\bin\hohoh.exe
Guess, which function that they invoked???
-----
-->small.display()
From void display function, Area = 64
-->small.display(100)
From int display function, Area = 64
-->small.display(gross,100)
From two int display function, Area = 64
-->small.display(payroll)
From float display function, Area = 64
-->medium.display()
From void display function, Area = 80
-->large.display(pi)
From float display function, Area = 180
Press any key to continue . . .

```

- This program illustrates some of the overloaded function names and the rules for their use. You will recall that the function selected is based on the **number** and **types** of the formal parameters only. The **type of the return value** is not significant in overload resolution.
- In this case there are three constructors. The constructor which is actually called is selected by the number and types of the parameters in the definition.
- In line 66 of the main program the three objects are declared, each with a different number of parameters and inspection of the results will indicate that the correct constructor was called based on the number of parameters. The code segment is:

```
many_names    small, medium(10), large(12, 15);
```

- In the case of the other overloaded methods, the number and type of parameters is clearly used to select the proper method. You will notice that the one method uses a single integer and another uses a single float type variable, but the system is able to select the correct one.
- As many overloading as desired can be used provided that all of the **parameter patterns are unique**.
- Other example is the << operator which is part of the cout class, which operates as an overloaded function since the way it outputs data is a function of the type of its input variable or the field we ask it to display.
- Many programming languages have overloaded output functions so you can output any data with the same function name. Compile and run this program.

### 13.14 Default Methods

- Examine the program deftmeth.cpp carefully, illustrating those methods provided by the compiler, and why you sometimes can't use the defaults but need to write your own, to do the job the defaults were intended to do for you.

```
1. //program deftmeth.cpp, default method
2. #include <iostream.h>
3.
4. #include <string.h>
5. #include <stdlib.h>
6.
7. //-----class declaration part-----
8. class def
9. {
10.     int    size;           //A simple stored value
11.     char   *string;        //A name for the stored data
12. public:
13.     //This overrides the default constructor
14.     def(void);
15.
16.     //This overrides the default copy constructor
17.     def(def &in_object);
18.
19.     //This overrides the default assignment operator
20.     def &operator=(def &in_object);
21.
22.     //This destructor should be required with dynamic allocation
23.     ~def(void);
24.
25.     //And finally, a couple of ordinary methods
26.     void set_data(int in_size, char *in_string);
27.     void get_data(char *out_string);
28. };
29.
30. //-----class implementation-----
31. def::def(void)
32. {
33.     size = 0;
34.     string = new char[2];
35.     strcpy(string, "");
36. }
37.
38. def::def(def &in_object)
39. {
40.     size = in_object.size;
41.     string = new char[strlen(in_object.string) + 1];
42.     strcpy(string, in_object.string);
43. }
44.
45. def& def::operator=(def &in_object)
46. {
47.     delete [] string;
48.     size = in_object.size;
49.     string = new char[strlen(in_object.string) + 1];
50.     strcpy(string, in_object.string);
51.     return *this; //this pointer
52. }
53.
54. def::~~def(void)
55. {
56.     delete [] string;
```

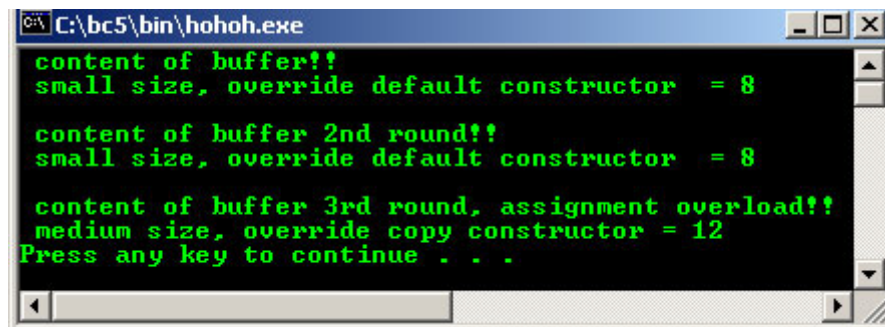


```

57. }
58.
59. void def::set_data(int in_size, char *in_string)
60. {
61.     size = in_size;
62.     delete [] string;
63.     string = new char[strlen(in_string) + 1];
64.     strcpy(string, in_string);
65. }
66.
67. void def::get_data(char *out_string)
68. {
69.     char temp[10];
70.     strcpy(out_string, string);
71.     strcat(out_string, " = ");
72.     itoa(size, temp, 10);
73.     strcat(out_string, temp);
74. }
75.
76. //-----main program-----
77. void main()
78. {
79.     char buffer[80];
80.     def my_data;
81.     my_data.set_data(8, " small size, override default constructor ");
82.     my_data.get_data(buffer);
83.     cout<<" content of buffer!!\n"<<buffer<<"\n";
84.
85.     def more_data(my_data);
86.     more_data.set_data(12, " medium size, override copy constructor");
87.     my_data.get_data(buffer);
88.     cout<<"\n content of buffer 2nd round!!\n"<<buffer<<"\n";
89.
90.
91.     my_data = more_data;
92.     my_data.get_data(buffer);
93.     cout<<"\n content of buffer 3rd round, assignment
94.                                     overload!!\n"<<buffer<<"\n";
95.
96.     system("pause");
97. }

```

97 lines: Output:



- Even if you do not include any constructors or operator overloading, you get a few defined automatically by the compiler.
- Before we actually look at the program, a few rules must be followed in order to deliver a useful implementation such as:
  - If no constructors are defined by the writer of a class, the compiler will automatically generate a **default constructor** and a **copy constructor**. Both of these constructors will be defined for you later.
  - If the class author includes any **constructor** in the class, the default constructor will not be supplied by the compiler.
  - If the class author does not include a **copy constructor**, the compiler will generate one, but if the writer includes a copy constructor, the compiler will not generate one automatically.
  - If the class author includes an **assignment operator**, the compiler will not include one automatically; otherwise it will generate a default assignment operator.

- Any class declared and used in a C++ program must have some way to construct an object because the compiler, by definition, must **call a constructor** when we **instantiate an object**. If we don't provide a constructor, the compiler itself will generate one that it can call during construction of the object.
- This is the **default constructor**, it is compiler work and we have used it implicitly in our programs examples. The default constructor **does not initialize** any of the member variables, but it sets up all of the internal class references it needs, and calls the **base constructor** or constructors if they exist.
- Base constructor is a constructor in the base class, and will be discussed in **inheritance** Module.
- Line 14 of the program declares a default constructor as shown below which is called when you instantiate an object with no parameters.

```
def(void);
```

- In this case, the constructor is necessary because we have an embedded string in the class that requires a **dynamic allocation** and an **initialization** of the string pointer to the NULL.
- It will take little thought to see that our constructor is much better than the default constructor which would leave us with an uninitialized pointer. The default constructor is used in line 81 of this program example (8 instead of 12) as shown below.

```
my_data.set_data(8, " small size, override default constructor ");
```

### 13.15 The Copy Constructor

- The copy constructor is generated automatically for you by the compiler if you fail to define one. It is **used to copy the contents of an object to a new object during construction of that new object**.
- If the compiler generates it for you, it will simply copy the contents of the original into the new object byte by byte, which may not be what you want.
- For simple classes with no pointers, that is usually sufficient, but in our program example, we have a pointer as a class member so a byte by byte copy would copy the pointer from one to the other and they would both be pointing to the same allocated member.
- For this reason, we declared our own copy constructor in line 16 as shown below.

```
//This overrides the default copy constructor
def(def &in_object);
...
```

- And implemented in lines 38 to 43 as shown below.

```
def::def(def &in_object)
{
    size = in_object.size;
    string = new char[strlen(in_object.string) + 1];
    strcpy(string, in_object.string)
}
...
```

- A careful study of the implementation will reveal that the new class will indeed be identical to the original, but the new class has its own string to work with.
- Since both constructors contain dynamic allocation, we must assure that the allocated data is destroyed when we are finished with the objects, so a **destructor is mandatory** as implemented in lines 54 through 57 as shown below.

```
def::~def(void)
{
    delete [] string;
}
...
```

- And the copy constructor is used in line 86 as shown below.

```
more_data.set_data(12, " medium size, override copy constructor");
```

- Generally, copy constructors are invoked whenever a copy of an object is needed such as function call by value, when returning an object from a called function or when initializing an object to be a copy of another object of the same class.

### 13.16 The Assignment Operator

- Also, an assignment operator is required for this program, because the default assignment operator simply copies the source object to the destination object byte by byte. This would result in the same problem we had with copy constructor.
- The assignment operator is declared in line 20 as shown below:

```
//This overrides the default assignment operator
def &operator=(def &in_object);
...
```

- And defined in lines 45 through 52 as shown below:

```
def& def::operator=(def &in_object)
{
    delete [] string;
    size = in_object.size;
    string = new char[strlen(in_object.string) + 1];
    strcpy(string, in_object.string);
    return *this;
}
...
```

- Here, we de-allocate the old string in the existing object prior to allocating room for the new text and copying the text from the source object into the new object. The assignment operator is used in line 91 as shown below.

```
my_data = more_data;
```

- It should be fairly obvious to you that when a class is defined which includes any sort of **dynamic allocation**, the above **three methods** should be included in addition to the proper destructor.
- If any of the **four entities** are omitted, the program may have terribly erratic behavior. Compile and run this program example.

#### Program Examples and Experiments

##### Example #1

- The following example tries to show the pointer and string manipulation in class. Study the program execution flow.

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

class PlayText
{
    char *NewPointerVariable;

public:
    PlayText(char *InputFromMainProgPointer);
    int GetData(void);
};

PlayText::PlayText(char *InputFromMainProgPointer)
{
    cout<<"Location: constructor implementation part\n";
    cout<<"Examine the flow of execution!!!\n";
    cout<<"String brought by object from main program is
"<<"\n"<<InputFromMainProgPointer<<"\n"<<"\n";
    cout<<"through pointer variable *InputFromMainProgPointer\n";
    cout<<"Assign this string to class member pointer variable\n";
    cout<<"*NewPointerVariable\n";
    cout<<"Next Location: Main program\n\n";

    NewPointerVariable = InputFromMainProgPointer;
}

int PlayText::GetData(void)
{
    cout<<NewPointerVariable<<" = ";
    return 100;
}
```

```

void main()
{
    PlayText  small("of small size");

    cout<<"cout string of main program \"Area of...\" mix \n";
    cout<<"with variable NewPointerVariable and silly integer value\n";
    cout<<"from implementation part by using small.GetData()....\n";
    cout<<"Go to implementation part after next cout, and the output is: \n\n";
    cout<<"Area of wall surface ";
    cout<<small.GetData()<<"\n";
    system("pause");
}

```

**Output:**

#### Example #2

- In this example we can see how efficient regarding the memory allocation and de allocation up been implemented.

```

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

class LearnPointer
{
    int *MemVarPoint;
public:
    LearnPointer(void);
    void SetData(int InputData);
    int GetData(void)
    { return *MemVarPoint; };
    ~LearnPointer(void);
};

LearnPointer::LearnPointer(void)
{
    *MemVarPoint = 100;
    cout<<"In Constructor: Address of pointer *MemVarPoint is "<<&MemVarPoint<<"\n";
}

void LearnPointer::SetData(int InputData)
{
    cout<<"In Method: Address of pointer *MemVarPoint is "<<&MemVarPoint<<"\n";
    *MemVarPoint = InputData;
}

LearnPointer::~~LearnPointer(void)
{
    delete MemVarPoint;
}

void main()
{
    LearnPointer SimpleData;
    int x;

    cout<<"First time: This is constructor value = "<<SimpleData.GetData()<<"\n\n";
}

```

```

        SimpleData.SetData(1000);
        cout<<"Set the data, call method from implementation part\n";
        cout<<"Second time: Override the constructor value = "<<SimpleData.GetData()<<"\n";
        cout<<"\nLet get data from user\n";
        cout<<"Please key in one integer value: ";
        cin>>x;
        SimpleData.SetData(x);
        cout<<"Third time: New data from user = "<<SimpleData.GetData()<<"\n";
        cout<<"\n...See...different data, at different time but"<<endl;
        cout<<"same memory (address) allocation..."<<endl;
        system("pause");
    }
}

```

Output:

```

C:\bc5\bin\hohoh.exe
In Constructor: Address of pointer *MemUarPoint is 0x19ea0ffc
First time: This is constructor value = 100

In Method: Address of pointer *MemUarPoint is 0x19ea0ffc
Set the data, call method from implementation part
Second time: Override the constructor value = 1000

Let get data from user
Please key in one integer value: 77
In Method: Address of pointer *MemUarPoint is 0x19ea0ffc
Third time: New data from user = 77

...See...different data, at different time but
same memory (address) allocation...
Press any key to continue . . .

```

### Example #3

- Compile and run the following program example of the new and delete operators.

```

//testing the new and delete keywords
#include <iostream.h>
#include <stdlib.h>

//-----class declaration part-----
class TestNewDel
{
    //member variables...
private:
    int TestVar;
    char *TestPtr;

    //member functions, constructor and destructor...
public:
    TestNewDel();
    int DisplayValue();
    char* DisplayStr();
    ~TestNewDel();
};

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

//simple function returning a value...
int TestNewDel::DisplayValue()
{
    return TestVar = 200;
}

//another function returning a string...
char* TestNewDel::DisplayStr()
{

```

```

        TestPtr = "Testing the new and delete";
        return TestPtr;
    }

//destructor...
TestNewDel::~TestNewDel()
{
    //test how destructor is invoked...
    //use static to retain previous value...
    static int y=1;
    cout<<"In Destructor, pass #"<<y<<endl;
    y++;
    //explicitly clean up...
    TestVar = 0;
}

//-----main program-----
int main()
{
    //instantiate an object...
    //constructor should be invoked...
    //with proper memory allocation...
    TestNewDel Obj1;

    cout<<"In main, testing 1...2...3..."<<endl;
    //display the data value...
    cout<<"Default constructor value assign to Obj1 = "<<Obj1.DisplayValue()<<endl;

    //invoke constructor explicitly...
    //remember to clean up later explicitly...
    TestNewDel* Obj2 = new TestNewDel;
    //reconfirm the allocation for Obj2...
    if(!Obj2)
        cout<<"Allocation to Obj2 failed!"<<endl;
    else
        cout<<"Allocation to Obj2 is OK!"<<endl;

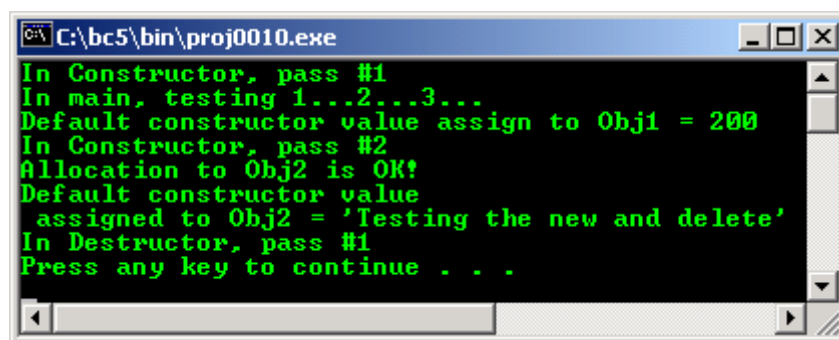
    //display the data value...
    cout<<"Default constructor value "
        <<"\n assigned to Obj2 = "<<"\' "<<Obj2->DisplayStr()<<"\' "<<endl;

    //explicitly clean up the allocation...
    delete Obj2;

    system("pause");
    return 0;
}

```

**Output:**

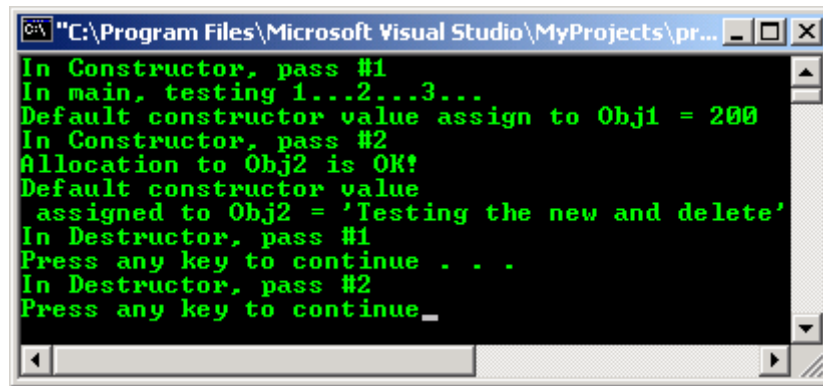


```

C:\bc5\bin\proj0010.exe
In Constructor, pass #1
In main, testing 1...2...3...
Default constructor value assign to Obj1 = 200
In Constructor, pass #2
Allocation to Obj2 is OK!
Default constructor value
assigned to Obj2 = 'Testing the new and delete'
In Destructor, pass #1
Press any key to continue . . .

```

- The output using Visual C++ 6.0 (VC60) as shown below look like the cleanup job was done properly. Destructor invoked two times that is for Obj1 and Obj2 each.
- For Borland users, if you want to see a complete output, please use its debugger such as Turbo Debugger.

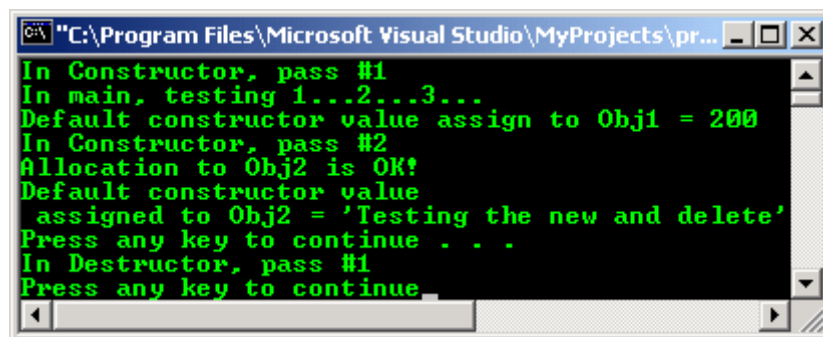


```
"C:\Program Files\Microsoft Visual Studio\MyProjects\pr...
In Constructor, pass #1
In main, testing 1...2...3...
Default constructor value assign to Obj1 = 200
In Constructor, pass #2
Allocation to Obj2 is OK!
Default constructor value
  assigned to Obj2 = 'Testing the new and delete'
In Destructor, pass #1
Press any key to continue . . .
In Destructor, pass #2
Press any key to continue _
```

- Next, comment out the following code from the program example:

```
delete Obj2;
```

- Recompile and re run the program. In this case, let see what happen if we forgot to do the clean up/de-allocation job explicitly for the new keyword use. The output using VC60 is shown below:



```
"C:\Program Files\Microsoft Visual Studio\MyProjects\pr...
In Constructor, pass #1
In main, testing 1...2...3...
Default constructor value assign to Obj1 = 200
In Constructor, pass #2
Allocation to Obj2 is OK!
Default constructor value
  assigned to Obj2 = 'Testing the new and delete'
Press any key to continue . . .
In Destructor, pass #1
Press any key to continue _
```

- Well, the clean up job was not done for Obj2, the destructor only invoked once. So, be careful of this behavior.

#### Example #4

```
//operators overloading example
#include <iostream.h>
#include <stdlib.h>
#include <assert.h>

//-----class declaration part-----
class TestArray
{
    //these overloaded << and >> must be friend of TestArray class...
    //so the overloaded << and >> of TestArray class are available
    //for the ostream and istream classes, u will learn later...
    //ostream and istream are C++ file I/O...
    //equal to operator(cout, object) function,
    friend ostream &operator<<(ostream &,TestArray &);
    //equal to operator(cin, object) function
    friend istream &operator>>(istream &,TestArray &);

public:
    //default constructor
    TestArray(int = 4);
    //copy constructor
    TestArray(const TestArray &);
    //destructor
    ~TestArray();

    //return the size of the an array
    int GetSize() const;
    //overloaded the assignment operator
    const TestArray &operator=(const TestArray &);
    //overloaded the == operator
    int operator==(const TestArray &) const;
    //overloaded the != operator
    int operator!=(const TestArray &) const;
```

```

        //overloaded the [] operator
        int &operator[](int);

    private:
        int *ptr;
        int size;
};

//-----class implementation part-----
//default constructor for TestArray class
TestArray::TestArray(int ArraySize)
{
    size = ArraySize;
    ptr = new int[size];
    assert(ptr != 0);
    for(int i = 0; i < size; i++)
        ptr[i] = 0;
}

//copy constructor for TestArray class
TestArray::TestArray(const TestArray &initial)
{
    size = initial.size;
    ptr = new int[size];
    assert(ptr != 0);
    for(int i = 0; i < size; i++)
        ptr[i] = initial.ptr[i];
}

//destructor for TestArray class
TestArray::~TestArray()
{
    delete[] ptr;
}

//get the array's size
int TestArray::GetSize() const {return size;}

//overloaded the subscript/index operator
int &TestArray::operator[](int index)
{
    assert((index >= 0) && (index < size));
    return ptr[index];
}

//== comparison, return 1 if true, 0 if false
int TestArray::operator==(const TestArray &rightside) const
{
    if (size != rightside.size)
        return 0;
    for(int i = 0; i < size; i++)
        if(ptr[i] != rightside.ptr[i])
            return 0;
    return 1;
}

//!= comparison, return 1 if true, 0 if false
int TestArray::operator!=(const TestArray &rightside) const
{
    if (size != rightside.size)
        return 1;
    for(int i = 0; i < size; i++)
        if (ptr[i] != rightside.ptr[i])
            return 1;
    return 0;
}

//overloaded assignment operator
const TestArray &TestArray::operator=(const TestArray &rightside)
{
    if(&rightside != this)
    {
        delete [] ptr;
        size = rightside.size;
        ptr = new int[size];
        assert(ptr != 0);
        for(int i = 0; i < size; i++)
            ptr[i] = rightside.ptr[i];
    }
    return *this;
}

```



```

}

//overloaded the >> operator
istream &operator>>(istream &input, TestArray &x)
{
    for(int i = 0; i < x.size; i++)
        input>>x.ptr[i];
    return input;
}

//overloaded the << operator
ostream &operator<<(ostream &output, TestArray &x)
{
    int i;
    for(i = 0; i < x.size; i++)
    {
        output<<x.ptr[i]<<' ';

        if((i+1)%10==0)
            output<<endl;
    }
    if (i % 10 !=0)
        output << endl;
    return output;
}

//-----main program-----
int main()
{
    //One(3) same as One = 3 and Two will use
    //the default constructor value, 4
    TestArray One(3), Two;

    cout<<"Array One's size is "
        <<One.GetSize()
        <<"\nWith initial data: "<<One;

    cout<<"Array Two's size is "
        <<Two.GetSize()
        <<"\nWith initial data: "<<Two;

    cout<<"\nNow, input 7 integers for the arrays:\n";
    cin>>One>>Two;
    cout<<"\nThen, the arrays content:\n"
        <<"One[]: "<<One
        <<"Two[]: "<<Two;

    cout<<"\nNew array, copy of One's array:";
    TestArray Three(One);
    cout<<"\nThe new array, Three size is "<<Three.GetSize()
        <<"\nArray initial value: "<<Three;

    cout<<"\nTesting: One == Three?\n";
    if(One == Three)
        cout<<"They are equal\n";
    else
        cout<<"They are not equal!";

    cout<<"\nTesting: One != Two?\n";
    if(One != Two)
        cout<<"They are not equal\n\n";

    cout<<"Assigning Two to One:\n";
    One = Two;
    cout<<"One: "<<One<<"Two: "<<Two;

    cout<<"\nTesting: One == Two?\n";
    if(One == Two)
        cout<<"They are equal\n";
    else
        cout<<"They are not equal!\n";

    system("pause");
    return 0;
}

```

**Output:**

- Well, as an assignment, I pass to you to repackage this program into three files named `testarray.h` as a header file, `testarray.cpp` as an implementation file and `mainarray.cpp` as main program.
- The following is the simple program example of the complex number for operator and function overloading.

```
//operator and function overloading
//the classic simple complex number example
#include <iostream.h>
#include <stdlib.h>

class complexnum
{
    private:
        double p, q;

    public:
        //constructor forming the a + bi
        //overloaded function, with two arguments...
        complexnum(double a, double b)
        {
            p = a;
            q = b;
        }

        //Constructor forming the a + 0i
        //overloaded function, with one argument...
        complexnum(double a)
        {
            p = a;
            q = 0;
        }

        //Returns the real part
        double realpart()
        {return p; }

        //Returns the complex part
        double imaginarypart()
        { return q; }

        //Addition operation of two complex numbers
        //overloaded operator +
        complexnum operator+(complexnum a)
```

```

{ return complexnum(a.p + p, a.q + q); }

//Addition a complex number and a double
//overloaded operator +
complexnum operator+(double a)
{ return complexnum(p + a, q);}

//Subtraction operation of two complex numbers...
//overloaded operator -
complexnum operator-(complexnum a)
{ return complexnum(p - a.p, q - a.q); }

//Addition a complex number and a double
//overloaded operator -
complexnum operator-(double a)
{ return complexnum(p - a, q); }
};

//display format for complex number...
//overloaded operator <<
ostream& operator<<(ostream& s, complexnum r)
{
//if no imaginary part...
if(r.imaginarypart() == 0)
    //return real part only...
    return s<<r.realpart();

//if imaginary part < 0, i.e negative...
else if(r.imaginarypart() < 0 )
{
    //and if no real part
    if(r.realpart() == 0)
        //return imaginary part only...
        return s<<r.imaginarypart()<<"i";
    else
        //return both real and imaginary parts...
        return s<<r.realpart()<<r.imaginarypart()<<"i";
}
else
{
    //and if no real part
    if(r.realpart() == 0)
        //return imaginary part only...
        return s<<r.imaginarypart()<<"i";
    else
        //return both, real and imaginary parts...
        return s<<r.realpart()<<" + "<<r.imaginarypart()<<"i";
}
}

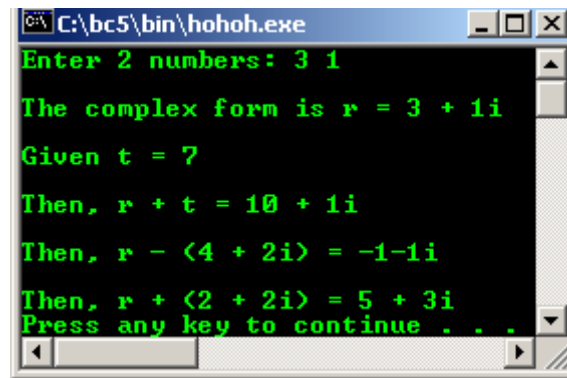
}

void main()
{
    double a,b;

    //get two numbers
    cout<<"Enter 2 numbers: ";
    cin>>a>>b;
    complexnum r = complexnum(a,b);
    cout<<"\nThe complex form is r = "<<r<<endl;
    complexnum t = complexnum(7.0);
    cout<<"\nGiven t = "<<t<<endl;
    //Addition of complex number and constant...
    cout<<"\nThen, r + t = "<<(r+t)<<endl;
    //Subtraction of complex number and complex number
    cout<<"\nThen, r - (4 + 2i) = "<<(r - complexnum(4,2))<<endl;
    //addition of complex number and complex number
    cout<<"\nThen, r + (2 + 2i) = "<<(r + complexnum(2,2))<<endl;
    system("pause");
}

```

**Output:**



- Keep in mind that for complex number manipulation you better use the standard C++ library, `<complex>`.
- Program example compiled using VC++/VC++ .Net.

```
//program deftmeth.cpp, default method
#include <iostream>
#include <cstring>
using namespace std;

//-----class declaration part-----
class def
{
    int    size;           //A simple stored value
    char   *string;        //A name for the stored data
public:
    //This overrides the default constructor
    def(void);

    //This overrides the default copy constructor
    def(def &in_object);

    //This overrides the default assignment operator
    def& operator=(def &in_object);

    //This destructor should be required with dynamic allocation
    ~def(void);

    //And finally, a couple of ordinary methods
    void set_data(int in_size, char *in_string);
    void get_data(char *out_string);
};

//-----class implementation-----
def::def(void)
{
    size = 0;
    string = new char[2];
    strcpy(string, "");
}

def::def(def &in_object)
{
    size = in_object.size;
    string = new char[strlen(in_object.string) + 1];
    strcpy(string, in_object.string);
}

def& def::operator=(def &in_object)
{
    delete [] string;
    size = in_object.size;
    string = new char[strlen(in_object.string) + 1];
    strcpy(string, in_object.string);
    return *this; //this pointer
}

def::~~def(void)
{
    delete [] string;
}

void def::set_data(int in_size, char *in_string)
```

```

{
    size = in_size;
    //delete the string size object...
    delete [] string;
    string = new char[strlen(in_string) + 1];
    strcpy(string, in_string);
}

void def::get_data(char *out_string)
{
    char temp[10];
    strcpy(out_string, string);
    strcat(out_string, " = ");
    itoa(size, temp, 10);
    strcat(out_string, temp);
}

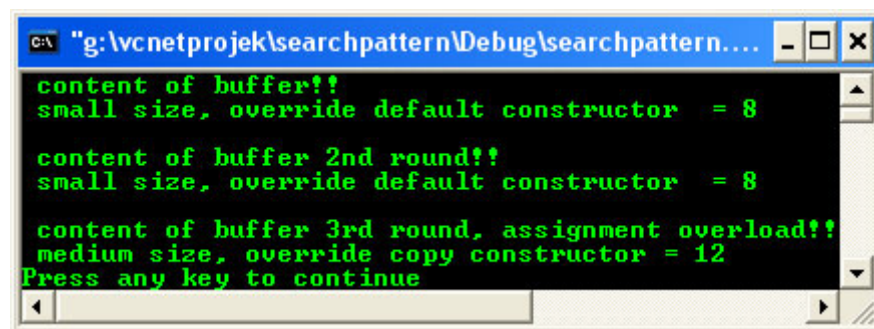
//-----main program-----
void main()
{
    char buffer[80];
    def my_data;
    my_data.set_data(8, " small size, override default constructor ");
    my_data.get_data(buffer);
    cout<<" content of buffer!!\n"<<buffer<<"\n";

    def more_data(my_data);
    more_data.set_data(12, " medium size, override copy constructor");
    my_data.get_data(buffer);
    cout<<"\n content of buffer 2nd round!!\n"<<buffer<<"\n";

    my_data = more_data;
    my_data.get_data(buffer);
    cout<<"\n content of buffer 3rd round, assignment overload!!\n"<<buffer<<"\n";
}

```

**Output:**



```

C:\ "g:\vcnetprojek\searchpattern\Debug\searchpattern...
content of buffer!!
small size, override default constructor = 8

content of buffer 2nd round!!
small size, override default constructor = 8

content of buffer 3rd round, assignment overload!!
medium size, override copy constructor = 12
Press any key to continue

```

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

```

/////--program objarray.cpp-/////
/////--FEDORA 3, g++ x.x.x-/////
/////--object and array-/////
#include <iostream>
using namespace std;

//-----class declaration-----
class wall
{
    int length;
    int width;
    static int extra_data;
    //declaration of the extra_data static type
public:
    wall(void);
    void set(int new_length, int new_width);
    int get_area(void);
    //inline function
    int get_extra(void) {return extra_data++;}
};

//-----class implementation-----
int wall::extra_data; //Definition of extra_data

```

```

//constructor, assigning initial values
wall::wall(void)
{
    length = 8;
    width = 8;
    extra_data = 1;
}

//This method will set a wall size to the two input parameters
void wall::set(int new_length, int new_width)
{
    length = new_length;
    width = new_width;
}

//This method will calculate and return the area of a wall instance
int wall::get_area(void)
{
    return (length * width);
}

//-----main program-----
int main()
{
    //instantiates 7 objects
    wall    small, medium, large, group[4];

    //assigning values
    small.set(5, 7);
    large.set(15, 20);

    //group[0] uses the default
    for(int index=1; index<4; index++)
        group[index].set(index + 10, 10);

    cout<<"Sending message-->small.get_area()\n";
    cout<<"Area of the small wall is "<<small.get_area()<<"\n\n";
    cout<<"Sending message-->medium.get_area()\n";
    cout<<"Area of the medium wall is "<<medium.get_area()<<"\n\n";
    cout<<"Sending message-->large.get_area()\n";
    cout<<"Area of the large wall is "<<large.get_area()<<"\n\n";

    cout<<"New length/width group[index].set(index + 10, 10)\n";
    for(int index=0; index<4; index++)
    {
        cout<<"Sending message using an array-->group"<<"["<<index<<"].get_area()\n";
        cout<<"An array of wall area "<<index<<" is "<<group[index].get_area()<<"\n\n";
    }

    cout<<"extra_data = 1, extra_data++\n";
    cout<<"Sending message using-->small.get_extra() or \n";
    cout<<"array, group[0].get_extra()\n";
    cout<<"Extra data value is "<<small.get_extra()<<"\n";
    cout<<"New Extra data value is "<<medium.get_extra()<<"\n";
    cout<<"New Extra data value is "<<large.get_extra()<<"\n";
    cout<<"New Extra data value is "<<group[0].get_extra()<<"\n";
    cout<<"New Extra data value is "<<group[3].get_extra()<<"\n";
    return 0;
}

```

```

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

```

```

Sending message-->small.get_area()
Area of the small wall is 35

```

```

Sending message-->medium.get_area()
Area of the medium wall is 64

```

```

Sending message-->large.get_area()
Area of the large wall is 300

```

```

New length/width group[index].set(index + 10, 10)
Sending message using an array-->group[0].get_area()
An array of wall area 0 is 64

```

```

Sending message using an array-->group[1].get_area()
An array of wall area 1 is 110

```

```
Sending message using an array-->group[2].get_area()  
An array of wall area 2 is 120  
  
Sending message using an array-->group[3].get_area()  
An array of wall area 3 is 130  
  
extra_data = 1, extra_data++  
Sending message using-->small.get_extra() or  
array, group[0].get_extra()  
Extra data value is 1  
New Extra data value is 2  
New Extra data value is 3  
New Extra data value is 4  
New Extra data value is 5
```

-----o0o-----

### Further reading and digging:

1. [Check the best selling C/C++ and object oriented books at Amazon.com.](#)
2. Any reference to Data Structures Using C.
3. Any reference to Data Structures Using C++. But may be it is better to use the container, iterator and algorithm of [Standard Template Library \(STL\)](#) (:o).