Chapter 8

Destructor & Operator Overloading

Destructor

- A **destructor** is a function that is called when an object is no longer required.
  - A **constructor** is a function which is called when a new object is created.
  - A constructor is usually used to initiate an object.
  - A destructor is usually used to destroy an object,
    - This is necessary, when some data members are dynamically allocated. (See Chapter 4)

Dynamic Memory Allocation

- Sometimes depending on the input data, you may allocate different amount of space for storing different types of variables at execution time

```
int n = 0;
cout << "Input the size of the vector - ";
cin >> n;
int vector[n];
```

error C2057: expected constant expression

Free Store (Heap)

- To hold a string entered by the user, there is no way you can know in advance how large this string could be.

- Free Store - When your program is executed, there is unused memory in your computer.
- You can dynamically allocate space within the free store for a new variable.

The new Operator

- Request memory for a double variable, and return the address of the space
  - `double* pvalue = NULL;`
  - `pvalue = new double;`
- Initialize a variable created by new
  - `pvalue = new double(9999.0);`
- Use this pointer to reference the variable (indirection operator)
  - `*pvalue = 1234.0;`

The delete Operator

- When you no longer need the (dynamically allocated) variable, you can free up the memory space.
  - `delete pvalue;`
    - Release memory pointed to by pvalue
  - `pvalue = 0;`
    - Reset the pointer to 0

- After you release the space, the memory can be used to store a different variable later.

Allocating Memory Dynamically for Arrays

- Allocate a string of twenty characters
  - `char* pstr;`
  - `pstr = new char[20];`
  - `delete [] pstr;`
    - Note the use of square brackets to indicate that you are deleting an array.
  - `pstr = 0;`
    - Set pointer to null

Dynamic Allocation of Multidimensional Arrays

- Allocate memory for a 3x4 array
  - `double (*pbeans)[4];`
  - `pbeans = new double [3][4];`
- Allocate memory for a 5x10x10 array
  - `double (*pBigArray)[10][10];`
  - `pBigArray = new double [5][10][10];`

- You always use only one pair of square brackets following the delete operator, regardless of the dimensionality of the array.
  - `delete [] pBigArray;`

The Default Destructor

- The destructor for a class is a member function with the same name as the class, preceded by a tilde (~).
  - For the `CBox` class, the prototype of the clas destructor is `~CBox();`
  - A destructor has no parameters.
- Ex8_01.cpp on P.400

```
~CBox()
{
  cout << “Destructor called.” << endl;
}
```

Class CMessage (1)

- Suppose you want to define a class
  - Each object contains a text string.
  - You don't want to declare a data member as a large character array (like `char [200]`),
  - So you'll allocate memory in the free store for the message when an object is created.
- This is your constructor:

```
CMessage(const char* text = “Default message”)
{
  pmessage = new char[strlen(text) + 1];
  strcpy(pmessage, text);
}
```

strlen, strcmp, strcpy

```
#include <iostream>
#include <cstring>
using std::cout;
using std::endl;

int main()
{
   char a[20] = "NCNU";
   char b[20] = "Sunday";

   cout << sizeof a << " " << strlen(a) << endl;
   // size = 20, string length = 4

   if (strcmp(a,b) < 0)
        cout << "The string " << a
               << " is less than " << b << endl;

   strcpy(a, b);
   cout << a << endl;
}
```

Destructors and Dynamic Memory Allocation

```
CMessage(const char* text = “Default message”)
{
  pmessage = new char[strlen(text) + 1];
  strcpy(pmessage, text);
}

~CMessage()
{
  cout << “Destructor called.” << endl;
  delete [] pmessage;
}
```

Ex8_02.cpp on P.403

- As the output indicates, the destructor is called only once.
  - The object motto is created automatically, so the compiler also calls the destructor automatically.
  - If you manually “delete pM”, it will free the memory pointed to by pM.
  - Because the pointer pM points to a CMessage object, this causes the destructor to be invoked.

Behavior of a Default Copy Constructor

```
CMessage motto1("Radiation fades your genes.");
CMessage motto2(motto1); // Calls default copy constructor
```

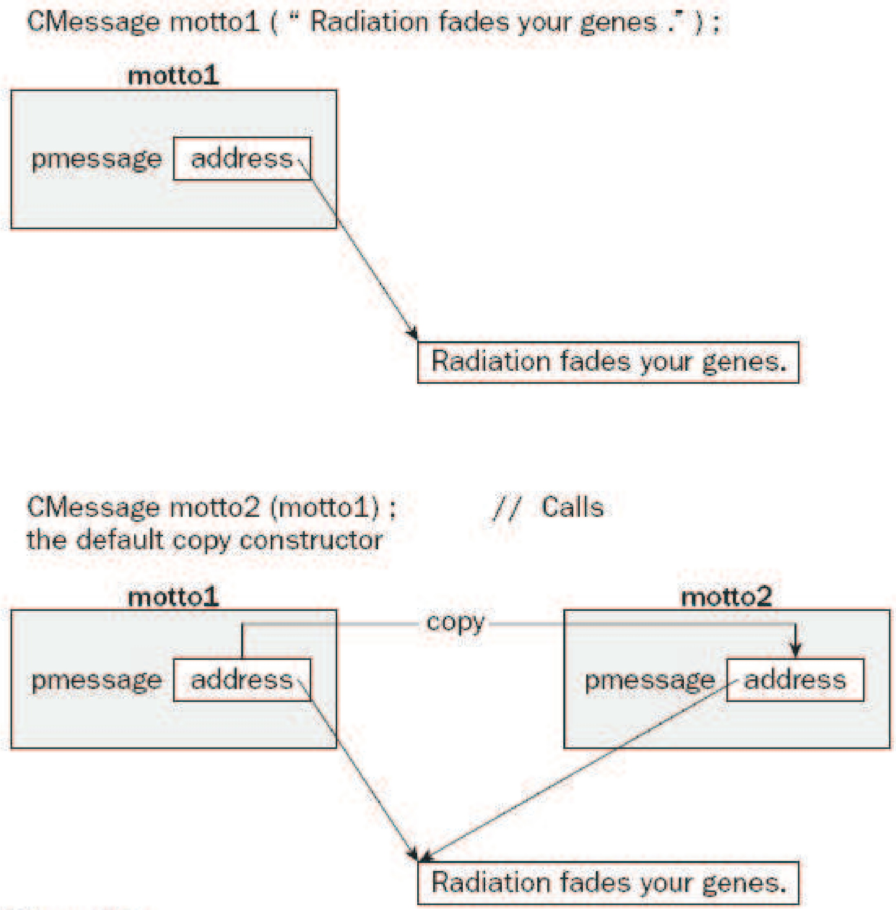

Figure 8-1

Implementing a Copy Constructor

- We don't want the two objects sharing the same string in the memory.
- If motto1 is destroyed, the pointer in motto2 will become invalid.
- Let us implement a copy constructor to generate an object which is identical but independent of the old one.

```
CMessage(const CMessage& initM)
{
    pmessage = new char [ strlen(initM.pmessage) +1 ];
    strcpy(pmessage, initM.pmessage);
}
```

Operator Overloading

- **Operator overloading** is a very important capability.
  - It allows you to make standard C++ operators, such as +, -, * and so on, work with objects of your own data types.
  - We want to write
    - if (box1 > box2)
  - instead of
    - if (IsGreaterThan(box1, box2))

- Let us recall some background of **function overloading** (Chapter 6).

Function Overloading

- Function overloading allows you to use the same function name for defining several functions as long as they each have different parameter lists.
- When the function is called, the compiler chooses the correct version according to the list of arguments you supply.
- The following functions share a common name, but have a different parameter list:
  - `int max(int array[], int len);`
  - `long max(long array[], int len);`
  - `double max(double array[], int len);`

Ex6_07.cpp on P.287

- Three overloaded functions of max()
- In main(), C compiler inspect the argument list to choose different version of functions.

Signature

- The signature of a function is determined by its name and its parameter list.
- All functions in a program must have unique signatures

- The following example is not valid overloading
  - `double max(long array[], int len);`
  - `long max(long array[], int len);`
- A different return type does not distinguish a function, if the signatures are the same.

Implementing an Overloaded Operator

```
class CBox
{
  public:
   bool operator> (CBox& aBox) const;
}
```

- The word `operator` here is a keyword.
- You declare the `operator>()` function as const because it doesn't modify any data members of the class. (P.362)

Using an Overloaded Operator

- if (box1 > box2)
  cout << “box1 is greater than box2”;
- if (box1.operator>(box2))

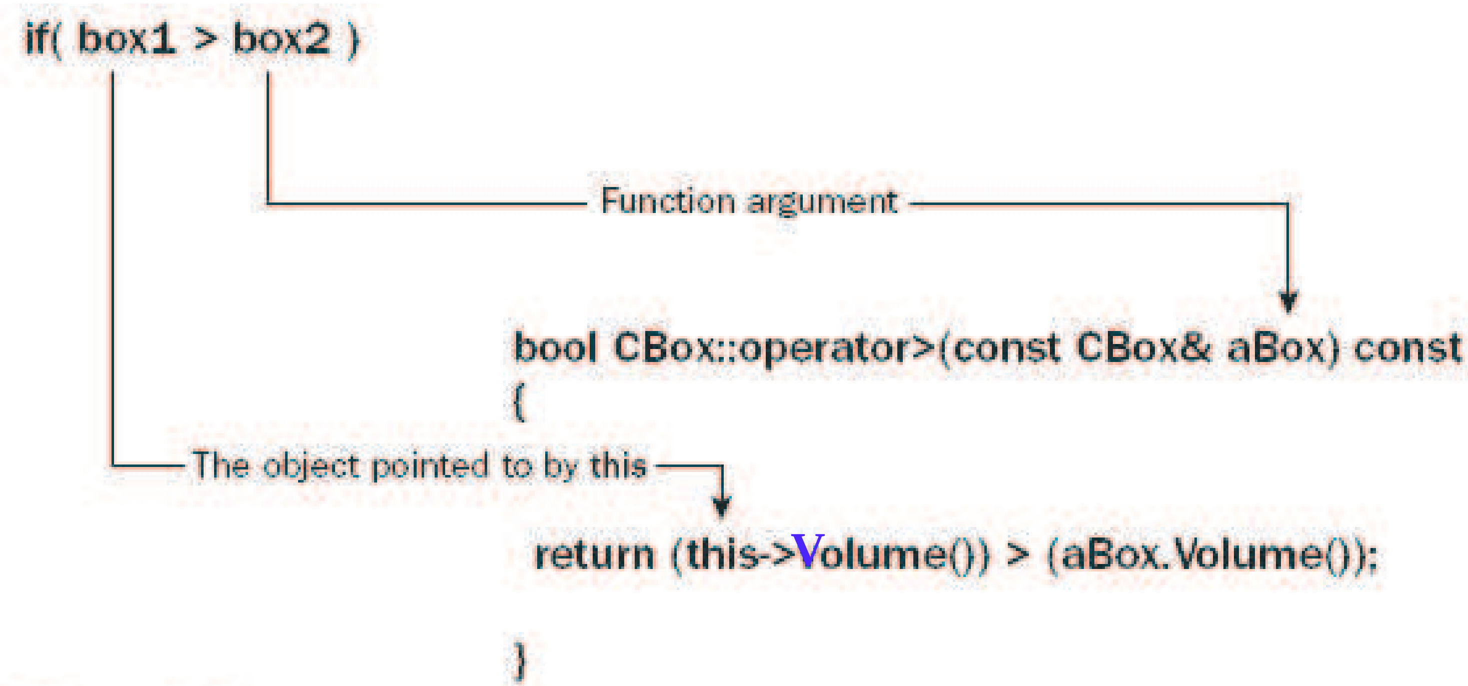

Figure 8-3

Ex8_03.cpp on P.412

```
bool CBox::operator> (CBox& aBox) const
      {
            return this->Volume() > aBox.Volume();
      }
```

- The left operand is defined implicitly by the pointer this.
- The basic > operator returns a value of type int
  - 1 for true
  - 0 for false.
- It will be automatically converted to bool.

Overloading the Assignment Operator

- What's wrong with the default assignment?
  - It simply provides a member-by-member copying process, similar to that of the default copy constructor.
  - They suffer from the same problem, when some data members are allocated dynamically.

Fixing the Problem

```
CMessage& operator= (const CMessage& aMess)
{
  // Release memory for 1st operand
  delete [] pmessage;
  pmessage = new char [ strlen(aMess.pmessage) + 1];

  // Copy 2nd operand string to 1st
  strcpy(this->pmessage, aMess.pmessage);

  // Return a reference to 1st operand
  return *this;
}
```

Why Do You Need to Return Something?

- Consider this statement
  - motto1 = motto2 = motto3;
- The assignment operator is right-associative, so it translates into
  - motto1 = (motto2.operator=(motto3));
  - motto1.operator=(motto2.operator=(motto3));

- You must at least return a CMessage object.

Why Do You Need to Return a Reference?

- Consider another example
  - (motto1 = motto2) = motto3;
- This translates into
  - (motto1.operator=(motto2))   = motto3;
- If the return type is merely CMessage instead of a reference, a temporary copy of the original object is returned.
  - Then you are assigning a value to a temporary object!
  - Make sure that your return type is CMessage&.

Check Addresses, If Equal

- The first thing that the operator function does is to delete the memory allocated to the first object, and reallocate sufficient memory to accommodate the new string.
- What happens to this statement?
  - motto1 = motto1
- Add this checking:

```
if (this == &aMess)
        return *this;
```