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 (P.194)

- 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.410

```
~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.413

- 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
```

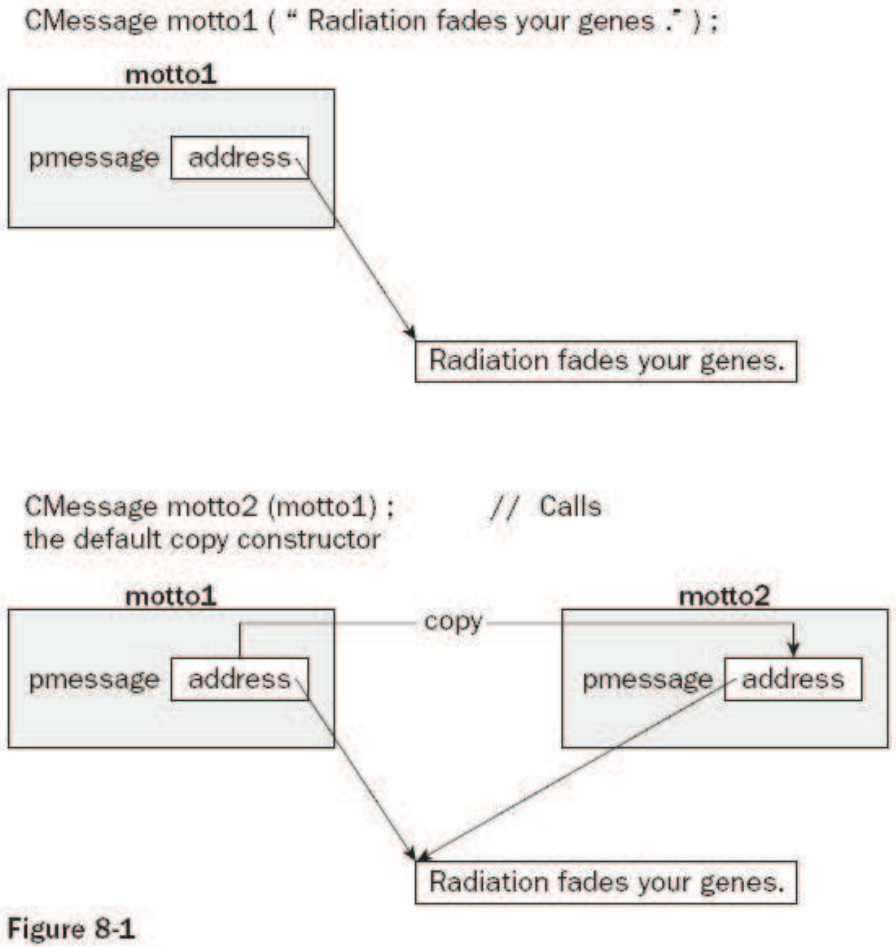

Figure 8-1

Q: What will the second `motto2.ShowIt()` display?

```
CMessage motto1("A stitch in time saves
  nine.");
CMessage motto2(motto1);
motto2.ShowIt();      // Display 2nd message
strcpy(motto1.pmessage, "Time and tide wait
  for no man.");
motto1.ShowIt();      // Display 1st message
motto2.ShowIt();      // Display 2nd message
```

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);
}
```

Exercise: Modify Ex8_02.cpp to implement this copy constructor.