

Copy Constructor, Destructor and Assignment Operator

In this Week

- Default Copy Constructor
- Overloaded Copy Constructor
- Deep Copy Vs Shallow Copy
- Destructor
- Default Assignment Operator
- Overloaded Assignment Operator

Default Copy Constructor

- Consider the **RationalNumber** class we have developed previously
- Now, observe the following C++ statements and determine its output:

```
RationalNumber r1(5, 8);    //Non default constructor
RationalNumber r2 = r1;    //Construct r2 from a copy of r1
cout << "Originally, r1 = " << r1 << " and r2 = " << r2 << endl;
r2.setNumerator(1);        //Modify r2
cout << "Now, r1 = " << r1 << " and r2 = " << r2 << endl;
```

- Now, consider the second statement closely
RationalNumber r2 = r1;
- First of all, we ask the question: Is this a valid statement? Answer **YES!**

Default Copy Constructor

- The statement **RationalNumber r2 = r1;** is valid because what is happening is that the new **RationalNumber** object **r2 is being constructed from a copy of r1**
- But we haven't provided a member function for copying an object!
- Indeed! But **C++ provides a default copy constructor**
- So what does the default copy constructor do?
- **The default copy constructor simply copies the VALUES of all the member variables of r1 to the corresponding member variables of r2**

Default Copy Constructor

- We may use the copy constructor in two ways
 - One way is as we showed
RationalNumber r2 = r1;
 - Another way is
RationalNumber r2(r1);
- In both cases, r2 is being constructed from a copy of r1
- Also we note that copy constructor is different from assignment operator (coming soon...)

Overloaded Copy Constructor

- Sometimes the default C++ copy constructor might not be enough to construct a new object from a copy of an existing objects
- In such cases, we need to provide our own copy constructor member function in order to construct objects correctly
- When we provide our own copy constructor, we call it **overloaded copy constructor** because it will overload the default C++ copy constructor
- In order to appreciate why we need our own overloaded copy constructor, let us define a new class that has a dynamically allocated memory member variable and see what the default copy constructor will do in that case
- For demonstration purposes, let us design a class named **SmartArray** to represent arrays
- Our **SmartArray** class will represent dynamic arrays but also with more features as we will soon explore
- **A SmartArray class has two member variables: A dynamic array and the size of the array**

Overloaded Copy Constructor

- The declaration of the **SmartArray** class is therefore:

```
#include <iostream>
using namespace std;

class SmartArray
{
private:
    int *A;
    int size;

public:
    //Constructors
    SmartArray();
    SmartArray(const int *A, const int &size);

    //Getters, Setters, operators and other functions
    int getSize() const;
    int& operator[](const int &index) const;
    void append(const int &e);
    friend ostream& operator << (ostream &outputStream, const SmartArray &L);
};
```

Overloaded Copy Constructor

- Let us first discuss the indexing operator [] overloading given by **int& operator [] (const int &index) const;**
- The indexing operator works with pointers (arrays)
- Given an array A, element at index k is given by the indexing operator A[k]
- In C++ indexing operator is interpreted as a binary operator whose left hand side operand is the pointer and right hand side operand is the index
- Therefore the way we look at this operator is as follows
A[k] is interpreted as **A [] k**
- When overloading this operator for our **SmartArray** class, the calling object (left hand side operand) will be a **SmartArray** object and the index will be an argument to the operator function (right hand side operand)
- Also remember that not only can we access element of an array A as A[k] but also we can assign the element a new value as A[k] = value
- This means this operator should return **not the literal R-value of the element at a given index**; but rather the memory location; which is why we are returning by reference. **Huh... but also it is a const function! How?**

Overloaded Copy Constructor

- The implementation of the member functions will therefore be as follows:

```
SmartArray::SmartArray()
{
    this->size = 0;
}

SmartArray::SmartArray(const int *A, const int &size)
{
    this->size = size;
    if (this->getSize() > 0)
    {
        this->A = new int[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = A[i];
    }
}

int SmartArray::getSize() const
{
    return this->size;
}

int& SmartArray::operator[](const int &index) const
{
    if (index >= 0 && index < this->getSize())
        return this->A[index];
    else
    {
        cout << "ERROR! Index out of bounds." << endl;
        abort();
    }
}
```


Overloaded Copy Constructor

```
void SmartArray::append(const int &e)
{
    //First create a temporary array whose size is this->size+1
    int *temp = new int [this->getSize() + 1];

    //Copy the elements of this->A to temp
    for (int i = 0; i < this->getSize(); i++)
        temp[i] = this->A[i];

    //Copy the element to be appended to temp
    temp[this->getSize()] = e;

    //Delete the existing array this->A
    if (this->getSize() > 0)
        delete[] this->A;

    //Make the array this->A to point to temp and increment the size
    this->A = temp;
    this->size += 1;
}

ostream& operator << (ostream &outputStream, const SmartArray &L)
{
    outputStream << "[";
    for (int i = 0; i < L.getSize() - 1; i++)
        outputStream << L[i] << ", ";
    if (L.getSize() > 0)
        outputStream << L[L.getSize() - 1];
    outputStream << "]" ;
    return outputStream;
}
```

Overloaded Copy Constructor

```
int main()
{
    //Test Default Constructor
    SmartArray L1;
    cout << "Default SmartArray: " << L1 << endl;

    //Test Non-Default Constructor
    int *x = new int[3];
    x[0] = 7; x[1] = 2; x[2] = 5;
    SmartArray L2(x, 3);
    cout << "Non-Default SmartArray: " << L2 << endl;

    //Test getSize and operator []
    cout << "Printing the SmartArray L2 elements: ";
    for (int i = 0; i < L2.getSize(); i++)
        cout << L2[i] << " ";
    cout << endl;

    //Test Index out of bound case
    //cout << "The element of L1 at index 0 is " << L1[0] << endl;

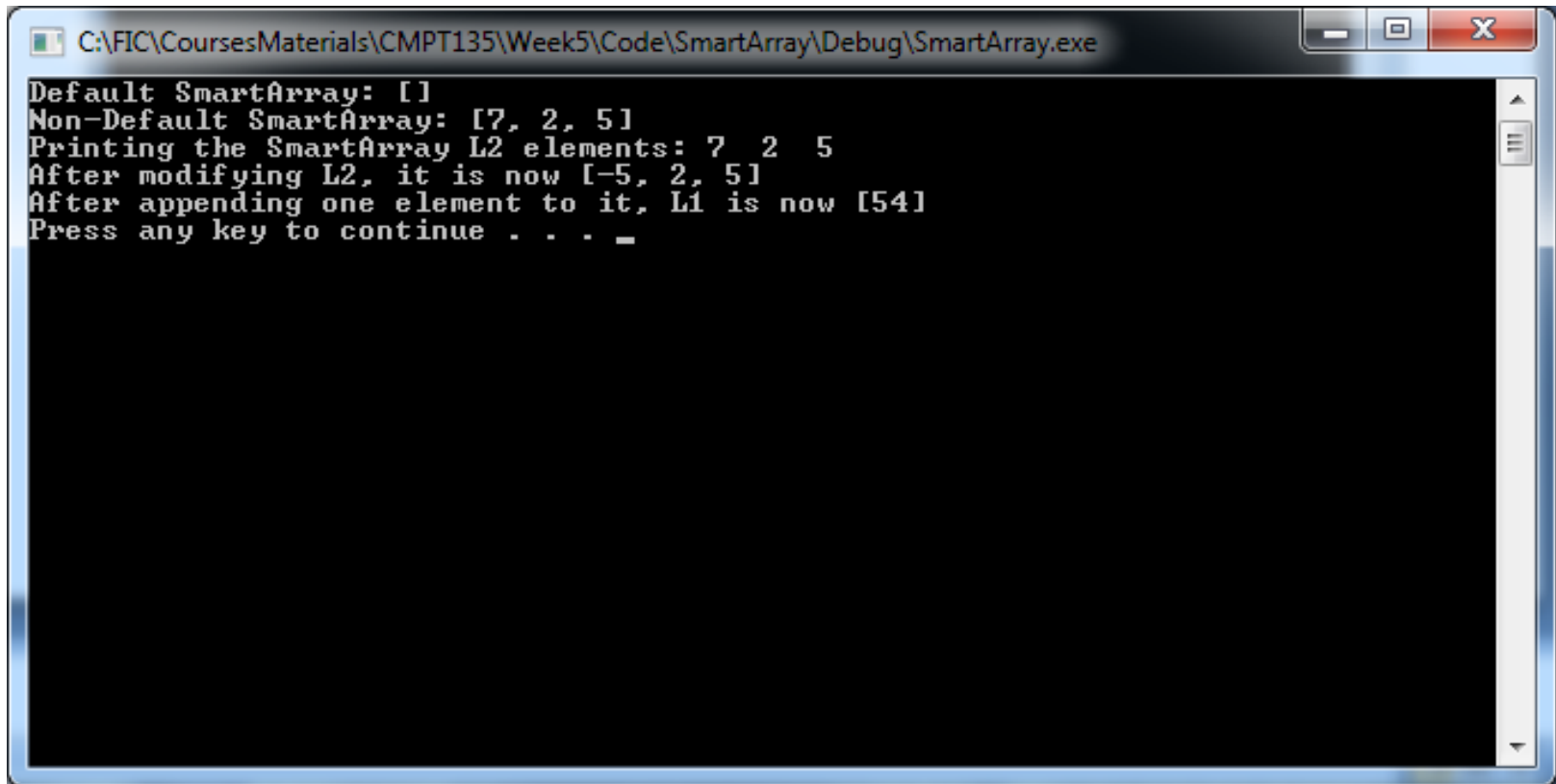
    //Test modification of element with operator []
    L2[0] = -5;
    cout << "After modifying L2, it is now " << L2 << endl;

    //Test Append member function
    L1.append(54);
    cout << "After appending one element to it, L1 is now " << L1 << endl;

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

Overloaded Copy Constructor

- The output of this main program will then be



```
C:\FIC\CoursesMaterials\CMPT135\Week5\Code\SmartArray\Debug\SmartArray.exe
Default SmartArray: []
Non-Default SmartArray: [7, 2, 5]
Printing the SmartArray L2 elements: 7 2 5
After modifying L2, it is now [-5, 2, 5]
After appending one element to it, L1 is now [54]
Press any key to continue . . . _
```

Overloaded Copy Constructor

- Now, let us use the default copy constructor in order to copy one **SmartArray** object to another and see what happens

```
//Make a default copy of one SmartArray object to another
SmartArray L3 = L2;
cout << "L3 which is a copy of L2 is " << L3 << endl;

//Modify an element of L3
L3[1] = 33;
cout << "After modifying an element of L3, now L3 is " << L3 << endl;
cout << "Interestingly L2 is also modified and is " << L2 << endl;

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

- What is the output of this code?**

Overloaded Copy Constructor

- Surprisingly, or not surprisingly, the output will be as follows

```
Default SmartArray: []  
Non-Default SmartArray: [7, 2, 5]  
Printing the SmartArray L2 elements: 7 2 5  
After modifying L2, it is now [-5, 2, 5]  
After appending one element to it, L1 is now [54]  
L3 which is a copy of L2 is [-5, 2, 5]  
After modifying an element of L3, now L3 is [-5, 33, 5]  
Interestingly L2 is also modified and is [-5, 33, 5]  
Press any key to continue . . . _
```

- Why? Because the default copy constructor copied the values of the member variables of L2 into the member variables of L3
- This means the value of the size of L2 was copied to the size of L3 and the value of the pointer variable A of L2 (which is a memory address) was copied to the pointer variable of L3.
- Therefore both L2 and L3 will point to the same consecutive memory on the heap and modifying element of L3 will also modify L2; and vice versa!!!

Overloaded Copy Constructor

- We conclude that the **default C++ copy constructor** does **Shallow Copy**
- That is, it does not know much about our class and therefore copies only at a high level
- It does not go deep to copy allocated memory
- This is very **dangerous**. For example, if we delete the allocated memory of L2, then automatically the memory of L3 is also deleted. Accessing L3 then will crash the program
- For this reason, it is important **we provide our own copy constructor that copies not the pointer value but rather creates a new dynamic memory and copies the elements**

Overloaded Copy Constructor

- The overloaded copy constructor of our own will look like

- **Declaration**

SmartArray(const SmartArray &L);

- **Definition**

```
SmartArray::SmartArray(const SmartArray &L) //Copy constructor
{
    this->size = L.getSize();
    if (this->getSize() > 0)
    {
        this->A = new int[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = L[i];
    }
}
```

Overloaded Copy Constructor

- Now, we provide the same main program and its output to see the impact of our copy constructor
- We say, our own copy constructor does a DEEP copy and avoids the entanglement of L2 and L3

```
Default SmartArray: []
Non-Default SmartArray: [7, 2, 5]
Printing the SmartArray L2 elements: 7  2  5
After modifying L2, it is now [-5, 2, 5]
After appending one element to it, L1 is now [54]
L3 which is a copy of L2 is [-5, 2, 5]
After modifying an element of L3, now L3 is [-5, 33, 5]
Interestingly L2 is unchanged and is [-5, 2, 5]
Press any key to continue . . . _
```


Destructor Member Function

- Recall that whenever we create a dynamic array, we need to delete it in order to free the memory we had allocated
- This means, in the previous example, when the program terminates the memories allocated to L1, L2 and L3 are still not freed
- **Question:** How do we free dynamically allocated memory inside objects?
- **Proposal:** Can we delete the allocated memories of L1, L2 and L3 from within the main program? **No!** Because the pointer member variable of the SmartArray class is a private member variable and thus can not be accessed from outside the class
- **Answer:** We use a special member function called **destructor!**

Destructor Member Function

- The destructor member function looks like:

- **Destructor: Declaration**

```
~SmartArray();
```

- **Destructor: Definition**

```
SmartArray::~~ SmartArray()  
{  
    if (this->getSize() > 0)  
    {  
        delete[] this->A;  
        this->size= 0;  
    }  
}
```

- **Remark:** Check if memory was allocated before deleting

Destructor Member Function

- The destructor is important because it is automatically called every time an object goes out of scope and therefore its memory automatically deleted
- While objects with destructor member function will automatically be deleted when they go out of scope, sometimes we may need to delete objects before they go out of scope
- For example, in order to delete object L1, we proceed as follows:

L1.~SmartArray();

- The main program demonstrating this follows...

Deleting Objects

```
//Make a deep copy of one SmartArray object to another
SmartArray L3 = L2;
cout << "L3 which is a copy of L2 is " << L3 << endl;

//Modify an element of L3
L3[1] = 33;
cout << "After modifying an element of L3, now L3 is " << L3 << endl;
cout << "Interestingly L2 is unchanged and is " << L2 << endl;

//Let us delete L2
L2.~SmartArray();
cout << "After deleting it, L2 is " << L2 << endl;
cout << "Interestingly L3 is still " << L3 << endl;

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

```
Default SmartArray: []
Non-Default SmartArray: [7, 2, 5]
Printing the SmartArray L2 elements with indexing operator: 7 2 5
After modifying L2, it is now [-5, 2, 5]
After appending one element to it, L1 is now [54]
L3 which is a copy of L2 is [-5, 2, 5]
After modifying an element of L3, now L3 is [-5, 33, 5]
Interestingly L2 is unchanged and is [-5, 2, 5]
After deleting it, L2 is []
Interestingly L3 is still [-5, 33, 5]
Press any key to continue . . . _
```

Deleting Objects

- In order to appreciate the overloaded copy constructor, now do the following and explain what happened:
 - Comment out your copy constructor declaration
 - Comment out your copy constructor definition
 - Run the same main program you have in the previous page
 - Not only will your program print garbage data for L3 but also your program will **crash after the program ends!!!** Explain why!
- Afterwards uncomment the copy constructor so that to have a correct class to work with in the following sections

The Default Assignment Operator

- Now, consider the statements

```
SmartArray L4; //Default constructor  
L4 = L3; //This is NOT copy constructor!
```

- What does the last statement **L4 = L3;** do?
- It does **NOT** call copy constructor! Because copy constructor is called only when object is **constructed by copying from another**; that is **copy constructor is called only when there is declaration of a new object and initialized** to another object as in like

```
SmartArray L3 = L2; //Copy Constructor
```

The Default Assignment Operator

- So what does the statement **L4 = L3;** do?
- **Answer:** It calls the **DEFAULT C++ ASSIGNMENT OPERATOR** to assign L3 to L4
- The default assignment operator copies the **VALUES** of all member variables of L3 onto L4
- This means pointers will be copied but not their allocated memory

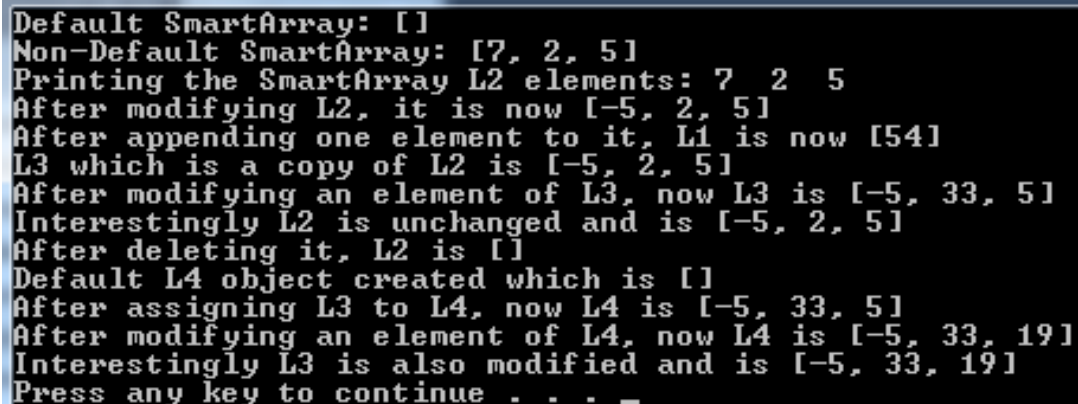
The Default Assignment Operator

- Consider the following program and analyze its output

```
//Make a default assignment of one SmartArray object to another
SmartArray L4;
cout << "Default L4 object created which is " << L4 << endl;
L4 = L3;
cout << "After assigning L3 to L4, now L4 is " << L4 << endl;

//Modify an element of L4
L4[2] = 19;
cout << "After modifying an element of L4, now L4 is " << L4 << endl;
cout << "Interestingly L3 is also modified and is " << L3 << endl;

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

A screenshot of a terminal window showing the output of the C++ program. The output is as follows:

```
Default SmartArray: []
Non-Default SmartArray: [7, 2, 5]
Printing the SmartArray L2 elements: 7 2 5
After modifying L2, it is now [-5, 2, 5]
After appending one element to it, L1 is now [54]
L3 which is a copy of L2 is [-5, 2, 5]
After modifying an element of L3, now L3 is [-5, 33, 5]
Interestingly L2 is unchanged and is [-5, 2, 5]
After deleting it, L2 is []
Default L4 object created which is []
After assigning L3 to L4, now L4 is [-5, 33, 5]
After modifying an element of L4, now L4 is [-5, 33, 19]
Interestingly L3 is also modified and is [-5, 33, 19]
Press any key to continue . . . _
```


The Default Assignment Operator

- Again, both L4 and L3 will have pointers pointing to the same allocated memory on the heap
- Therefore changing any element in L4 changes the element in L3 and vice versa
- But **worse still**, when L3 or L4 goes out of scope or is manually deleted, then its allocated memory to which both L3 and L4 point to will be deleted
- This means the remaining object will have its memory deleted too! **Dangerous!!!**

Overloaded Assignment Operator

- In order to avoid any memory entanglement during assignment operation, we therefore need to overload the default assignment operator whenever our class has a pointer member variable
- Also recall that chain assignment is allowed in C++ as follows:

```
int x, y, z;
```

```
z = x = y = 3;
```

- This assigns the value 3 to the variable y. Next it assigns the returned value of the assignment operation `y = 3` to the variable x
- This means the assignment operator returns a value!
- So what does it return? It returns the assigned value (or object) by reference! **Why by reference? This is mainly for speed reasons**
- In the next section, we write our own assignment operator to overload the default C++ operator

Overloaded Assignment Operator

- The overloaded assignment operator will have the declaration given by

SmartArray& operator = (const SmartArray& L);

- The overloaded assignment operator is a binary operator and has a function name given by **operator =**
- The calling object of the operator is the left hand side operand of the assignment operator
- It takes one argument which is the right hand side operand of the assignment operator
- It returns a **reference** of the class type

Overloaded Assignment Operator

- The implementation of the operator member function will then look like:

```
SmartArray& SmartArray::operator = (const SmartArray &L)
{
    //Delete the left hand side object's memory
    this->~SmartArray();
    //Now copy the right hand side to the left
    this->size = L.getSize();
    if (this->getSize() > 0)
    {
        this->A = new int[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = L[i];
    }
    return *this;
}
```

Overloaded Assignment Operator

- Now, we can see the same main program and its output

```
//Make an assignment of one SmartArray object to another
SmartArray L4;
cout << "Default L4 object created which is " << L4 << endl;
L4 = L3;
cout << "After assigning L3 to L4, now L4 is " << L4 << endl;

//Modify an element of L4
L4[2] = 19;
cout << "After modifying an element of L4, now L4 is " << L4 << endl;
cout << "Interestingly L3 is unchanged and is " << L3 << endl;

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

```
Default SmartArray: []
Non-Default SmartArray: [7, 2, 5]
Printing the SmartArray L2 elements: 7 2 5
After modifying L2, it is now [-5, 2, 5]
After appending one element to it, L1 is now [54]
L3 which is a copy of L2 is [-5, 2, 5]
After modifying an element of L3, now L3 is [-5, 33, 5]
Interestingly L2 is unchanged and is [-5, 2, 5]
After deleting it, L2 is []
Default L4 object created which is []
After assigning L3 to L4, now L4 is [-5, 33, 5]
After modifying an element of L4, now L4 is [-5, 33, 19]
Interestingly L3 is unchanged and is [-5, 33, 5]
Press any key to continue . . .
```

Overloaded Assignment Operator

- This looks good so far. But it has a very fundamental problem
- Consider the following assignment operation

```
//Assign L4 to L4 and see what happens  
L4 = L4;  
cout << "After assigning L4 to L4, now L4 is " << L4 << endl;
```

- What will be the output of this code?
- You might think it will give the following output
After assigning L4 to L4, now L4 is [-5, 33, 19]
- But it does **NOT** give such output. Rather it gives the following output

After assigning L4 to L4, now L4 is []

- **Why?**

Overloaded Assignment Operator

- Why does it give empty array?
- To see why, we need to look at our overloaded assignment operator
- Observe that the first statement in our overloaded assignment operator function is

this->~SmartArray();

- When the **same object is on both sides of the assignment operator**, this will delete the data and setting the size to zero.
- After that we will not create any new memory for the data because the size is zero. Neither will we execute the for loop block for the same reason
- This means the data pointer will completely lose the original data
- So practically we have deleted the elements and got empty array.

Overloaded Assignment Operator

- What should we do then to avoid this problem?
- **Answer:-** We must first check for self assignment; that is when an object is assigned to itself
- How do we do that?
- **We check if the memory address of the left hand side operand is the same as the memory address of the right hand side operand**
- Alternatively, we could compare if the values of the member variables (that is, the **size** and the **elements of the data array**) of the right hand side operand are equal to the values on the left side

Overloaded Assignment Operator

- Since the memory address check is more efficient, we will use that approach to do self assignment check as shown below

```
SmartArray& SmartArray :: operator = (const SmartArray &L)
{
    //Check for self assignment. If so, do nothing.
    if (this == &L)
        return *this;
    //Delete the left hand side object's memory
    this->~SmartArray();
    //Now copy the right hand side to the left
    this->size = L.getSize();
    if (this->getSize() > 0)
    {
        this->A = new int[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = L[i];
    }
    return *this;
}
```

Overloaded Assignment Operator

- The following program demonstrates self assignment and chain assignment

```
//Make an assignment of one SmartArray object to another
SmartArray L4;
cout << "Default L4 object created which is " << L4 << endl;
L4 = L3;
cout << "After assigning L3 to L4, now L4 is " << L4 << endl;

//Modify an element of L4
L4[2] = 19;
cout << "After modifying an element of L4, now L4 is " << L4 << endl;
cout << "Interestingly L3 is unchanged and is " << L3 << endl;

L4 = L4;
cout << "After assigning L4 to L4, now L4 is " << L4 << endl;

SmartArray L5, L6, L7;
L6 = L5 = L7 = L4;
cout << "L5 is " << L5 << endl;
cout << "L6 is " << L6 << endl;
cout << "L7 is " << L7 << endl;

system("Pause");
return 0;
```

}

Overloaded Assignment Operator

- The output of the program will be

```
Default SmartArray: []
Non-Default SmartArray: [7, 2, 5]
Printing the SmartArray L2 elements with indexing operator: 7 2 5
After modifying L2, it is now [-5, 2, 5]
After appending one element to it, L1 is now [54]
L3 which is a copy of L2 is [-5, 2, 5]
After modifying an element of L3, now L3 is [-5, 33, 5]
Interestingly L2 is unchanged and is [-5, 2, 5]
After deleting it, L2 is []
Interestingly L3 is still [-5, 33, 5]
Default L4 object created which is []
After assigning L3 to L4, now L4 is [-5, 33, 5]
After modifying an element of L4, now L4 is [-5, 33, 19]
Interestingly L3 is unchanged and is [-5, 33, 5]
After assigning L4 to L4, now L4 is [-5, 33, 19]
L5 is [-5, 33, 19]
L6 is [-5, 33, 19]
L7 is [-5, 33, 19]
Press any key to continue . . . _
```

- Note that we can **NOT** have a statement like

SmartArray L5 = L6 = L7 = L4;

because the chain operation is possible only on existing objects. That is, our overloaded function that allows chain operations is assignment operator (not the copy constructor)

Concluding Remarks

- In C++, given a class (example the **SmartArray** class), the statement
SmartArray L2 = L1; //Assuming L1 was already declared, OR
SmartArray L2(L1); //Assuming L1 was already declared

calls the **copy constructor**; while the statement

L2 = L1; //assuming L1 and L2 were already declared

call the **assignment operator**

- In order to avoid shallow copy and perform a deep copy during copy constructor or assignment operation, we need to provide our own copy constructor and assignment operator member functions that overload the default C++ default copy constructor and assignment operator
- While a class can have several constructors, it has only one destructor
- While it is good for overloaded assignment operator to return the assigned object so that to allow chain assignment; it is not a MUST for the overloaded assignment operator to return anything. It can return by value, by reference or be void too.