OOP
Lecture 14

30/11/2021

Class Activity :

Write a class “List” which consists of a dynamic array with flexible size. Write the following member functions,

- Parameterized Constructor with parameter size (default value 10).
- addElement function to add an element at the end of the array
- insertElement function to insert an element at a specified index
- removeElement function to remove an element from the array
- Overload ostream << operator to print array with cout

Discussion :

First of all, we need to analyze what the requirement is? In this question we need to create a dynamic array with flexible size. Flexible size means the size of the array should be increased or decreased automatically when we add or remove elements from the array. So, in such a case we can use a technique where we define a large dynamic array and take a variable named “currentIndex” to show the number of elements in the array and a variable “size” which holds the total size of the array.

We can implement our logic that we can increase or decrease the current size of the array one by one as new elements are added/inserted or deleted, and when the total size of the array is reached, then we need to increase the total size of the array. In such logic, our currentSize is known as the logical size of the array and the total size is known as the physical size of the array.

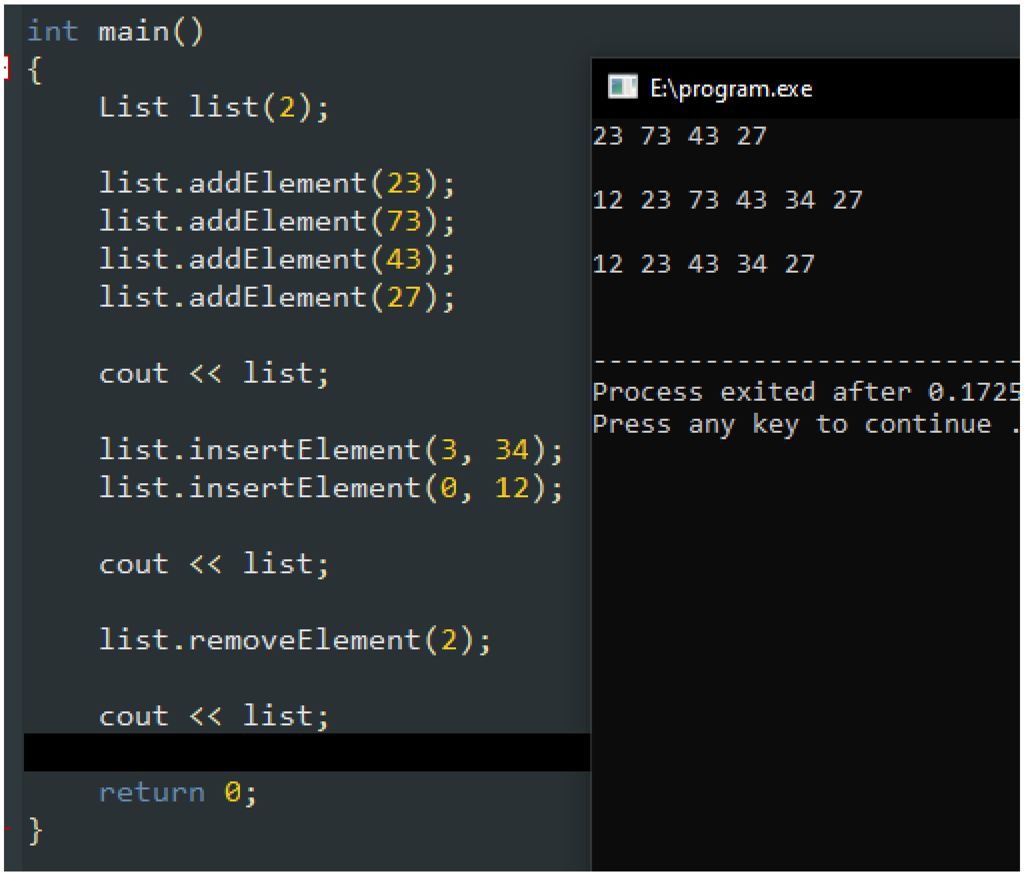
It is important to note that we cannot change the physical/actual size of the array once it is declared and defined. The only way to change the size of the array is to declare another array with a new size and copy our previous array into the new array.

For best practice, try to implement the above logic before moving further.

Solution Code :

```
class List
{
        int size;
        int *array;
        int currentSize;

// Private function to increase the size of the array if logical size
(currentSize) equals to the physical size ( total size )
        void resize(int newSize)
        {
                int i;
                size = newSize;

        // New array with greater size
                int *newArray = new int[size];

        // Copying the elements of array into new array
        // Running loop till currentSize because it shows
        // the number of elements in the array
                for ( i = 0; i < currentSize; i++ )
                        newArray[i] = array[i];

        // Deleting our Previous Array
                delete[] array;

        // Benefit of using the dynamic array is that we only
        // need to assign the address to the pointer and it starts
        // to point the memory
                array = newArray;
        }
        Public:
// Parameterized Constructor with size
         List(int size)
         {
                  if ( size <= 0 )
                           size = 10; //Default value is 10

                  this->size = size;

                  array = new int[size];
                  currentSize = 0;
                }

// Function to add a new element at the end of the array
// Here, we applied a check if ( currentSize == size ), it means
// that we have to increase the size of the array when the
// current size(number of elements) reaches the actual size of the
// array
        void addElement( const int VALUE )
        {
                if ( currentSize == size )
                        resize(size + 10);

                array[currentSize] = VALUE;
                currentSize++;
        }

// For now we are assuming that the INDEX < currentSize
// otherwise we need to apply check for it as well
// Same logic to check if max size of array has reached
        void insertElement( const int INDEX, const int VALUE )
        {
                if ( currentSize == size )
                        resize(size + 10);
// Shifting the array towards right
                for ( int i = currentSize; i >= INDEX; i-- )
                        array[i] = array[i-1];

                array[INDEX] = VALUE;
                currentSize++;
        }

// Shifting array towards left till the position which element
// is to be removed.
        void removeElement( const int INDEX )
        {
                for ( int i = INDEX; i < currentSize - 1; i++ )
                        array[i] = array[i+1];

                currentSize--;
        }

// overloading stream << operator for output
        friend ostream& operator << (ostream &out, List &l)
        {
                int i;

                for ( i = 0; i < l.currentSize; i++ )
                        out << l.array[i] << ' ';
                out << "\n\n";

                return out;
        }
// Destructor to delete array
        ~List()
        {       delete[] array;      }
};
```

Output with main() function :



```
int main()
{
    List list(2);

    list.addElement(23);
    list.addElement(73);
    list.addElement(43);
    list.addElement(27);

    cout << list;

    list.insertElement(3, 34);
    list.insertElement(0, 12);

    cout << list;

    list.removeElement(2);

    cout << list;

    return 0;
}
```

E:\program.exe

```
23 73 43 27

12 23 73 43 34 27

12 23 43 34 27


--------------------------
Process exited after 0.1725
Press any key to continue .
```

Using list[i] :

Now we need to use the object of the List class like this list[i], which ultimately uses the array defined in the class. To do this we need to overload the brackets [ ] operator in the class. The syntax will be the same.

```
int operator [ ] (const int INDEX)
{
        return x[INDEX];
}
```

int is the return type here which shows that this function is returning integer value. But, we can see that we can only use this function for read-only purposes, i.e it cannot update the values at the indexes like if we try to do list[2] = 50; then it will give an error because list[2] will return an integer and integer = integer gives error. It is only valid for the read-only purpose like cout << list[2]; it will print the value of array at index 2.

If we want to use it for write purposes as well then, we need to return the reference to the integer value instead of the original integer value. What?? You forgot what is reference in programming!! 😕

Looks like someone is not paying attention to the lectures… 🧐

Anyhow, in lecture 9 and 10, we discussed reference to value and object in detail. Refer to it for further understanding.

```
int& operator [] (const int INDEX)
{
        return x[INDEX];
}
```

With this code, we can use it for both reading and writing purposes; it means that list[2] = 50; (initialization) and cout << list[2]; (retrieving data) both are valid in this scenario. But but but, there's a problem here. When we are initializing values directly like this list[2] = 50; in this case we are not incrementing the currentSize of the array, it will still be zero. Remember the logical and physical size of the array. We always deal with the logical size which shows the number of elements in the array. It is still zero. So, what's the solution to this problem? No no no… writing **currentSize++;** in the function is not the solution, Because in that case if we wrote two statements like :

```
list[0] = 50;
list[0] = 10;
```

In this case, we are overwriting the same value and the logical size of the array should be 1. But the operator function is called 2 times and if we had written the currentSize++; in the function, it would be 2 so far.

The real solution to this problem is another detail, but that is the story for another lecture. Till now, we will use this operator for reading purposes only.

Here, our class with a flexible size of dynamic array ends. Just a tip for everyone who’s reading this, gather the idea of the vectors in C++, it will surely help you to face the aggregation of the next class. 🙃

Now we are going to discuss some basic concepts of the classes, grasp the complete idea about them.

Consider the following code:

```
class Special
{
  public:
       Special()
       {
              cout << "Constructor Called\n";
       }

       Special(const Special &s)
       {
              cout << "Copy Constructor Called\n";
       }
};
int main()
{
       Special s1, s2;
       Special s3 = s2;
       return 0;
}
```

What should be the output of the following program?

```
---------------------------------------------
|       Constructor Called              |
|       Constructor Called              |
|       Copy Constructor Called         |
---------------------------------------------
```

Hope, you got this right!!

Consider the next code and guess the output:

```
 class Special
               {
                 public:
                      Special()
                      {
                              cout << "Constructor Called\n";
                      }

                      Special(const Special &s)
                      {
                              cout << "Copy Constructor Called\n";
                      }

                      void operator + (Special s)
                      {
                              cout << "+ Operator Called\n";
                      }

               };
```

```
int main()
{
        Special s1, s2;
        Special s3 = s2;
        s1 + s2;
        return 0;
}
```

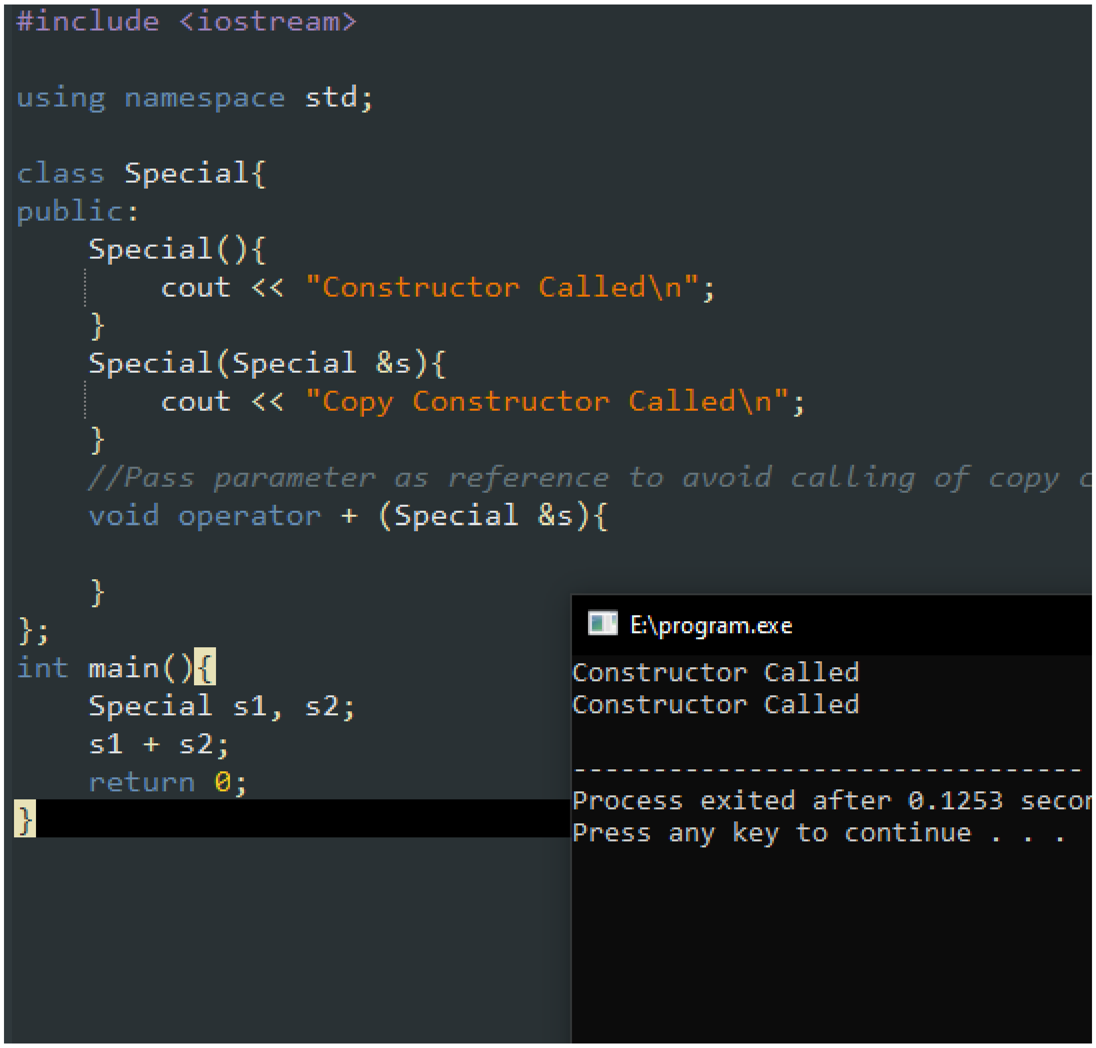
Output :

```
---------------------------------------------
|       Constructor Called              |
|       Constructor Called              |
|       Copy Constructor Called         |
|       Copy Constructor Called         |
|       + Operator Called               |
---------------------------------------------
```

Why does the copy constructor called 2 times?
Because we didn't use the & sign in the operator overloading (+ operator). So, when we passed s2 as a parameter to the + function, instead of making its reference, it created a whole new copy of the object s2 with a new name s. To make this copy the copy constructor is called.

Quick Quiz : If we change the return type of the + operator function to Special instead of void and return *this; what should be the output of the code. And also try to write Special& and check the output. Surely, it will give an idea about when and where, which function is called.

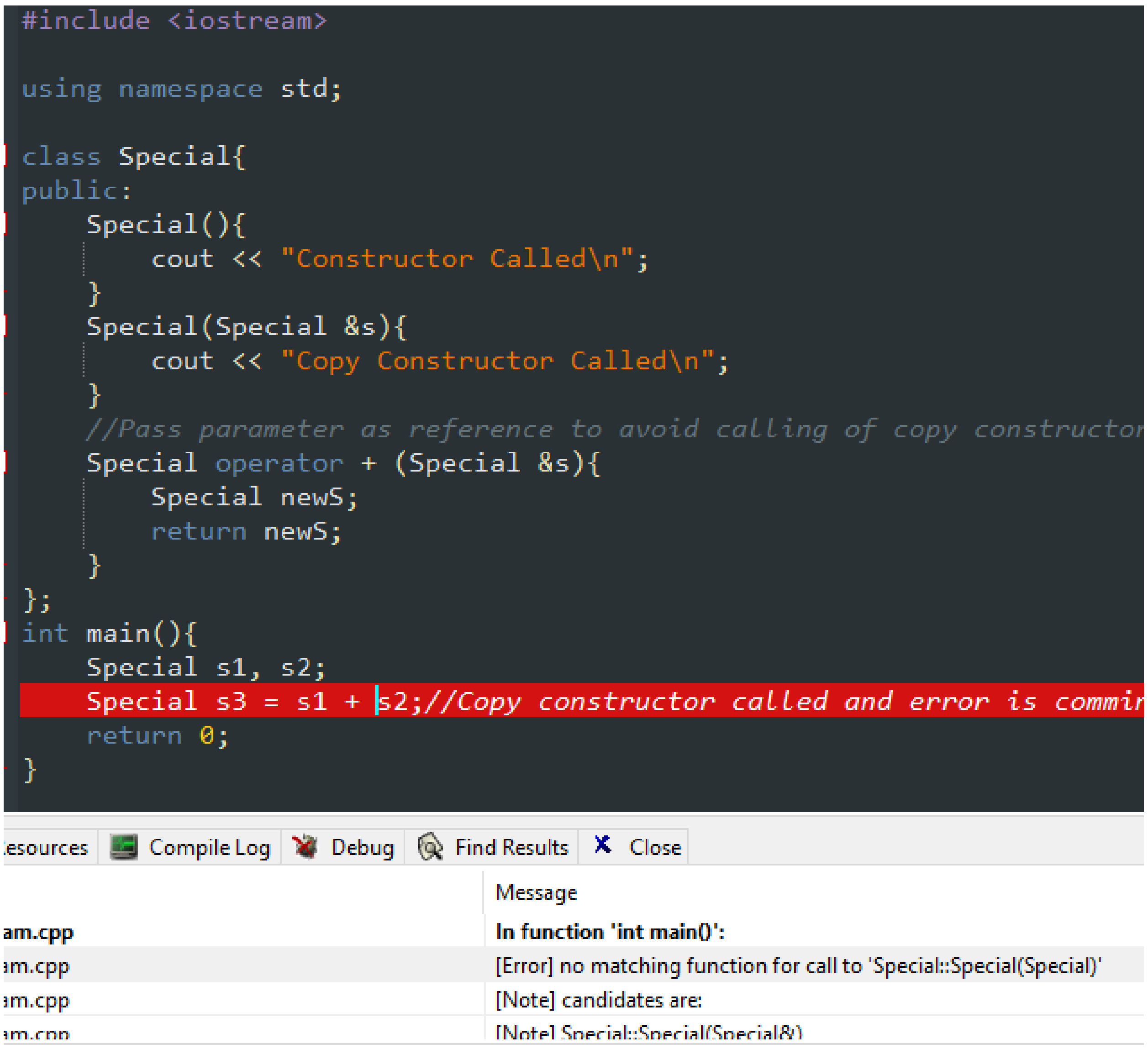
Some Samples are given here,



```
#include <iostream>

using namespace std;

class Special{
public:
    Special(){
        cout << "Constructor Called\n";
    }
    Special(Special &s){
        cout << "Copy Constructor Called\n";
    }
    //Pass parameter as reference to avoid calling of copy c
    void operator + (Special &s){

    }
};
int main(){
    Special s1, s2;
    s1 + s2;
    return 0;
}
```

E:\program.exe

```
Constructor Called
Constructor Called


--------------------------------
Process exited after 0.1253 secon
Press any key to continue . . .
```



```
#include <iostream>

using namespace std;

class Special{
public:
    Special(){
        cout << "Constructor Called\n";
    }
    Special(Special &s){
        cout << "Copy Constructor Called\n";
    }
    //Pass parameter as reference to avoid calling of copy constructor
    Special operator + (Special &s){
        Special newS;
        return newS;
    }
};
int main(){
    Special s1, s2;
    Special s3 = s1 + s2;//Copy constructor called and error is commin
    return 0;
}
```

```
am.cpp    In function 'int main()':
am.cpp    [Error] no matching function for call to 'Special::Special(Special)'
am.cpp    [Note] candidates are:
am.cpp    [Note] Special::Special(Special&)
```

Why does this error occur?
There is a concept called Anonymous object. An anonymous object is an object which has no name. Like in the above example, we can see that the returning object from s1 + s2 is newS but as it is defined in the function it is a local object and it will be destroyed when the function ends. But, as we didn't use the & sign with the return type, a copy of the newS will be returned to the main function and immediately it will be passed to the copy constructor to make its copy into the object s3. So, the object will be anonymous here. Here, it is an important point to discuss that the anonymous objects are only read-able objects. It means that it cannot be

passed to the write-able objects. Here, we need to write const keyword in the prototype of the constructor to make it able to accept the read-only objects as well as write-able objects. This concept also continues with the assignment operator and even the member functions. The key term to remember is that what is an anonymous object? And anonymous objects cannot be passed to the non-const parameter.

--THE END--