Session 5

Week -2

The rule of three is one of the rules of C++ under rules of thumb for building an exception safe code. These rules prescribe how default members of the class should be used for exception free practice.

The rule of three is also known as the Law of Big Three or The Big Three and prescribes for class that, if a class defines any of the mentioned three then it should probably explicitly define all three −

- destructor
- copy constructor
- copy assignment constructor

These three are special member functions of class. If none of them is explicitly defined by the programmer, then the compiler provides implicit versions. If any one of the above is explicitly defined that means implicit versions for the other two must be improper and must be redefined.

This happens because implicitly generated constructors and assignment operators' shallow copy the data members. We require deep copy when class contains pointers pointing to dynamically allocated resources.

The default destructors remove the unused objects. When there is no copy constructor defined then the destructor will run twice, once for objects that contain the copy and second for objects from which the data members are copied. To avoid this, explicit definition becomes necessary.

Let us understand with an example where there is no copy constructor and no copy assignment operator, but destructor is present –

- This happened because the destructor is called twice when the program goes out of scope. First deletion of Num1 and then for Num2. Default copy constructor creates a copy of pointer ptr but not allocated memory for it. So, when Num1 is removed subsequent ptrs cause the program to crash.

Helping Code (For Operator Overloading)

Example(Operator Overloading Concept)

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

using namespace std;

class Test
{
public:
	Test() {}
	Test(const Test& t)
	{
		cout << "Copy constructor called " << endl;
	}

	Test& operator = (const Test& t)
	{
		cout << "Assignment operator called " << endl;
		return *this;
	}
};

// Driver code
int main()
{
	Test t1, t2;
	t2 = t1;
	Test t3 = t1;
	getchar();
	return 0;
}
```

Example (Rule of three)

```
using namespace std;
#include <iostream>
#include <cstdlib>
#include<string.h>

#include <stdio.h>
class Numbers {
private:
    int num;
    int* ptr;
public:
    Numbers(int n, int* p) //copy constructor
    {
        num = n;
        ptr = new int[num];
    }
    ~Numbers() //destructor
    {
        delete ptr;
        ptr = NULL;
    }

    Numbers& operator=(Numbers const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Numbers copy(other);
        return *this;
    }

};
int main() {
    int arr[4] = { 11, 22, 33, 44 };
    Numbers Num1(4, arr);
    // this creates problem
   //Numbers Num2(Num1);
    Numbers Num2 = Num1;

    return 0;
}
```

Dynamic SafeArray Contruction

```
#include <iostream>
#include <cstdlib>
#include<string.h>
using namespace std;
using namespace std;
class atype {
	int ncols;
	int* dynamicArray;
public:		atype()
{
	ncols = 0;
	dynamicArray = new int[ncols];
}
	  atype(int col) {
		ncols = col;
		dynamicArray = new int[ncols];
	  }
	  ~atype() {
		delete[] dynamicArray;
	  }

	  //user inserting elements in 2d array
	  void fillArray() {
		for (int in = 0;in < ncols;++in)
		{
			int value;
			cout << "enter value";
			cin >> value;
			dynamicArray[in] = value;
		}


	  }

	  ////bound checking-safe array implementation
	  int& operator [](int i)
	  {
		if (i<0 || i>  ncols - 1)
		{
			cout << "Boundary Error\n";
			exit(1);
		}
		return dynamicArray[i];
	  }


//////////////////////////////////////////////////////////////////////////////
	  atype& operator=(const atype& rhs)    //assignment operator
	  {
		if (this == &rhs)
			return *this;
		delete[] dynamicArray;
```

```
                ncols = rhs.ncols;
                dynamicArray = new int[ncols];
                memcpy(dynamicArray, rhs.dynamicArray, sizeof(int) * ncols);
                return *this;

        }


};
int main() {
        int columns;
        cout << "enter cols" << endl;
        cin >> columns;
        atype ob1(columns);
        ob1.fillArray();
        atype ob2 = ob1;
        atype ob3;
        ob3 = ob1;
        cout << ob1[1] << endl;
        cout << ob1[4] << endl;     //checking bounds of array

}
```

Exception Handling for Arrays

One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc). C++ provides following specialized keywords for this purpose.
try: represents a block of code that can throw an exception.
catch: represents a block of code that is executed when a particular exception is thrown.
throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Following are main advantages of exception handling over traditional error handling.

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
2) *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.
In C++, a function can specify the exceptions that it throws using the throw keyword.

The caller of this function must handle the exception in some way (either by specifying it again or catching it)
3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Example -1

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

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

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```

Example-2

2) There is a special catch block called 'catch all' catch(…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

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

int main()
{
    try {
        throw 10;
    }
    catch (char* excp) {
        cout << "Caught " << excp;
    }
```

```
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}
```

3) Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int

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

int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Task

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

class Test {
public:
    Test() { cout << "Constructor of Test " << endl; }
    ~Test() { cout << "Destructor of Test " << endl; }
};

int main()
{
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}
```

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

int main()
{
    int foo[] = { 16, 2, 77, 40, 12071 };
    int size = sizeof(foo) / sizeof(int);
    try {
        //throw foo;
        throw (pair<int*, int>(foo, size));
    }
    catch (int* foo)
    {
        for (int i = 0; i < size; i++)
            cout << foo[i] << ",";
    }
    catch (pair<int*, int>& ip)
    {
        cout << "pair.." << endl;
        for (int i = 0; i < ip.second; i++)
            cout << ip.first[i] << ",";
    }
    cout << endl;
}
```

Write a program that creates a **2D array** of **5x5** values of **type boolean**. Suppose indices represent people and that the value at **row i**, **column j** of a **2D array** is true just in case **i and j** are **friends** and **false otherwise**. Use initializer list to instantiate and initialize your array to represent the following configuration: (*** means "friends"**)

	0	1	2	3	4
0		*		*	*
1	*		*		*
2		*			

3	*				*
4	*	*		*	

Write a method to check whether **two people** have a **common friend**. For example, in the example above, **0 and 4** are **both friends with 3** (so they have a common friend), whereas **1 and 2** have **no common friends**.

```
#include<iostream>
using namespace std;
int main()
{
	char a[3][3] = { {'0','x','x'},{'x','0','x'},{'0','x','x'}};

	for (int i = 0; i <= 2; i++)
	{
		for (int j=0; j<=2; j++ )
			{
			cout << " " << "" << a[i][j]<<" ";
		}
		cout << "" << endl;
	}

}

void test()
{


}
```

Specify the size of columns of 2D array

```
void processArr(int a[][10]) {
   // Do something
}
```

Pass array containing pointers

```
void processArr(int *a[10]) {
   // Do Something
}

// When callingint *array[10];
for(int i = 0; i < 10; i++)
   array[i] = new int[10];
processArr(array);
```

Pass a pointer to a pointer

```
void processArr(int **a) {
   // Do Something
}
// When calling:
int **array;
array = new int *[10];
for(int i = 0; i <10; i++)
   array[i] = new int[10];
processArr(array);
```

Algorithm Discussion

```
void test(char b[][3])
{
	int i;
	int a1[3], b1[3], c1[3] = {0};
	cout << "Connected Person"<< endl;
	for (int i = 0; i <= 2; i++) //Working to create list of Connected Ids
	{
		for (int j = 0; j <= 2; j++)
		{
			if (i==0)
			{
				if (b[i][j] == '-')
				{
					a1[j] = j;
					cout << " " << "" << a1[j];
				}
				else
					a1[j] = 0;
			}
			if (i == 1)
			{
				if (b[i][j] == '-')
				{
					b1[j] = j;
					cout << " " << "" << b1[j];
				}
				else
					b1[j] = 0;
			}
			if (i == 2)
			{
				if (b[i][j] == '-')
				{
					c1[j] = j;
					cout << " " << "" << c1[j];
				}
				else
					c1[j] = 0;
			}
		}
		cout << "" << endl;
	}
	cout << "Common Friend" << endl;
	for (int k = 0; k <=2; k++)
	{
			for (int h = 0; h <=2; h++)
			{
				if (a1[h] == b1[h])
				{
					cout << "IS" << h << endl;
				}

				cout << "Common Friend OF 0->" << h <<"Nothing" <<endl;
			}
	}
}
```