National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Computing

CS212: Object Oriented Programming

Class: BSCS-5AB

Lab 09: Composition, Aggregation & Exception Handling in OOP using C++

Date: 02-05-2015

Time: 09:00 am – 12:00 pm / 2pm to 5pm

Instructor: Ms. Hirra Anwar

Lab 09: Exception Handling in OOP C++

Introduction

Exception Handling provides a mechanism to detect and report an exceptional circumstance, so that corrective action can be taken to set right the error occurred. Error Handler in C++ consists of three major keywords namely **try**, **throw** and **catch**. The "**try**" block is a set of code which generates an exception which is thrown using the "**throw**" statement. The "**catch**" block is used to catch the exception and handles it appropriately.

Objectives

- Learn composition and aggregation
- Exception Handling in C++

Tools/Software Requirement

You will need Visual Studio.

Description

Follow all the content given below to understand the concept of the given topics. Run all the given examples to get a better understanding.

COMPOSITION AND AGGREGATION

Composition:

Composition is a kind of association where the composite object has sole responsibility for the disposition of the component parts. The relationship between the composite and the component is a strong "has a" relationship, as the composite object takes ownership of the component. This means the composite is responsible for the creation and destruction of the component parts. An object may only be part of one composite. If the composite object is destroyed, all the component parts must be destroyed. The part has no life of itself and cannot be transferred to another object. Composition enforces encapsulation as the component parts usually are members of the composite object.

Aggregation:

Aggregation is a kind of association that specifies a whole/part relationship between the aggregate (whole) and component part. This relationship between the aggregate and component is a weak "has a" relationship, as the component may survive the aggregate object. The component object may be accessed through other objects without going through the aggregate object. The aggregate object does not take part in the lifecycle of the component object, meaning the component object may outlive the aggregate object. The state of the component object still forms part of the aggregate object.

Examples:

1) Composition:

```
Class A
{
    public:
       A();
      ~A();
};

Class B
{
    public:
       B();
      ~B();
    private:
       A a;//A composition to B
};
```

Aggregation:

```
Class A
{
    public:
       A();
      ~A();
};
```

```
Class B
{
    public:
        B();
       ~B();
        SayHello()
        {
            a = new A();
        }
    private:
        A *a;//A aggregation to B
};
```

In the following examples, pTeacher is created independetly of cDept, and then passed into cDept's constructor. Note that the department class uses an initialization list to set the value of m_pcTeacher to the pTeacher value we passed in. When cDept is destroyed, the m_pcTeacher pointer destroyed, but pTeacher is not deallocated, so it still exists until it is independently destroyed(Aggregation).

```
class Teacher
{
private:
    string m_strName;
public:
    Teacher(string strName)
        : m_strName(strName)
    {
    }

    string GetName() { return m_strName; }
};

class Department
```

```
{
private:
   Teacher *m_pcTeacher; // This dept holds only one teacher

public:
   Department(Teacher *pcTeacher=NULL)
      : m_pcTeacher(pcTeacher)
   {
   }
};

int main()
{
   // Create a teacher outside the scope of the Department
   Teacher *pTeacher = new Teacher("Bob"); // create a teacher
   {
      // Create a department and use the constructor parameter to pass
      // the teacher to it.
      Department cDept(pTeacher);

   } // cDept goes out of scope here and is destroyed

   // pTeacher still exists here because cDept did not destroy it
   delete pTeacher;
}
```

Exception Handling:

Follow the lab practice given below.

Lab Practice 1

```
#include <iostream>

using namespace std;


int main()
 {
   int x,y;
   cout << "Enter values of x::";
   cin >> x;
   cout << "Enter values of y::";
   cin >> y;
   int r=x-y;
   try
     {
      if ( r!=0)
       {
         cout << "Result of division is x/r:: " << x/r << "\n";
       }
      else
       {
         throw ( r);
       }
     }
   catch( int r)
      {
        cout << "Division by zero exception::value of r is::" << r << "\n";
      }
   cout << "END";
   return 0;
 }
```

Result:

```
Enter values of x::12
Enter values of y::12
Division by zero exception::value of r is::0
```

Lab Practice 2

```
#include<iostream>
#include<conio.h>
using namespace std;
void test(int x)
{
  try
  {
         if(x>0)
           throw x;
     else
           throw 'x';
  }

  catch(int x)
  {
         cout<<"Catch a integer and that integer is:"<<x;
  }

  catch(char x)
  {
         cout<<"Catch a character and that character is:"<<x;
  }
}

void main()
{

  cout<<"Testing multiple catches\n:";
  test(10);
  test(0);
  getch();
}
```

Standard exceptions

The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions. It is called exception and is defined in the <exception> header file under the namespace std. This class has the usual default and copy constructors, operators and destructors, plus an additional virtual member function called what that returns a null-terminated character sequence (char *) and that can be overwritten in derived classes to contain some sort of description of the exception.

Read the tutorial on the following link

http://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm

We have placed a handler that catches exception objects by reference (notice the ampersand & after the type), therefore this catches also classes derived from exception, like our myex object of class myexception.

Lab Practice 3

```
// standard exceptions
#include <iostream>
#include <exception>
using namespace std;

class myexception: public exception
{
  virtual const char* what() const throw()
  {
    return "My exception happened";
  }
} myex;

int main () {
  try
  {
    throw myex;
  }
  catch (exception& e)
  {
    cout << e.what() << endl;
  }
  cin.get();
  return 0;
}
```

Lab Practice 4

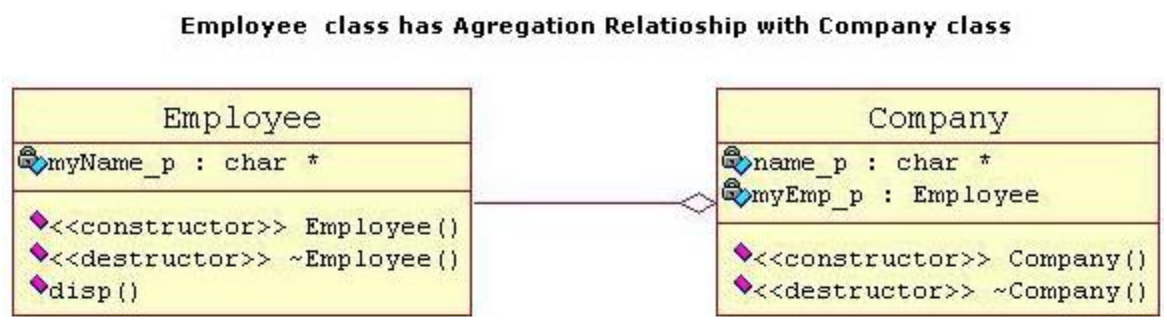
```
#include <iostream>
using namespace std;

//define our exception class
class DivideByZero {};

double divide(int a, int b) {
   if(b == 0)
      throw DivideByZero();
   return (double) a / b;
}

int main() {
   try {
      double d = divide(4, 0);
      cout << d << endl;
   }catch(DivideByZero){
      cerr << "Exception : cannot divide by Zero" << endl;
   }

}
```

Lab Task 1

Implement the following composition example illustrated in the diagram. Implement the required classes.

A single Employee can not belong to multiple Companies (legally), but if we delete the Company, Employee object will not destroy.

Here is respective Model and Code for the above example.


Employee class has Agregation Relatioship with Company class
Employee
myName_p : char *
<<constructor>> Employee()
<<destructor>> ~Employee()
disp()
Company
name_p : char *
myEmp_p : Employee
<<constructor>> Company()
<<destructor>> ~Company()

Lab Task 2

Write two classes' mother.cpp and child.cpp. Both classes can be derived from a base class person.cpp.

```
class Person
{

   protected:
   char name[15];
   int age;
};
```

Member functions of both classes are **getdata()** and **takedata().**By using exception handling technique apply checks on the both classes.

1. Age should not be greater than 100.Raise an exception.
2. Compare the age of mother.cpp and children.cpp raise an exception if the age of child is greater than mother.

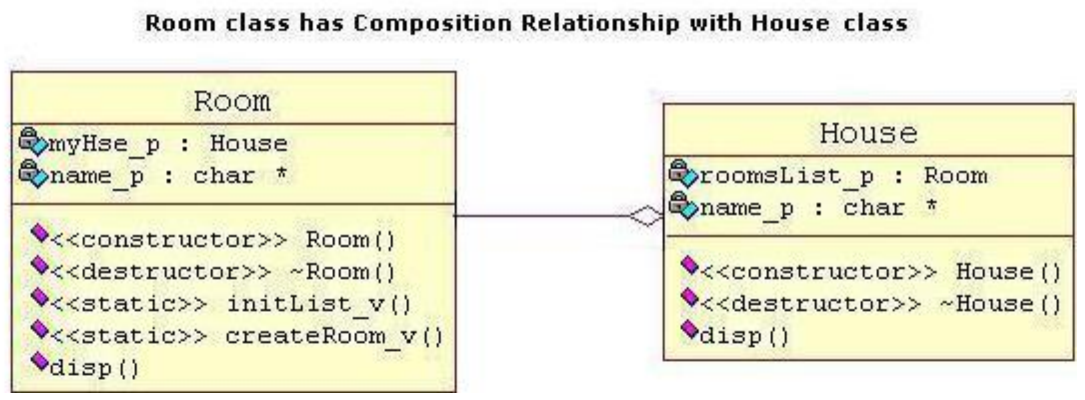
Bonus Question:

Implement the following composition example illustrated in the diagram. Implement the required classes.

Lab Task 3

House can contain multiple rooms there is no independent life for room and any room cannot belong to two different houses. If we delete the house room will also be automatically deleted.

Here is respective Model and Code for the above example.



Deliverables

Source code for examples and Task 1 & Task 2 separately.