Module 19
Exception Handling - II

Introduction

In this module, we will look further into the EH process. We have already seen how exceptions are thrown and caught in the previous module and various ways in which they can be used by library designers and users. In this module, we will look at how one can catch multiple exceptions, replace default EH functions like terminate and unexpected and how the uncaught_exception is used to find out if there is an ongoing exception handling process.

Catching multiple exceptions

We can have a single try block followed by a catch block containing multiple catch statements. A small example program 19.1 illustrates the point. We have now a program which can throw multiple types of exceptions, int, char, and NewEx. We have a catch statement block which contains statements for all these types as well as a catch-all. We invite the user to choose string type which is not managed by the normal catch. The string type, if chosen will be managed by catch-all. Here is the code.

```
//Program 19.1
//MultipleCatch.cpp
#include <iostream>
#include <string>
using namespace std;

class NewEx
{
private:
   int ExNo;
   string ExMsg;
public:
   NewEx(int ErrNum, string t_msg)
   {
      ExNo = ErrNum;
      ExMsg = t_msg;
   }
   void DisplayEx()
   {
      cout << "Error Number is "<< ExNo << "\n";
      cout << ExMsg << "\n";
   }
};

void ExGen()
{
   cout << "Inside ExGen() \n";
   try
   {
      cout << "Inside Try block\n";

      NewEx Error1(20,"Error Testing");
```

```
        cout << "Press 1 for int 2 for char , 3 for NewEx and 4 for string";
        int reply;
        cin >> reply;
        cout << "Assuming error and throwing exception now\n";
        switch(reply)
        {
            case 1:
                throw 10;
            case 2:
                throw 'a';
            case 3:
                throw Error1;
            default :
                throw "This is something else";
        }
        cout << "This statement will not be executed";
    }
    catch (int IntEx)
    {
        cout << "Integer Exception" << IntEx << " is caught in the function \n";
    }
    catch (char)
    {
        cout << "Character Exception is caught in the function \n";
    }
    catch (NewEx Ex)
    {
        Ex.DisplayEx();
    }
    catch (...)
    {
        cout << "Catching anything else\n";
    }
}
int main()
{
    ExGen();// return normally
    return 0;
}
```

You can see that the program has multiple catch statements, two of them for int and char, the built-in types. It is syntactically written to throw such things and catch them but practically they are of little value.

An important point to note is two different types of catch arguments. Look at following two ways of writing arguments in the catch statement.

```
    catch (char)
catch (NewEx Ex)
```

In case of char, only the type (char) is specified but the variable is not specified, in the second case, both the type (NewEx) and the variable (Ex) are specified. Both are valid ways to write the catch statement. If one does not need the variable to store the value of the exception, he may only use the type and if he wants to use the variable thrown by the exception, he should provide the variable as well. In case of a second catch, we would like to have the Ex object as we would like to call the DisplayEx member function for that variable.

Exception specification

What if the library designer expects those who modify the function later to allow only a few exceptions thrown from the function and if any other exception, even by mistake, is thrown than caught at the time of compiling those functions themselves? C++ designers have provided a mechanism for specifying what types of exceptions can be thrown from the function, which is known as exception specification. We will use a specific syntax where we append the function definition header with throw keyword and possible types of exceptions to be thrown in the parenthesis next. Here is an example.

```
void ExGen()throw(int, char)
```

Thus, the ExGen now only allows int and char to be thrown out of the function. If ever the function is modified and throws something other than the specified types, a built-in function unexpected() is called which in turn calls abort() and terminates the program. This is a design level test and important for the designer to manage the exceptions thrown out. The two-step process of calling unexpected which in turn calling abort() is for a reason now known to you. One can modify the behavior of the unexpected function as well.

In short, the exception Specification is a List of allowed exceptions which can be thrown from the function which are not handled inside the function. It is also a mechanism of specifying what types of exceptions can be thrown from the function, and if any other exception is thrown then the program flags an error. We need a specific syntax where we append the function definition header with throw keyword and possible types of exceptions to be thrown in the parenthesis next.

One typical case is where we write a function definition header as follows

```
void ExGen()throw()
```

The empty parenthesis now ensures that the function does not throw any exception now.

Another important point to note is that the exception specification only applies to exceptions thrown outside the function, if the body of the function throws some exception nor part of the exception specification and catches that exception within the body itself, it is fine.

Re-throw

We have seen so far that either the exception is caught at some function or higher up, wherever there is a try and a respective catch block exists. However, we may need to handle the exception in a function and also let it pass up in the hierarchy sometimes. Let us try to understand.

Suppose if a function is passed an int argument which it is going to use as a denominator in some mathematical expression. Unfortunately, that argument is found to be zero. The programmer who is writing this function must write an exception handling code to manage this. He may need to provide throw when that argument is found to be zero, which in turn suspends the execution of the program and wind up everything else before terminating. The programmer writes a catch statement to do all this. Now the user comes with an additional requirement, the caller of the function must be informed about this anomaly. The caller, who passes the arguments, have nothing to do with the files open or connections but it has to be informed about this argument. It is quite possible that these arguments are read from the input and the calling function might need to ask the user to provide input yet again. If the exception is caught by the programmer in his own function, the caller will get a normal return statement and have no idea about what went wrong. If the programmer passes the exception up, the caller has no idea about the resources allocated by the called function and cannot do the winding up process. What we want is to have the programmer handle the exception and pass that up as well. Re-throw is a facility for doing so. Following example clarifies the point. This is the program that we have already seen with two differences, for int and char exceptions, we have an additional throw () statement in the body of the catch statement which catches that exception, which re-throws the exception. We have similar try and catch in main where that exception is caught again and processed in the respective catch block. Here is the program.

```
//Program 19.1
//ReThrow.cpp
#include <iostream>
#include <string>
using namespace std;

class NewEx
{
private:
   int ExNo;
   string ExMsg;
public:
   NewEx(int ErrNum, string t_msg)
   {
      ExNo = ErrNum;
      ExMsg = t_msg;
   }
   void DisplayEx()
   {
      cout << "Error Number is "<< ExNo << "\n";
      cout << ExMsg << "\n";
   }
};

void ExGen()
{
   cout << "Inside ExGen() \n";
   try
   {
      cout << "Inside Try block\n";

      NewEx Error1(20,"Error Testing");
      cout << "Press 1 for int 2 for char , 3 for MyException and 4 for string";
```

```
        int reply;
        cin >> reply;
        cout << "Assuming error and throwing exception now\n";
        switch(reply)
        {
           case 1:
              throw 10;
           case 2:
              throw 'a';
           case 3:
              throw Error1;
           default :
              throw "This is something else";
        }
        cout << "This statement will not be executed";
    }
    catch (int IntEx)
    {
        cout << "Integer Exception" << IntEx << " is caught in the function \n";
        throw(IntEx);
    }
    catch (char CharEx)
    {
        cout << "Character Exception" << CharEx << " is caught in the function \n";
        throw(CharEx);
    }
    catch (NewEx Ex)
    {
        Ex.DisplayEx();
    }
    catch (...)
    {
        cout << "Catching anything else\n";
    }
}
int main()
{
    try {
        ExGen(); }
    catch (int IntEx)
    {
        cout << "Integer Exception" << IntEx << " is caught in the main \n";
    }
    catch (char CharEx)
    {
        cout << "Character Exception" << CharEx << " is caught in the main \n";
    }
    return 0;
}
```

Inside Try block
Press 1 for int 2 for char , 3 for MyException and 4 for string1
Assuming error and throwing exception now
Integer Exception10 is caught in the function
Integer Exception10 is caught in the main

In short, the process of rethrowing is about throwing the very exception that is caught by a catch block from within the catch block higher up.

What if a re-thrown exception is not caught up in the hierarchy? Like an earlier case of a thrown exception not caught, it will invite terminate which in turn calls abort to terminate the program.

The set_terminate () function

The set_terminate () function is used to set a user-defined function to act as the terminate () function rather than system's default function. The default terminates () just calls abort () and does nothing else. If we write our own terminate, the advantage is that we can close the file and deallocate memory etc. before going down. Let us see an example to understand how the set_terminate() can be put to use for informing the system that now onwards use MyTermiante() function when no catch is found for a thrown exception rather than default terminate().

```
//Program 19.2
//SetTermiante.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

class NewEx
{
private:
   int ExNo;
   string ExMsg;
public:
   NewEx(int ErrNum, string t_msg)
   {
      ExNo = ErrNum;
      ExMsg = t_msg;
   }
   void DisplayEx()
   {
      cout << "Error Number is "<< ExNo << "\n";
      cout << ExMsg << "\n";
   }
};

void ExGen()
{
   cout << "Inside ExGen() \n";

   NewEx Error1(20,"Error sample");
   throw Error1;
}
void MyTermiante()
{
   cout << "user defined terminate\n";
   exit(-1);
}
int main()
{
   void MyTerminate();
   set_terminate(MyTermiante);
   cout << "Inside Main\n";
   ExGen();// this would generate exception for which no catch available
   cout <<"This will not be executed";
}
```

Inside Main
Inside ExGen()
user-defined terminate
Program ended with exit code: 255

Closely observe the program. It is quite similar to earlier programs with a few differences you must note.

First, there is a header called exception is additionally provided.

```
#include <exception>
```

This header will allow set_termiante function's prototype to be known. Another is a new function called MyTermiante() as follows. It is the new terminate function which is going to replace the default terminate function.

```
void MyTermiante()
{
   cout << "user defined terminate\n";
   exit(-1);
}
```

It does only two things, indicating that it is user-defined terminate and then call exit(-1). The default terminate is called in the event of runtime error related to exceptions. The terminate in turn calls abort() to terminate the program. The indirection provided through terminate is useful in providing an option to use to specify his own terminate function. This is the advantage of this two-phase solution.

The third thing is to inform the system that the MyTerminate function is now to be used as the terminate function. It is done by calling set_termiante() function and specifying MyTerminate as the argument.

```
set_terminate(MyTermiante);
```

Once this statement is encountered, the EH system calls the MyTermiante function instead of the default one as and when the appropriate catch is not found for an exception. To indicate that, we have neither specified the try catch block in ExGen or the main. When the exception of type NewEx (the Error1) is thrown, it cannot find a matching catch statement and it does not close itself in a normal fashion. It has to call the terminate function but we have replaced the default terminate with our own terminate so the code written in the MyTermiante function is executed as shown in the output. Though we have taken a simple example and so MyTermiante does not do anything else but you can see that the programmer can specify the clean-up job at this place. He can close all open files, terminate connections, indicate the remote server that the client is closing down so the server is aware of the client's departure, deallocate memory which is dynamically allocated and so on. After the advent of C++11's garbage collector system, the last job may not be needed but others are still needed. In real time systems, even when the garbage collector is around, it is advisable to deallocate memory as soon as its usage is over to provide real-time performance.

In summary, a set_terminate is a function which tells the EH system that the argument (which is a name of a function) is now the new terminate function and that function specified in the argument should be used instead of the built-in terminate.

set_unexpected()

Having our own terminate function instead of a default one saves us from abnormal termination when the exception is not caught. There are, alas, other ways the program terminates in an abnormal fashion. When exception specification is provided and the function throws an exception which is not allowed, the program calls unexpected() which in turn calls abort. Setting MyTermiante does not solve this problem. We need another solution here. However, C++ designers have also provided an answer to this situation first by providing a two-step process and second by providing a mechanism to have our own unexpected function to be used instead.

That means, set_unexpected the function instructs the EH system to replace the built-in unexpected function with the user-specified function specified in the argument of set_unexpected.

The example is in order to illustrate how one can define a set_unexpected() function and use it for handling cases where the function throws an exception not allowed by exception specification.

```
//Program 19.3
//SetUnexpected.cpp
#include <exception>
#include <iostream>
#include <string>
using namespace std;

class NewEx
{
private:
   int ExNo;
   string ExMsg;
public:
   NewEx(int ErrNum, string t_msg)
   {
      ExNo = ErrNum;
      ExMsg = t_msg;
   }
   void DisplayEx()
   {
      cout << "Error Number is "<< ExNo << "\n";
      cout << ExMsg << "\n";
   }
};

void ExGen() throw(int,char)
{
   cout << "Inside ExGen() \n";

   NewEx Error1(20,"Error sample");
   throw Error1;
```

```
}
void MyUnExpected()
{
   cout << "user defined unexpected \n";
   exit(-1);
}
int main()
{
   set_unexpected(MyUnExpected);
   void MyUnexpected();

   cout << "Inside Main\n";
   try{
   ExGen();// this would generate exception for which no catch available
   }
   catch (int ExInt)
   {
      cout << "Integer exception is cautht";
   }
   catch(...)
   {
      cout<< "Other than integer exception is caught";
   }
}
```

Inside Main
Inside ExGen()
user defined unexpected
Program ended with exit code: 255

The code contains a few new things for us. Look at the definition of ExGen now.

```
void ExGen() throw(int,char)
{
   cout << "Inside ExGen() \n";

   NewEx Error1(20,"Error sample");
   throw Error1;
}
```

The additional part with throw(int, char) in the definition header indicates that the function is allowed to throw only two types of exceptions, int as well as char. If the function throws any other exception, the unexpected() function should be called. We have already seen that in the default specification, the unexpected calls the abort() and the program terminates as a result. However, in the above-mentioned case, it is not going to happen. We have defined our own unexpected function using the following code.

```
void MyUnExpected()
{
   cout << "user defined unexpected \n";
   exit(-1);
}
```

It is a very simplified user-defined function which only hints that it is user-defined and then calls exit(). We have also instructed the EH system to use this function in the case of violation of exception specification instead of the default unexpected.

```
set_unexpected(MyUnExpected);
```

You can confirm that the user-specified function is called from the output shown. This is how one can do a clean-up job when the unexpected expression is thrown.

The need for two different functions

Why set_termiante and set_unexpected are provided as different functions? Why cannot the designer can simplify the matter and provide a single option? Remember set_unexpected is important for the designer as the unexpected throwing is to be controlled from the designer's end. On the contrary, set_terminate is more important for the library user. He uses the library function and if a library function throws an exception which the user cannot catch, set_termiante is handy. The designer promises that the function X throw only Y set of exceptions, if the function somehow throws other exceptions than covered by Y, that is a design error and must be handled there. In case of terminate, it is called because the user has used some function but has not provided means of handling an error. If an exception is thrown from that function and fails to get proper handler (as the user has forgotten to provide that), terminate is called, this is the user side error and user must provide means for catching and handling such errors.

We have learned to take care of two important issues but there is one more issue that we have to still look at. We specified that the destructors should not be throwing exceptions. If the destructor throws an exception, and some other exception is already thrown and not completely handled, it will result in chaos and program is abnormally terminated. We can write our own terminate function for a clean-up action but in this case, it is possible for us to avoid that case. The destructor is called in two different cases when EH system is in effect. In case 1 when the program is closing in the normal fashion and the destructor is invited for managing objects going out of scope. In case 2 when an exception is thrown and the EH system is deleting objects due to that. In case 1, it is safe to throw an exception (as there is no ongoing EH process), while it is unsafe in the second case. A function called uncaught_exception() returns the status indicating if there is an uncaught exception still left to be handled in the system. A programmer can check if it is so and avoid throwing in that case. This is a better solution than allowing it to take the termination route.

Uncaught_exception()

We have already seen in the previous module that the exception handling process initiates stack unwinding process where all objects defined prior to the execution of the throw statement to the beginning of the try block will be destroyed. There is normally not an issue except for the case that some of the objects which are being destroyed as part of the EH process might have their own destructors. The C++ system has to invoke those destructors as the part of the destruction process. The original exception is still not completely handled. (It is completely handled only after all the objects until the beginning of the try block is

destroyed.) what if any of the object being destroyed and whose destructor is invoked, throws an exception? We know that this is going to create a problem and the program terminates without handling any of the exceptions. We need a way to check if there is still an uncaught (unhandled) exception before throwing. The function uncaught_exception() provides a way out.

In short, Uncaught_exception is a function which returns true if an exception is being handled right now and false otherwise. This function is typically called by destructors to see if it is safe to throw exception or not

The programmer writes the code in the following fashion.

If uncaught_exception() //if there is an exception still to be handled
{do without throwing}
else
{follow normal destruction algorithm and throw if need be}

Let us demonstrate the process of using this uncaught exception using another program.

```
//program 19.4
//UncaughtException.cpp
#include <exception>
#include <iostream>
using namespace std;
class cMsg
{
   char *pMsg;
public:
   cMsg(char Msg[], unsigned long lenMsg)
   {
      pMsg = new char [lenMsg+1];
      for (int i=0;i<lenMsg;i++)
         pMsg[i]=Msg[i];
      pMsg[lenMsg]=0; // Null
   }
   friend ostream & operator << (ostream & tOut, cMsg & tData);
   ~cMsg()
   {
      if (uncaught_exception() == true)
      {
         cout << "There is an uncaught exception in process when ";
         cout << *this;
         delete [] pMsg;
      }
      else
      {
         cout<< "There is no uncaught exception when ";
         cout << *this;
      }
   }
};
   ostream & operator << (ostream & tOut, cMsg & tData)
   {
   tOut << tData.pMsg;
   return tOut;
```

```
    }
    void ExGen()
    {
        char OutMsg[]="Msg defined outside Try block \n";
        char InMsg[]="Msg defined inside Try block \n";
        cMsg oMeg1(OutMsg,strlen(OutMsg));
        try
        {
        cMsg oMsg2(InMsg,strlen(InMsg));
            cout << "Inside the function\n";
            cout << "Inside the try block \n";
            int dummy = 10;

            throw dummy;
        }
        catch (int)
        {
            cout << "Caught dummy\n";
        }
    }
// now the destructor for oMsg1 would be called
// which finds that an exception has already been caught
int main()
{
    ExGen();
    return 0;
}
```

here is an uncaught exception in the process when Msg defined inside Try block
Caught dummy
There is no uncaught exception when Msg defined outside Try block

Explanation

The code described in 19.4 is quite complex and needs explanation. It is also quite different than other programs that we have looked at so far. We have a message class with following code. The message class named as cMsg contains a message as a C type string. We have a constructor which takes the message and the length passed to it and assign pMsg the message that is passed. The interesting part is the design of the destructor. It calls the uncaught_exception() and if it is true it takes a different route than when it is false.

```
class cMsg
{
    char *pMsg;
public:
    cMsg(char Msg[], unsigned long lenMsg)
    {
        //assign message to *pMsg
    }
    …..
    ~cMsg()
    {
        if (uncaught_exception() == true)   {     ..       }
        else
        { ..   }
    }
};
```

Two objects of the cMsg are defined in the program. One is inside the Try block while the second is outside the try block. Remember our discussion of exception handling. The object within the try block must be deleted before the exception is caught. Only when the

exception is caught and the control comes out of the ExGen function, the message defined outside the try block goes out of scope and its destructor is called.

Look at the output of the program. Two destructor messages are displayed for two messages. The destructor message for an outside message is displayed after the exception is caught while the message defines inside is destroyed before and the destructor message indicates so.

When the InMsg is being destroyed the exception (thrown as a dummy) is in progress so the message indicates so. When the OutMsg is being destroyed, the exception is caught and the control has already come out of the ExGen function so there is no uncaught exception when the OutMsg goes out of scope. So when the destructor of OutMsg is invoked, it displays that there is no uncaught exception.

You can also see the use of *this in the destructor to refer to the object being destroyed. It is a real example where there is a need for *this. We have chosen a class with C type string as a member because a constructor with dynamic memory allocation is needed to acquire the space to accommodate the characters. A destructor is a must in such classes.

Summary

In this module, we have seen how a single try block is possible to be followed by multiple catch statements. We have also seen how unhandled exception is managed by calling set_termiante with user-defined terminate function and how exception specification related errors are managed by calling set_unexpected with user-defined unexpected function. A destructor is called when the objects constructed prior to the thrown exception. It can be a problem if the destructor itself throws an exception. There is a function uncaught_exception for checking if there is an uncaught exception ongoing at the time of destructor being executed.