**Lab 13**

**Exception Handling**

# Objectives

**The C++ programming skills that should be acquired in this lab:**

To understand the use of an Exception.

To learn the syntax of throwing and catching an exception.

To learn about user-defined exceptions.

**Exceptions** provide a way to react to exceptional circumstances (like runtime errors) in our program by transferring control to special functions called *handlers*.

To catch exceptions, we must place a portion of code under exception inspection. This is done by enclosing that portion of code in a *try block*. When an exceptional circumstance arises within that block, an exception is thrown that transfers the control to the exception handler. If no exception is thrown, the code continues normally, and all handlers are ignored.

An exception is thrown by using the **throw** keyword from inside the try block. **Exception handlers** are declared with the keyword **catch**, which must be placed immediately after the try block.

# Example 13.1

|  |
| --- |
| // exceptions  #include <iostream>  using namespace std;    int main () {  try  {  throw 20;  }  catch (int e)  {  cout << "An exception occurred. Exception Nr. " << e << endl;  }  return 0;  } |
| Output:  An exception occurred. Exception Nr. 20 |

The code under exception handling is enclosed in a try block. In this example this code simply throws an exception:

throw 20;

A throw expression accepts one parameter (in this case the integer value 20), which is passed as an argument to the exception handler.

The exception handler is declared with the catch keyword. As you can see, it follows immediately the closing brace of the try block. The catch format is similar to a regular function that always has at least one parameter. The type of this parameter is very important, since the type of the argument passed by the throw expression is checked against it, and only in the case they match, the exception is caught.

We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed.

If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last:

try {

// code here

}

catch (int param) { cout << "int exception"; }

catch (char param) { cout << "char exception"; }

catch (...) { cout << "default exception"; }

In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

After an exception has been handled the program execution resumes after the try-catch block, not after the throw statement!.

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception to its external level. This is done with the expression throw; with no arguments. For example:

|  |
| --- |
| try {  try {  // code here  }  catch (int n) {  throw;  } }  catch (...) {  cout << "Exception occurred";  } |

## User Defined Exceptions

You can also create a class to handle exceptions thrown by other classes. Below is the general syntax illustrating an exception handler.

**class AClass //a class**

**{**

**public:**

**class AnError //exception class**

**{**

**;**

**}**

**void Func() //a member function**

**{**

**if( /\* error condition \*/ )**

**throw AnError(); //throw exception**

**}**

**}**

**;**

**void main() //application**

**{**

**try**

**{**

**AClass obj1;**

**obj1.Func(); //may cause error**

**}**

**catch(AClass::AnError) //exception handler**

**{**

**//(catch block)**

**//tell user about error, etc.**

**}**

**}**

## Example 13.2

The following example illustrates throwing and catching exceptions from the Stack class.

|  |
| --- |
| #include<iostream>  using namespace std;  const int MAX = 3; //stack holds 3 integers  class Stack  {  private:  int st[MAX]; //array of integers  int top; //index of top of stack  public:  //---------------------------------------------------  class Range //exception class for Stack  {  //note: empty class body  };  //---------------------------------------------------  Stack() //constructor  {  top = -1;  }  //---------------------------------------------------  void push(int var)  {  if (top >= MAX - 1) //if stack full,  throw Range(); //throw exception  st[++top] = var; //put number on stack  }  //---------------------------------------------------  int pop()  {  if (top < 0) //if stack empty,  {  throw Range();  } //throw exception  return st[top--]; //take number off stack  }  }; |

In the above example, the Range class is handling the exceptions thrown when the stack is either empty or full. We could perfectly define two separate classes to handle the exceptions of each type.

## Example 13.3

|  |
| --- |
| #include<iostream>  using namespace std;  const int MAX = 3; //stack holds 3 integers  class Stack  { private:  int st[MAX]; //stack: array of integers  int top; //index of top of stack  public:    //----------------------------------------------  class Full { }; //exception class  class Empty { }; //exception class  //----------------------------------------------    Stack() //constructor  {  top = -1;  }  //---------------------------------------------------  void push(int var) //put number on stack  {  if(top >= MAX-1) //if stack full,  throw Full(); //throw Full exception  st[++top] = var;  }    //---------------------------------------------------  int pop() //take number off stack  {  if(top < 0) //if stack empty,  throw Empty(); //throw Empty exception  return st[top--];  }  }; |

|  |
| --- |
| void main()  {  Stack s1;  try {  s1.push(11);  s1.push(22);  s1.push(33);  s1.push(44); //stack full  }  catch (Stack::Range) //exception handler  {  cout << "Exception: Stack Full" << endl;  }  //---------------------------------------------------  try {  cout << "1:" << s1.pop() << endl;  cout << "2:" << s1.pop() << endl;  cout << "3:" << s1.pop() << endl;  cout << "4:" << s1.pop() << endl;//stack empty  }  catch (Stack::Range) //exception handler  {  cout << "Exception: Stack Empty" << endl;  }  cout << "Arrive here after catch or normal exit)";  } |

## Exceptions with Arguments

In the above examples, the exception classes we created were empty. It is also possible to have more useful exception classes with data members to store information about the origin and source of exception for example. The following example illustrates this idea for the Distance class.

## Example 13.4

|  |
| --- |
| class Distance  { private:  int feet; float inches; public:  class InchesEx //exception class  {  public:  char origin[20];  float iValue;  InchesEx(string or, float in)  {  strcpy(origin,or);  iValue = in;  }  }; //end of exception class    Distance(int ft = 0, float in = 0.0)  {  if(in >= 12.0)  throw InchesEx("Constructor", in);  feet = ft; inches = in;  }  //------------------------------------------------  void getdist() //get length from user  {  cout << "\nEnter feet: "; cin >> feet; cout << "Enter inches: "; cin >> inches;  if(inches >= 12.0)  throw InchesEx("getdist() function", inches);  }  //------------------------------------------------  void showdist() //display distance {  cout << feet << "\'-" << inches << '\"';  }  };  void main()  { try  {  Distance dist1(17, 3.5); Distance dist2; dist2.getdist();  cout << "\ndist1 = "; dist1.showdist(); cout << "\ndist2 = "; dist2.showdist();  }  catch(Distance::InchesEx ix) //exception handler  {  cout << "\nInitialization error in "  << ix.origin << ".\n Inches value of "  << ix.iValue << " is too large.";  }  cout << endl;  getch();  } |
|  |

Note: Implement all the examples given in this manual.

**Exercise 1**

|  |
| --- |
| Create a class Student with data members age and marks, both of type integer. Provide a parameterized constructor and a set member function to set the data members. The age should be a positive number between 15 and 40 while the marks should be between 0 and 100. Create a class InvalidRange as exception handler of the Student class. The student class should throw an exception if any of its member functions attempts to assign an invalid value to the data members.  In the main program, create objects of class student, assign valid/invalid values to the data members and catch the exceptions accordingly. |

**Code :**

#include <iostream>

using namespace std;

class Student {

int age;

int marks;

public:

Student(int a = 0, int b = 0) {

setAge(a);

setMarks(b);

}

void setAge(int a){

if (a >= 15 && a <= 40) {

age = a;

}

else {

throw(InvalidRange("Invalid Range"));

}

}

void setMarks(int a) {

if (a >= 0 && a <= 100) {

marks = a;

}

else {

throw(InvalidRange("Invalid Range"));

}

}

class InvalidRange {

string err;

public:

InvalidRange(string a) : err(a){}

string getErr() {

return err;

}

};

};

int main() {

try {

Student s1(30, 40);

Student s2(10, -50);

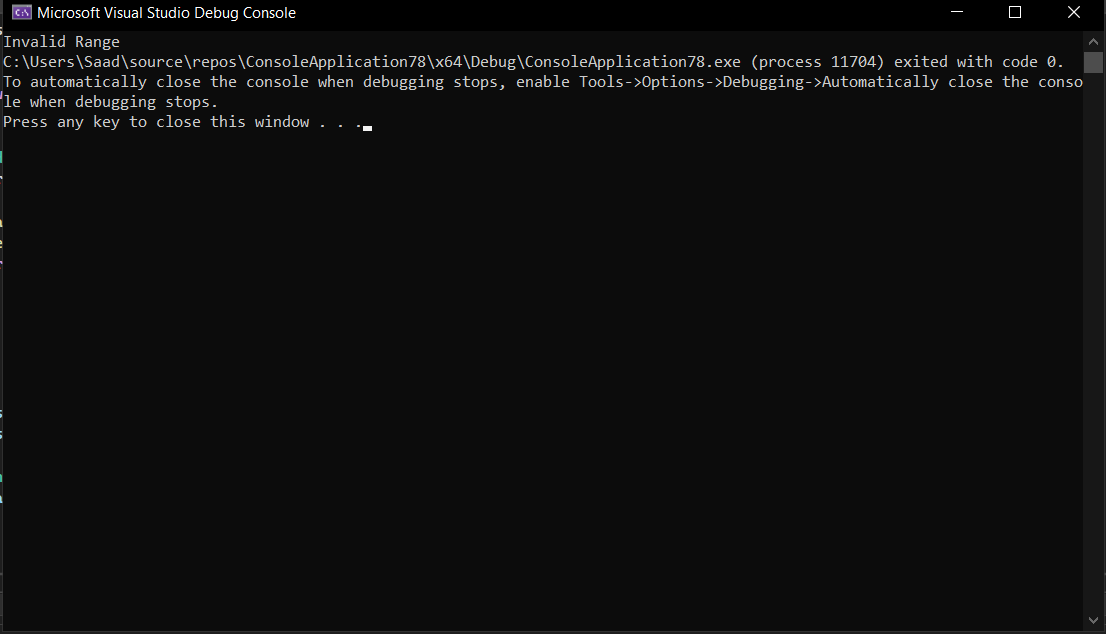
}

catch (Student::InvalidRange a) {

cout << a.getErr();

}

}

****

**Exercise 2**

|  |
| --- |
| Modify the above program to add two private data members (a string and an integer) to the class InvalidRange. Provide appropriate constructor and get methods for the data members. The string should store the name of the function throwing the exception while the integer should store the invalid value the user has supplied. |

#include <iostream>

using namespace std;

class Student {

int age;

int marks;

public:

Student(int a = 0, int b = 0) {

setAge(a);

setMarks(b);

}

void setAge(int a){

if (a >= 15 && a <= 40) {

age = a;

}

else {

throw(InvalidRange("Age",a));

}

}

void setMarks(int a) {

if (a >= 0 && a <= 100) {

marks = a;

}

else {

throw(InvalidRange("Marks",a));

}

}

class InvalidRange {

string err;

int value;

public:

InvalidRange(string a,int b) : err(a),value(b){}

string getErr() {

return err;

}

int getValue() {

return value;

}

};

};

int main() {

try {

Student s1(30, 40);

Student s2(10, -50);

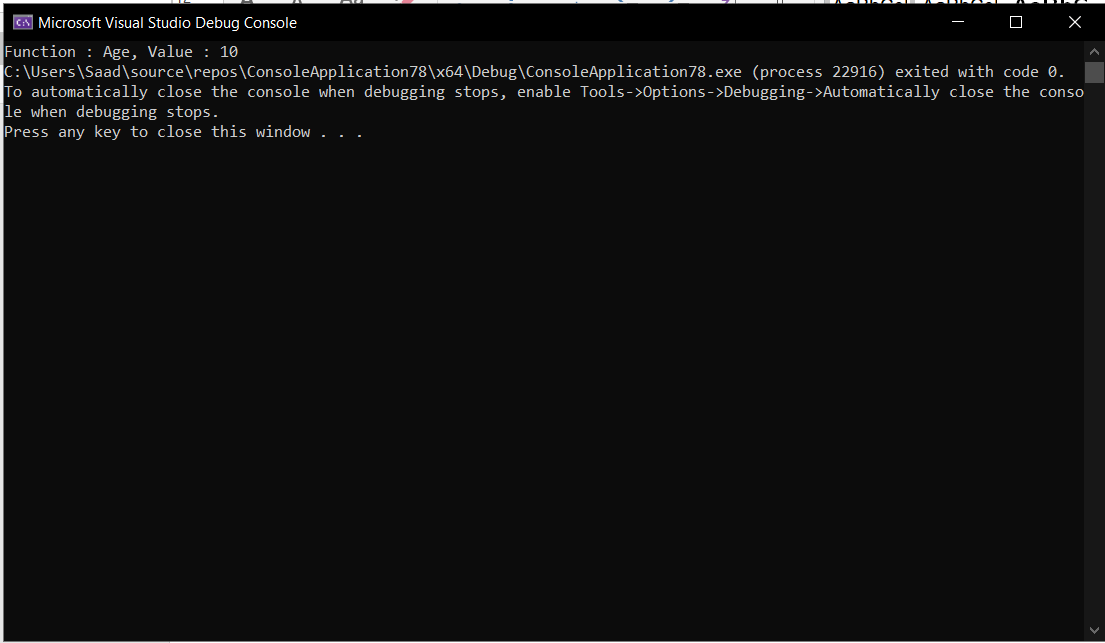
}

catch (Student::InvalidRange a) {

cout << "Function : "<< a.getErr() << ", Value : " << a.getValue();

}

}

****

**Exercise 3**

|  |
| --- |
| Extend the program in Exercise 2 to have two separate classes for exception handling, InvalidMarks and InvalidAge. |

#include <iostream>

using namespace std;

class Student {

int age;

int marks;

public:

Student(int a = 0, int b = 0) {

setAge(a);

setMarks(b);

}

void setAge(int a){

if (a >= 15 && a <= 40) {

age = a;

}

else {

throw(InvalidAge("Age",a));

}

}

void setMarks(int a) {

if (a >= 0 && a <= 100) {

marks = a;

}

else {

throw(InvalidMarks("Marks",a));

}

}

class InvalidMarks {

string err;

int value;

public:

InvalidMarks(string a,int b) : err(a),value(b){}

string getErr() {

return err;

}

int getValue() {

return value;

}

};

class InvalidAge {

string err;

int value;

public:

InvalidAge(string a, int b) : err(a), value(b) {}

string getErr() {

return err;

}

int getValue() {

return value;

}

};

};

int main() {

try {

Student s1(30, 40);

Student s2(15, -50);

}

catch (Student::InvalidAge a) {

cout << "Function : "<< a.getErr() << ", Value : " << a.getValue();

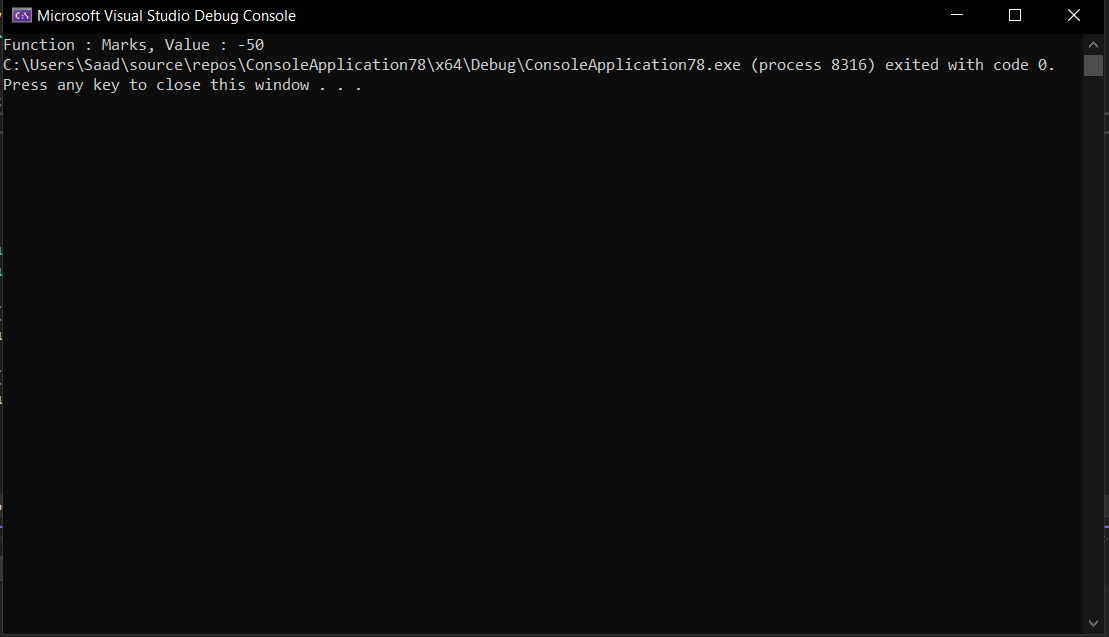
}

catch (Student::InvalidMarks a) {

cout << "Function : " << a.getErr() << ", Value : " << a.getValue();

}

}

****

+++++++++++++++++++++++++