Exceptions

Exceptions:
Exceptions are used to signal errors or unexpected events that occur while a program is running.

Error testing is usually a straightforward process involving if statements or other control mechanisms. For example, the following code segment will trap a division-by-zero error before it occurs:

```cpp
if (denominator == 0)
    cout << “ERROR: cannot divide by zero, \n”;
else
    quotient = numerator / denominator;
```

But what is similar code is part of a function that returns the quotient as in the following example:

```cpp
double divide (int numerator, int denominator)
{
    if (denominator == 0)
        { cout << “ERROR: cannot divide by zero, \n”; return 0; }
    else
        return (static_cast<double>(numerator) / denominator);
}
```

Function commonly signal error conditions by returning a predetermined value. The function is this example returns 0 when division by 0, however this is unreliable, because 0 is a valid result of a division. Even though the function displays an error message, the part of the program that calls the function cannot check whether an error has occurred.
Throwing an exception:
One way of handling complex error conditions is with exceptions.
An exception is a value or an object that signals an error. When the error occurs, an exception is “thrown”. For example, the following code shows the divide function, modified to throw an exception when division by zero has been attempted.

```cpp
double divide( int numerator, int denominator)
{
    if (denominator == 0)
        throw "ERROR: cannot divide by zero, \n";
    else
        return static_cast<double>(numerator) / denominator;
}
```

The following statement causes the exception to be thrown:

```
throw "ERROR: cannot divide by zero, \n";
```

The throw key word is followed by an argument, which can be any value. The nature of the argument is used to determine the nature of the error. The function above simply throws a string containing an error message.

The line containing a throw statement is known as the throw point. When a throw statement is executed, control is passed to another part of the program known as an exception handler. When an exception is thrown by a function, the function aborts.
Handling an Exception

To handle an exception, a program must have a try/catch construct. The general form of try/catch construct is:

```
try
{
    //code here calls functions or object member
    //function that might throw an exception
}
catch (ExceptionParameter)
{
    // code here handles the exception
}
//Repeat as many catch blocks as needed
```

The first part of the construct is the `try` block. This starts with the key word try and is followed by a block of code executing any statements that might directly or indirectly cause an exception to be thrown.

The try block is immediately followed by one or more catch blocks, which are the exception handlers. A catch block starts with the key word catch, followed by a set of parentheses containing the definition of an exception parameter.

For example, here is a try/catch construct that can be used with the divide function:

```
try
{
    quotient = divide( num1, num2);
    cout<< “The quotient is “<< quotient <<endl;
}
catch (char *exceptionString)
{
    cout << exceptionString;
```
This program demonstrates an exception being thrown and caught.

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

//function prototype
double divide(int, int);

int main()
{
    int num1, num2; //to hold two numbers
    double quotient; //To hold the quotient of the numbers

    //Get two numbers
    cout << "Enter two numbers: ";
    cin >> num1 >> num2;
    try
    {
        quotient = divide(num1, num2);
        cout << " The quotient is " << quotient << endl;
    }
    catch (char *exceptionString)
    {
        cout << exceptionString;
    }
    cout << "End of the program \n";

    return 0;
}
```
double divide (int numerator, int denominator)
{
    if (denominator == 0)
        throw "ERROR: Cannot divide by zero.\n";
    return static_cast<double>(numerator) / denominator;
}
After the catch block has finished, the program resumes with the first statement after the try/catch construct.
What if an Exception is Not Caught?

There are two possible ways for a thrown exception to go uncaught. The first possibility is for the try/catch construct to contain no catch blocks with an exception parameter of the right data type.
The second possibility is for the exception to be thrown from outside the try block. In either case, the exception will cause the entire program to abort execution.

Object Oriented Exception Handling with Classes:

In an earlier assignment, we had to define a Rectangle class. That class had two transformer functions setWidth and setLength for setting the rectangle dimensions. If a negative value was passed to either of these functions, the class displayed an error message and aborted the program. The following code shows an improved version of the Rectangle class. This version throws an exception when a negative value is passed to setWidth or setLength

```cpp
// Specification file for the Rectangle class
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
    private:
        double width; // The rectangle's width
        double length; // The rectangle's length

    public:
        // Exception class
        class NegativeSize {
            
        };

        // Default constructor
        Rectangle()
            { width = 0.0; length = 0.0; }

        // Mutator functions, defined in Rectangle.cpp
```
void setWidth(double);
void setLength(double);

// Accessor functions
double getWidth() const
{ return width; }

double getLength() const
{ return length; }

double getArea() const
{ return width * length; }

#endif

Notice the empty class declaration that appears in the public section. The NegativeSize class has no members. The only important part of the class is its name which will be used in the exception handling code.

// Implementation file for the Rectangle class.
#include "Rectangle.h"

/******************************
******************************
void Rectangle::setWidth(double w)
{
    if (w >= 0)
        width = w;
    else
throw NegativeSize();
}

//******************************************************************************
// setLength sets the value of the member variable length. *
//******************************************************************************

void Rectangle::setLength(double len)
{
    if (len >= 0)
        length = len;
    else
        throw NegativeSize();
}

In the setWidth function, the parameter w is tested by the if statement. If the w is greater than or equal to 0, its value is assigned to the width member variable. If w holds a negative number, the statement
throw NegativeSize(); is executed

The throw statement argument NegativeSize() causes an instance of the NegativeSize() class to be created and thrown as an exception.

The same thing happens in the setLength function..

This way of reporting errors is much more graceful than simply aborting the program. Any code that uses the Rectangle class must have a catch block to handle the NegativeSize exceptions that the Rectangle class might throw.
// This program demonstrates Rectangle class exceptions.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
    int width;
    int length;

    // Create a Rectangle object.
    Rectangle myRectangle;

    // Get the width and length.
    cout << "Enter the rectangle's width: ";
    cin >> width;
    cout << "Enter the rectangle's length: ";
    cin >> length;

    // Store these values in the Rectangle object.
    try
    {
        myRectangle.setWidth(width);
        myRectangle.setLength(length);
        cout << "The area of the rectangle is "
             << myRectangle.getArea() << endl;
    }
    
}
catch (Rectangle::NegativeSize)
{
    cout << "Error: A negative value was entered.\n";
}
cout << "End of the program.\n";
return 0;

The catch statement catches the NegativeSize exception when it is thrown by any of the statements in the try block. Inside the catch statement’s between () is the name of the NegativeSize class.

Multiple exceptions:
In many cases a program will need to test for several different types of errors and signal which one has occurred. C++ allows you to throw and catch multiple exceptions. The only requirement is that each different exception be of a different type. You then code a separate catch block for each type of exception that may be thrown in the try block. For example, suppose we wish to expand the Rectangle class so it throws one type of exception when a negative value is specified for the width, and another type of exceptions when a negative value is specified for the length. We declare two different exception class, such as:

// Exception class for a negative width
class NegativeWidth { };  
class NegativeLength { };  

An instance of NegativeWidth will be thrown when a negative value is specified for negative width and an instance of NegativeLength will be thrown when a negative value is entered for the length.
Using exceptions to Recover from Errors:
Exception handling can be an effective way to recover from exceptions and get valid data from the user.

Modify the above code to include error recovery:

```cpp
bool tryAgain = true;

while (tryAgain)
{
    try
    {
        myRectangle.setWidth(width);
        tryAgain = false;
    }
    catch (Rectangle::NegativeWidth)
    {
        cout << "Please enter a non-negative value: ";
        cin >> width;
    }
}

//Complete the code for NegativeLength
```

Extracting Data from the Exception Class:

Sometimes, we might want an exception object to pass data back to the exception handler. Suppose, we would like the Rectangle class not only to signal when a negative value has been given, but also to pass the value back. This can be done, by giving the exception class members in which data can be stored.
Next, we expand the NegativeWidth class with a member variable and a constructor:

class NegativeWidth
{
    private:
        int value;

    public:
        NegativeWidth(int val)
        {
            value = val;
        }
        int getValue() const
        {
            return value;
        }
};

When we throw this exception, we want to pass the invalid value as an argument to the class’s constructor. This is done with the following statement in the setWidth member function:
throw NegativeWidth(w);

This throw statement creates an instance of the NegativeWidth class and passes a copy of the w variable to the constructor. The constructor then stores this number in the NegativeWidth’s member variable: value. The class instance carries this member variable to the catch block that intercepts the exception.
catch (Rectangle:: NegativeWidth e)
{
    cout << “Error: “ << e.getValue()
    << “ is an invalid value for the rectangle’s width. \n”;
}

The catch block defines a parameter object named e. This is necessary because we want to call the call’s getValue function to retrieve the value that caused the exception.