Visual C++

Lecture #14 Exception Handling

Introduction

The Visual C++ language provides built-in support for handling anomalous situations, known as exceptions, which may occur during the execution of an “unhandled” program. Exceptions are unexpected but frequently foreseeable errors that happen in a “unhandled” programs. “Exception handling” is a mechanism in which a programming construct is used to consistently trap, intercept and handle the error occurred during application execution. Interestingly, in most of the time, a human reader can possibly detect the program errors in the code. For example, no number can be divided by zero. In the following code, programmers will hit the situation when denominator (i) is zero. Since the error is now expected, programmers should try to handle it.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main() {
    String^ str = "";
    for (int i=-4; i<=4; i++)
    {
        str += (5/i) + " ";
    }
    MessageBox::Show(str);
}
```

In the above code, the counter variable i in the for loop will have a chance to equal 0. When i=0, the operation of 5/i will cause an error because nothing can be divided by zero. The above code will cause the following error message during the compilation.

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero....
```

Obviously, exceptions are run-time anomalies, such as division by zero, that require immediate handling when encountered by your program. In Visual C++, the process of raising an exception is called “throwing” an exception. A designated exception handler then “catches” the thrown exception.

The following is a sample code that has nothing wrong in syntax. However, it produces an unexpected result, possibly because the type casting fails. The compiler probably does not know how to convert a word “A” to from String to int type.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    int x = Convert::ToInt32("Apple");
    MessageBox::Show(x + "");
}
```
The above code generates the following error message. It also proves that exceptions are not syntax errors, they are just “unexpected” conditions.

```
Unhandled Exception: System.FormatException: Input string was not in a correct format.....
```

The following code, however, does not cause any exceptions because the string literal “1024” can be easily converted from `String` to `int` type. In the following code, there is no such thing called “unexpected condition”.

```
int main()
{
    int x = Convert::ToInt32("1024") * 2;

    MessageBox::Show(x + ",");
}
```

The `try..catch` structure

In Visual C++, programmers use the `try`, `catch`, and `finally` keywords to handle foreseeable exceptions. In a nutshell, a `try` block encloses one or more statements that might throw an exception. The `catch` block is the exception handler which is the handler that catches the exception. The following is the generic syntax. When an exception is defined by the .Net Framework, it is known as an `exception filter`. For example, `System.FormatException` is an exception filter because it has been defined.

```
try
{
    // Code to try goes here.
}
catch (ExceptionFilter^ e)
{
    // Code to handle the exception goes here.
}
```

The following demonstrates how to use the `try..catch` structure to handle the “`System.FormatException`” exception as shown in one of previously codes. The “try” part will make an attempt or effort to execute the problematic statement: `Convert::ToInt32("Apple")`. When the exception occurs, the “catch” part will immediately take over the process to handle the “unexpected condition” by replacing the problematic statement with `x = 0`. As a result, the program will not be forced to terminated in the middle of execution due to the “unexpected condition”. The program will display 0 as result.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    int x;

    try
    {
        x = Convert::ToInt32("Apple"); // exception
    }
    catch (FormatException^ e)
    {
        x = 0;
    }
```
The following demonstrates how to use a try..catch structure to handle the \((5/i)\) expression when \(i\) equals 0 (attempting to divide by zero). Inside the “catch” structure, the continue keyword force the process to skip to the next iteration. In other words, it will skip the iteration when \(i\) equals to 0, and jump the next iteration in which \(i\) equals 1. In the catch block, the code specifies that there might be an exception defined in the System.DivideByZeroException which will be thrown when there is an attempt to divide an integral or decimal value by zero.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ msg = "";
    String^ str = "";
    for (int i = -4; i <= 4; i++)
    {
        try
        {
            str += (5/i) + " "; // when i=0
        }
        catch (DivideByZeroException ^ e)
        {
            continue; // jump to next iteration
        }
    }
    MessageBox::Show(str);
}
```

The code will first attempt to let 5 divides by the current value of \(i\). The division should go through unless \(i = 0\). When \(i = 0\), the exception happens, and this exception will be caught by the catch statement. In the catch statement there is a parameter \(e\) declared as Exception type which represents errors that occur during application execution. According to the above code, the error message is assigned to a variable named “msg”, so it will not bounce to the console screen. The continue statement also forces the run-time to jump to next loop \((i=1)\). With this exception handling mechanism. The above program will run smoothly without any interruption. The error message will not be displayed.

```
-1 -1 -2 -5 5 2 1
```

The following figure explains why the above code only displays 8 numbers, because the \((5/0)\) expression is skipped. The trick is to abandon the execution of \((5/0)\) because it is a foreseeable exception.

<table>
<thead>
<tr>
<th>(i)</th>
<th>-4</th>
<th>-3</th>
<th>-2</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5/i)</td>
<td>5/(-4)</td>
<td>5/(-3)</td>
<td>5/(-2)</td>
<td>5/(-1)</td>
<td>continue</td>
<td>5/1</td>
<td>5/2</td>
<td>5/3</td>
<td>5/4</td>
</tr>
<tr>
<td>output</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In a nutshell, the try block is used by C# programmers to partition code that might be affected by an exception. The try block contains the guarded code that may cause the exception. During the execution of the program, statements inside the try block are the ones to be executed until
an exception is encountered. When the exception is detected, the associated catch block is used to handle the exception.

The following code will cause a **NullReferenceException** exception because a managed Object type cannot be casted to the unmanaged int type. The program will be forced to terminated when it encounters the exception. In Visual C++, **NullReferenceException** is the exception that is thrown when there is an attempt to dereference a null object reference.

```cpp
int main()
{
    Object^ o2 = nullptr;
    int i2 = (int) o2; // exception
    MessageBox::Show( i2 + "")
}
```

The following attempts to cast a null object that could raise the NullReferenceException exception, because the problematic statement is now placed inside a try block. When the exception is detected, the catch block will handle the exception by displaying a default message provided by the Message property, which is a property provided by the NullReferenceException class of the .Net Framework to describe the exception. The trick is to pass the exception message from runtime to a message box, thus, exception will not force the execution to terminate. In other words, the handling is to facilitate the runtime to redirect the error message, so the program will run smoothly without interruption.

```cpp
Object^ o2 = nullptr;
try
{
    int i2 = (int) o2; // exception
    MessageBox::Show( i2 + "")
}
catch (NullReferenceException^ e)
{
    MessageBox::Show( e->Message + "")
}
```

Visual C++ provides a generic System.Exception class for handling exceptions that cannot be identified. In the above example, the catch block specifies to detect an exception defined in the System.NullReferenceException class. The following demonstrates how to use the generic System.Exception class as filter because the Exception class is the base class of all exceptions.

```cpp
Object^ o2 = nullptr;
try
{
    int i2 = (int) o2; // exception
    MessageBox::Show( i2 + "")
}
catch (Exception^ e)
{
    MessageBox::Show( e->Message + "")
}
```

While the Exception class is the base class, Microsoft suggests programmers to specify the specifically-defined exception filters if possible. When an exception occurs in a try block, the system searches the associated catch blocks in the order they appear in application code, until it locates a catch block that handles the exception. A catch block that handles a specified exception can quickly and more precisely handle the exception compared to the Exception
class. Therefore, if the type filter can be identified, programmer should attempt to use it as filter.

It is possible to use more than one specific catch blocks in the same try..catch structure. In this case, the order of the catch blocks is important because the catch blocks are examined in sequence. Interestingly, the program is not always smart enough to catch the more specific exceptions before the less specific ones; however, it will at least attempt to do so. The compiler might thus produce an error if the sequence of catch blocks seems to imply that the sequence indicates the applicability. In this case, a later catch block which could be more appropriate might never be reached. In the following example, two catch blocks are used. The most specific exception, which comes first, is always caught; therefore, the second one is never used.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

void ProcessString(String^ s) {
    if (s == nullptr) {
        throw gcnew ArgumentNullException();
    }
}

int main()
{
    String^ s = nullptr; // For demonstration purposes.

    try {
        ProcessString(s);
    }
    // Most specific:
    catch (ArgumentNullException^ e) {
        MessageBox::Show(e->ToString());
    }
    // Least specific:
    catch (Exception^ e) {
        MessageBox::Show(e->ToString());
    }
}
```

### Common Exceptions

Many “exceptions” have been reported by programmers and have thus been collected by the .Net Framework. The following table lists common exceptions defined by the System namespace.

<table>
<thead>
<tr>
<th>Exception class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticException</td>
<td>A base class for exceptions that occur during arithmetic operations, such as System.DivideByZeroException and System.OverflowException.</td>
</tr>
<tr>
<td>ArrayTypeMismatchException</td>
<td>Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.</td>
</tr>
<tr>
<td>DivideByZeroException</td>
<td>Thrown when an attempt to divide an integral value by zero occurs.</td>
</tr>
</tbody>
</table>
IndexOutOfRangeException | Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.

InvalidCastException | Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.

NullReferenceException | Thrown when a null reference is used in a way that causes the referenced object to be required.

OutOfMemoryException | Thrown when an attempt to allocate memory (via new) fails.

OverflowException | Thrown when an arithmetic operation in a checked context overflows.

StackOverflowException | Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.

TypeInitializationException | Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

An arithmetic operation produces a result that is outside the range of the data type returned by the operation. The following illustrates a typical “DivideByZeroException”.

```c++
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    MessageBox::Show((7/0) + "");
}
```

The following example illustrates the `OverflowException` that is thrown by a multiplication operation that overflows the bounds of the `Int16` type. `OverflowException` is the exception that is thrown when an arithmetic, casting, or conversion operation in a checked context results in an overflow. The value 999999999 is a number far larger than the largest number an `Int16` type can handle, so it will throw an `OverflowException` exception.

```c++
int main()
{
    Int16 x = Convert::ToInt16(999999999);
}
```

The above example causes the following error message.

Unhandled Exception: System.OverflowException: Value was either too large or too small for an Int16.

The problem will go away if the `Int32` type is used to declare the variable “cube”. This is because the `Int32` type support up to 2,147,483,647 which is larger than 999,999,999. Therefore, there is no chance for `OverflowException` exception to happen.

```c++
int main()
```
The following presents a sample way to handle this exception with `try..catch` statement.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ str = nullptr;
    try
    {
        Int16 x = Convert::ToInt16(999999999);
        str = x::typeid + "";
    }
    catch (OverflowException^ e)
    {
        Int32 x = Convert::ToInt32(999999999);
        str = x::typeid + "";
    }
    MessageBox::Show(str);
}
```

A `NullReferenceException` occurs when the programmer tries to use a method or property of a reference type whose value is null. The following is a sample code that will cause a `NullReferenceException` because the "obj" object is set to null; therefore, the `GetType()` method would not find the reference to return a value; therefore, it raises an exception of null reference.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    Object^ obj = nullptr;
    MessageBox::Show(obj->GetType() + "");
}
```

The following is the error message caused by the above code.

```
Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an object.
```

The `OutOfMemoryException` is triggered by allocation instructions and is thrown by the execution engine. It can occur during any allocation call during runtime, and there are ways to predict the failure. The following, for example, attempts to allocate a large amount of memory which is not physically available.

```csharp
int main()
{
    long* data = new long[1000000000];
}
The following is a simple program that raises out-of-memory exception.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ s = gcnew String('a', Int32::MaxValue);
}
```

This code raises the following exception. The above program attempts to allocate a string that is extremely large and would occupy four gigabytes of memory. Therefore, the `OutOfMemoryException` is thrown by the runtime because this attempt is not achievable.

Unhandled Exception: OutOfMemoryException

The following makes more sense to the compiler because it attempts to allocate 1024 bytes for the character ‘a’.

```csharp
int main()
{
    String^ s = gcnew String('a', 1024);
}
```

The following demonstrates how to apply a `try..catch` structure to handle the exception.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ c = nullptr;
    try
    {
        c = gcnew String('a', Int32::MaxValue);
    }
    catch (OutOfMemoryException^ e)
    {
        c = gcnew String('a', 1024);
    }
    MessageBox::Show(c + "");
}
```

The `ArrayTypeMismatchException` is a type of exception that is thrown when an attempt is made to store an element of the wrong type within an array. In the following example, the “obj” array obtains all its elements from the “x” array except that the data type is casted from `String` to `Object`. However, the type casting is not permeant. After the duplication of “x” array completes, elements of the “obj” array remain being `String` type. The attempt to change the value of `obj[1]` to 12 will cause the `ArrayTypeMismatchException`.

```csharp
#using <System.dll>
```
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    array<String^>^ x = {"Cow", "Dog", "Cat", "Fish", "Goat"};
    array<Object^>^ obj = (array<Object^>^) x;
    obj[1] = (Object^) 12; // exception
}

The following adds a try..catch structure to handle such exception.

#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    array<String^>^ x = {"Cow", "Dog", "Cat", "Fish", "Goat"};
    array<Object^>^ obj = (array<Object^>^) x;
    try
    {
        obj[1] = (Object^) 12;
    }
    catch (ArrayTypeMismatchException^ e)
    {
        obj[1] = Convert::ToString(12);
    }
    MessageBox::Show(obj[1] + "");
}

An **InvalidCastException** exception is thrown when an explicit conversion from a base type or interface to a derived type fails at run time. For example, in Visual C++, the Object class is the base class of Int32 class (which has an alias int). It is not possible to cast the data type from Object to int. In the following example, the variable o is declared as an Object type which holds a string literal “0x3A41”.

Object^ o = "0x3A41";

The following statement attempts to convert “o” to int. This is not doable and will throw an InvalidCastException exception.

    int i;
    i = (int) o;

The **Convert::ToInt32(string, 16)** method can convert a string literal that represent a hexadecimal (base 16) value (e.g. “0x3A41”) to an integer of decimal (base 10) value. For example, the following code displays 14913 because 0x3A41 in hexadecimal is 14913 in decimal.

#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    int i = Convert::ToInt32("0x3A41", 16);
    MessageBox::Show(i.ToString());
}

Therefore, programmer can first convert the o variable from Object type to String^ type using
the ToString() method, and then use the ToInt32() method to convert the string literal from a
hexadecimal to an base 10 integer.

    int i = Convert::ToInt32(Convert::ToString(o), 16);

The following is a sample that uses try..catch to handle the exception.

#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    Object^ o = "0x3A41";
    int i;
    try
    {
        i = (int) o;
    }
    catch (InvalidCastException^ e)
    {
        i = Convert::ToInt32(Convert::ToString(o), 16);
    }
    MessageBox::Show(i.ToString());
}

**IndexOutOfRangeException** is another frequently seen exception which is thrown when an
attempt is made to create, identify, or access an element of an array with an index that is larger
than the last index of the array. In the following, the x array can only have 5 elements,
therefore, the largest index is 4. Yet, the for loop attempts to create 8 elements When the for
loop attempts to create the x[5] element (which is the sixth element of the x array), an
IndexOutOfRangeException happens.

    int[] x = new int[5];

    for (int i=0; i<=7; i++)
    {
        x[i] = i * i;
    }

The above code will cause the following error message.

    Unhandled Exception: System.IndexOutOfRangeException: Index was
outside the bounds of the array.....
The following demonstrates how to add a generic try block and a catch block to handle the exception. When an exception occurs, the continue statement will force the for loop to abandon the current iteration and skip to the next iteration.

```cpp
int main()
{
    array<int^>^ x = gcnew array<int^>(5);
    for (int i=0; i<=7; i++)
    {
        try
        {
            x[i] = i * i;
        }
        catch(IndexOutOfRangeException^ e)
        {
            continue;
        }
    }
}
```

It is necessary to note that the catch block can be empty, as shown below. In this case, it will catch the exception and keep the program running smoothly, but the catch block will not do any other thing.

```cpp
#using <System.dll>
#using <System.Windows.Forms.dll>
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    Object^ o2 = nullptr;
    try
    {
        int i2 = (int) o2;    // Error
    }
    catch (NullReferenceException^ e) {
    }
}
```

Although the catch block can be used without statements, this is not a recommended practice. In general, programmers should strive to catch exceptions that have been identified.

**The throw statement**

A throw statement can be used in a catch block to re-throw the exception that is caught by the catch statement. In other words, when placing inside a catch block, the throw statement can raise an exception. The following uses one of the previously discussed code as example, in which a catch block already handles the NullReferenceException exception; therefore, the program should not display any error message on screen. Ironically, the throw statement in the following example re-throws the exception again.

```cpp
int main()
{
    Object^ o2 = nullptr;
    try
    {
```
int i2 = (int) o2;  // Error
}
catch (NullReferenceException^ e)
{
  throw;
}
}

As a matter of fact, programmer can catch the original exception using the catch block, and then use the throw a custom-made exception, as shown in the following example.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
  Object^ o2 = nullptr;

  try
  {
    int i2 = (int) o2;  // Error
  }
  catch (NullReferenceException^ e)
  {
    throw gcnew InvalidCastException("Customized message.");
  }
}
```

The above code generates the following custom-made error message.

Unhandled Exception: System.InvalidCastException: Customized message.

With the throw statement, programmer can also raise an exception outside a catch block. In the following example, the try block contains a call to the ProcessString method which may cause an exception. The catch block contains the exception handler which will just display a message on the screen. When the exception occurs, the throw statement will be called, create an instance of the ArgumentNullException class and display an error message. Then, the program will move to the catch block and display the message provided by the Exception filter.

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

void ProcessString(String^ s)
{
  if (s == nullptr)
  {
    throw gcnew ArgumentNullException();
  }
}

int main()
{
  String^ s = nullptr;  // For demonstration purposes.

  try
  {
```
{  
    ProcessString(s);  
}

catch (Exception^ e)
{
    MessageBox::Show(e->ToString());
}
}

The output in the console is:

SystemArgumentNullException: Value cannot be null.
at TryFinallyTest.Main() Exception caught.

The **finally** block

The **finally** statement is a Microsoft extension to the C and C++ languages that enables target applications to guarantee execution of cleanup code when execution of a block of code is interrupted. A **finally** block enables programmer to clean up actions that are performed in a **try** block. If present, the **finally** block executes last, after the **try** block and any matched **catch** block. A **finally** block will always run, regardless of whether an exception is thrown or a **catch** block matching the exception type is found. A generic format in Visual C++ is to place the **finally** block below the last **catch** block.

```
try
{
    // Code to try goes here. 
}
catch (SomeSpecificException^ e)
{
    // Code to handle the exception goes here. 
}
finally
{
    // Code to execute after the try block goes here. 
}
```

In the following example, the **finally** block will always run regardless to **try** and **catch** block. During each iteration of the **for** loop, either the **try** or the **catch** block will run depending on whether the exception is detected, but the **finally** block must run no matter what.

```
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ str1 = nullptr;
    String^ str2 = nullptr;

    for (int i = -4; i <= 4; i++)
    {
        try
        {
            str1 += (5/i) + "    "; 
        }
        catch (Exception^ e)
        {
            str1 += "    ";
        } 
```
In the following example, the `catch` block will handle the exceptions that happens when the `x[i] = i * i` statement attempts to create the 6th, 7th, and 8th element, because the “x” array can only have 5 elements. During each iteration of the `for` loop, the `finally` block will check if the current value of `i` is less than the “size” (or “length”) of “x” array. The “size” (or “length”) of an array is the total number of elements the array is allowed to have. The “Length” property returns the value of the “size”.

```cpp
using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ str = ""
    array<int^>^ x = gcnew array<int^>(5);
    for (int i=0; i<=7; i++)
    {
        try
        {
            x[i] = i * i;
        } catch(IndexOutOfRangeException^ e)
        {
            continue;
        }

        finally
        {
            if (i< x->Length) { // the Length property
                str += x[i] + " ";
            }
        }
    }
}

MessageBox::Show(str);
```

In the following example, the `finally` block is used to close a file that is opened in the `try` block. The state of the file handle is checked before the file is closed. If the `try` block cannot open the file, the file handle still has the value null, so the arrangement is not to use the `finally` block to try to close it. Alternatively, if the file is opened successfully in the `try` block, the `finally` block will close the open file.

```cpp
using namespace System;
using namespace System::Windows::Forms;

try
{
    File^ f = gcnew File("test.txt");
    File::Stream^ stream = gcnew File::Stream(f, FileMode::Open, FileAccess::Read);
}

finally
{
    if (stream) stream->Close();
}
```

```cpp
using namespace System;
using namespace System::Windows::Forms;

try
{
    File^ f = gcnew File("test.txt");
    File::Stream^ stream = gcnew File::Stream(f, FileMode::Open, FileAccess::Read);
}

finally
{
    if (stream) stream->Close();
}
```

```cpp
using namespace System;
using namespace System::Windows::Forms;

try
{
    File^ f = gcnew File("test.txt");
    File::Stream^ stream = gcnew File::Stream(f, FileMode::Open, FileAccess::Read);
}

finally
{
    if (stream) stream->Close();
}
```
using namespace System;
using namespace System::Windows::Forms;
using namespace System::IO;

int main()
{
    FileStream^ file = nullptr;
    FileInfo^ fileinfo = gcnew FileInfo("C:\file.txt");
    try {
        file = fileinfo->OpenWrite();
        file->WriteByte(0xF);
    } finally {
        // Check for null because OpenWrite might have failed.
        if (file != nullptr) {
            file->Close();
        }
    }
}

Question 1. Given the following code segment, what possible exception will it throw?

for (int i = -4; i <= 4; i++) {
    str += (5/i) + " ";
}

A. ArithmeticException
B. OverflowException
C. IndexOutOfRangeException
D. DivideByZeroException

2. Given the following code segment, what possible exception will it throw?

int value = 987654321;
int cube = value * value * value;
MessageBox::Show(Convert::ToInt16(cube)+"");

A. ArithmeticException
B. OverflowException
C. IndexOutOfRangeException
D. DivideByZeroException

3. Given the following code segment, what possible exception will it throw?

String^ value = gcnew String('a', Int32::MaxValue);

A. OutOfMemoryException
B. OverflowException
C. StackOverflowException
D. TypeInitializationException

4. Given the following code, which can display the error in a message box when exception happens?

    catch (NullReferenceException^ e) { .... }

A. e->MessageBox::Show();
B. MessageBox::Show(e);
5. Given the following code segment, what possible exception will it throw?

```csharp
array<int^>^ x = gcnew array<int^>(5);
for (int i=0; i<=7; i++) {
    x[i] = i * i;
}
```

A. ArithmeticException
B. OverflowException
C. IndexOutOfRangeException
D. DivideByZeroException

6. How many times will the following catch statement display the error message on screen?

```csharp
catch (NullReferenceException^ e) { throw; }
```

A. 1
B. 2
C. 3
D. 4

7. Which statement is correct?
A. A finally block enables you to clean up actions that are performed in a try block.
B. If present, the finally block executes last, after the try block and any matched catch block.
C. A finally block always runs, regardless of whether an exception is thrown or a catch block matching the exception type is found.
D. The NullReferenceException constant does not have any affect in this case.

8. Given the following code, which statement is incorrect?

```csharp
String^ str = "";
array<int^>^ x = gcnew array<int^>(5);
for (int i=0; i<=7; i++) {
    try {
        x[i] = i;
        str += x[i] + " ";
    } catch(IndexOutOfRangeException) {
        if (i>= x->Length) {
            continue;
        }
    }
}
MessageBox::Show(str);
```

A. The x array has 5 elements.
B. The for loop will executes 8 times.
C. The result is 0 1 2 3 4.
D. The x array does not contain any element.

9. When an ArgumentNullException exception is thrown, which exception will be caught?

```csharp
catch(IndexOutOfRangeException^ e) {
```
```cpp
MessageBox::Show("1" + e->ToString());
} catch (ArgumentNullException^ e) {
  MessageBox::Show("2" + e->ToString());
} catch (Exception^ e) {
  MessageBox::Show("3" + e->ToString());
}

A. MessageBox::Show("1" + e->ToString());
B. MessageBox::Show("2" + e->ToString());
C. MessageBox::Show("3" + e->ToString());
D. MessageBox::Show("4" + e->ToString());

10. The __ exception is the exception that is thrown when an arithmetic, casting, or conversion operation in a checked context results in an overflow.
   A. ArithmeticException
   B. OverflowException
   C. OutOfMemoryException
   D. StackOverflowException
Lab #14     Exception Handling

Learning Activity #1:
1. Launch the Developer Command Prompt (not the regular Command Prompt) and change to the C:\cis223 directory.
2. Use Notepad to create a new text file named lab14_1.cpp with the following contents:

```cpp
//incorrect
#define <System.dll>
#define <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    int value = 987654321;
    double cube = Math::Pow(value, 3);
    MessageBox::Show(Convert::ToInt16(cube) + "]");
}
```

3. Compile and test the program to observe the error message.
4. Re-open the source file and change the source code to the following:

```cpp
//solution
#define <System.dll>
#define <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ str = ""
    int value = 987654321;
    double cube = Math::Pow(value, 3);
    try
    {
        MessageBox::Show(Convert::ToInt16(cube) + "]"); // error
    }
    catch (OverflowException^ e)
    {
        str = e->ToString(); // pass the exception message to str
        str = nullptr; // nullify str to clear the memory
        delete str; // destroy str
        cube = Math::Pow((long) value, 3); // calculate again with better precision
        MessageBox::Show(cube.ToString());
    }
```

Visual C++ Programming – Penn Wu, PhD  423
5. Type `cl /clr lab14_1.cpp /link /subsystem:windows /ENTRY:main` and press [Enter] to compile.
   Test the program. A sample output looks:

   ![Sample Output](image)

6. Download the “assignment template”, and rename it to `lab14.doc` if necessary. Capture a screen shot similar to the above figure and paste it to the Word document named `lab14.doc` (or `.docx`).

Learning Activity #2:

1. Use Notepad to create a new text file named `lab14_2.cpp` with the following contents:

   ```cpp
   //incorrect
   #using <System.dll>
   #using <System.Windows.Forms.dll>
   using namespace System;
   using namespace System::Windows::Forms;

   int main()
   {
       String^ str = "";

       for (int i = -4; i <= 4; i++)
       {
           str += (5/i) + " ";
       }

       MessageBox::Show(str);
   }
   ```

2. Compile and test the program to observe the error message.

3. Re-open the source file and change the source code to the following:

   ```cpp
   #using <System.dll>
   #using <System.Windows.Forms.dll>
    
   using namespace System;
   using namespace System::Windows::Forms;

   int main()
   {
       String^ str = "";
       String^ msg = "";

       for (int i = -4; i <= 4; i++)
       {
           try
           {
               str += (5/i) + " ";
           } 
           catch (Exception^ e) 
   ```
```cpp
{   
    msg = e->Message; // pass the exception message to msg
    continue; // when exception happen skip to next number
}
finally
{
    if (i==0)
    {
        str += " NaN ";
    }
}
}
MessageBox::Show(str);
}

4. Compile and test the program. A sample output looks:

![Sample output screenshot]

5. Capture a screen shot similar to the above figure and paste it to the Word document named lab14.doc (or .docx).

Learning Activity #3:
1. Use Notepad to create a new text file named lab14_3.cpp with the following contents:

   //Incorrect
   #using <System.dll>
   #using <System.Windows.Forms.dll>
   using namespace System;
   using namespace System::Windows::Forms;
   int main()
   {
       String^ str = "";
       array<int^>^ x = gcnew array<int^>(5);
       for (int i=0; i<=7; i++)
       {
           x[i] = i * i;
       }
       for (int i=0; i<x->Length; i++)
       {
           str += x[i] + " ";
       }
       MessageBox::Show(str);
   }

2. Compile and test the program to observe the error message.
3. Re-open the source file and change the source code to the following:

```csharp
#using <System.dll>
#using <System.Windows.Forms.dll>

using namespace System;
using namespace System::Windows::Forms;

int main()
{
    String^ str = "";
    String^ msg = "";

    array<int^>^ x = gcnew array<int^(5);

    for (int i=0; i<=7; i++)
    {
        try
        {
            x[i] = i * i;
        }
        catch(IndexOutOfRangeException^ e)
        {
            msg = e->Message;
            continue;
        }
    finally
    {
        if (i<x->Length) { // the Length property
            str += x[i] + " ";
        }
    }
    }

    str += "\nThe number of element is " + x->Length + "\n";

    for (int i=0; i<x->Length; i++)
    {
        str += x[i] + " ";
    }

    MessageBox::Show(str);
}
```

4. Compile and test the program. A sample output looks:

```
0 1 4 9 16
The number of element is 5
```

5. Capture a screen shot similar to the above figure and paste it to the Word document named lab14.doc (or .docx).

Learning Activity #4: OutOfMemoryException
1. Use Notepad to create a new text file named `lab14_4.cpp` with the following contents:

```cpp
#using <System.dll>
#using <System.Windows.Forms.dll>

class Program
{
    static void Main()
    {
        Array<int> LargeArray = gcnew Array<int>(987654321);
        for (int i=0; i<LargeArray.Length; i++)
        {
            LargeArray[i] = i;
        }
        MessageBox.Show("The last element is " + LargeArray[LargeArray.Length-1]);
    }
}
```

2. Compile and test the program to observe the error message.

3. Re-open the source file and change the source code to the following:

```cpp
#using <System.dll>
#using <System.Windows.Forms.dll>

using System;
using System::Windows::Forms;

int main()
{
    String^ msg = "";
    Array<int> LargeArray;

    try
    {
        LargeArray = gcnew Array<int>(987654321);
    }
    catch(OutOfMemoryException^ e)
    {
        msg = e->ToString();
        // set the Length to the largest possible value of an Int16.
        LargeArray = gcnew Array<int>(Int16::MaxValue);
    }

    for (int i=0; i<LargeArray.Length; i++)
    {
        LargeArray[i] = i;
    }
    MessageBox.Show("The last element is " + LargeArray[LargeArray.Length-1]);
}
```

4. Compile and test the program. A sample output looks:
1. Make sure the ‘x:\nippon’ directory DOES NOT exist (where ‘x’ indicates your system directory). Delete it if it exists.

2. Use Notepad to create a new text file named lab14_5.cpp with the following contents:

   ```cpp
   #using <System.dll>
   #using <System.Windows.Forms.dll>
   using namespace System;
   using namespace System::Windows::Forms;
   using namespace System::IO;
   int main()
   {
   String^ msg = "";
   String^ sDrive = System::Environment::GetEnvironmentVariable("homeDrive");
   Stream^ file = File::OpenWrite(sDrive + "\nippon\nagasaki.txt");
   int i = file->WriteByte(0xF);
   msg = "The nippon directory exists. No error."
   }
   ```

3. Compile and test the program to observe the error message.

4. Re-open the source file and change the source code to the following:

   ```cpp
   try
   {
   String^ msg = "";
   String^ sDrive = System::Environment::GetEnvironmentVariable("homeDrive");
   Stream^ file = File::OpenWrite(sDrive + "\nippon\nagasaki.txt");
   int i = file->WriteByte(0xF);
   msg = "The nippon directory exists. No error."
   }
   ```

5. Capture a screen shot similar to the above figure and paste it to the Word document named lab14.doc (or .docx).
catch (DirectoryNotFoundException^ e)
{
    msg = e->ToString();

    if (!Directory::Exists(sDrive + "\nippon"))
    {
        Directory::CreateDirectory(sDrive + "\nippon");
    }

    FileStream^ file = nullptr;
    FileInfo^ fi = gcnew FileInfo(sDrive + "\nippon\nagasaki.txt");

    msg = "No such directory or file. But, just created one. \n";
    msg += "Now check the '" + sDrive + "\nippon' to verify. \n";
}
finally
{
    // Check for null because OpenWrite might have failed.
    if (file != nullptr)
    {
        file->Close();
    }
    MessageBox::Show(msg);
}

5. Compile and test the program. The output looks:

   X:\nippon directory does not exist
   X:\nippon directory exist

   [Image of two screens showing program output]

6. Capture a screen shot similar to the above figure and paste it to the Word document named lab14.doc (or .docx).

Submittal
1. Complete all the 5 learning activities and the programming exercise in this lab.

2. Create a .zip file named lab14.zip containing ONLY the following self-executable files.
   - lab14_1.exe
   - lab14_2.exe
   - lab14_3.exe
   - lab14_4.exe
   - lab14_5.exe
   - lab14.doc (or lab14.docx or .pdf) [You may be given zero point if this Word document is missing]

3. Log in to course web site (e.g. Canvas or Blackboard), and enter the course site.

4. Upload the zipped file to Question 11 of Assignment as response.
Programming Exercise 14:
1. Launch the Developer Command Prompt.
2. Use Notepad to create a new text file named **ex14.cpp**.
3. Add the following heading lines (Be sure to use replace [YourFullNameHere] with the correct one).

   ```
   //File Name: ex14.cpp
   //Programmer: [YourFullNameHere]
   #using <System.dll>
   #using <System.Windows.Forms.dll>
   using namespace System;
   using namespace System::Windows::Forms;
   int main()
   {
   Object^ o = "0xb9d1";
   int i = (int) o;
   MessageBox::Show(o + " is converted to " + i);
   }
   ```

4. Try compile the above code now. You should receive the following message.

   ```
   Unhandled Exception: System.InvalidCastException: Specified cast is not valid.
   at main()
   at _mainCRTStartup()
   ```

5. Write a **try...catch** statement with an **InvalidCastException** exception to above code so the following message can be displayed without any interrupts.

6. Download the “programming exercise template”, and rename it to **ex14.doc** if necessary. **Copy your source code to the file** and then capture the screen shot(s) similar to the above figures and paste it to the Word document named “ex14.doc” (or .docx).


**Grading criteria:**
You will earn credit only when the following requirements are fulfilled. No partial credit is given.
- You successfully submit both source file and executable file.
- You must use try..catch structure to handle the **System.InvalidCastException** exception.
- Your program must meet all requirements.
- Your source code must be fully functional and may not contain syntax errors in order to earn credit.
- Your executable file (program) must be executable to earn credit.

**Threaded Discussion**
Note: Students are required to participate in the thread discussion on a weekly basis. Student must post at least two messages as responses to the question every week. Each message must be posted on a different date. Grading is based on quality of the message.

Question: Class, many exceptions or errors can be handled by a nested "if" structure. What is/are the advantage(s) or disadvantage(s) of using the try..catch structure in Visual C++. If you can, provide an example to support your perspective. [There is never a right-or-wrong answer for this question. Please free feel to express your opinion.]

Be sure to use proper college level of writing. Do not use texting language.