Lecture 12: Exception Handling

One of the things you are required to do as a responsible programmer is to ensure that your program allows only valid data to be accepted and permanently stored in data files. Your program should also anticipate and recover from unexpected circumstances. Exception handling helps in this area. This lecture focuses on the topic under the following captions:

- Introduction
- Throwing Exceptions
- Catching Exceptions
- Aborting a Program
- Handling Exception Thrown by the new Operator
- Summary and Concluding Remarks

12.1 Introduction

Inexperienced programmers usually think their program will always work as expected. On the other hand, experienced programmers know that things do not always work as expected. Smart programming is about taking care of the expected as well as the unexpected. Programmers refer to the unexpected situations as *exceptions*.

The following are some examples of scenarios that will cause C++ program errors (exceptions):
- The user enters a character where an integer is expected;
- The program uses an array subscript that is outside of the range of valid subscript values for a given array;
- An attempt is made at dividing by zero;
- An attempt is made to read from a file that does not exist.

Two broad categories of exceptions can be identified:
- Fatal errors are those errors that cause program or system failure.
- Integrity (non-fatal) errors are errors that affect the integrity of data manipulated by the program, but are not fatal.

Your program must effectively handle both types of errors. C++ provides the programmer with strategies for handling exceptions by *throwing* and *catching* exceptions.

12.2 Throwing Exceptions

When an error is detected within a function, you can *throw* a message or an expression forwarded (to some other section of your program where it will be *caught*). The syntax of the **throw** statement is as follows:

```
throw (<Object>);
```

Here are three important things you need to know about throwing exceptions:

1. The object thrown may be a variable, object (of a class), a numeric array, or a string.
   A function can throw several objects of differencing types.

2. If the exception is to be caught, then **throw** must be executed either from within a **try-block**, or from a function called within the **try-block**.

3. At least one statement within the **try-block** must either be a call of a function that issues a **throw**, or a **throw** statement.

12.2 Throwing Exceptions (continued)

A function that issues a **throw** statement must be invoked from a **try-block**. A **try-block** includes the following structure:

```
try
{
  // …  code to be executed
}
```

Figure 12-1 provides section of code that includes a **try-block** for reading in two variables that are to be validated. Figure 12-2 provides a corresponding listing of a function that reads in these variables. If the validation test is unsuccessful, an exception is thrown. The assumption here is that this function will be called from within a **try-block**, and will be subsequently caught.

Figure 12-1: Setting up a try-block

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <ctype.h>

const int REFERENCE = 1980000;

int main(int argc, char *argv[])
{
  int thisNumber;
  string thisName;
  // …
  try {
    inputData (thisNumber, thisName);
    // …
  }
  // … Rest of the program
}
```

Figure 12-2: Throwing an Exception from a Function

```
// Assume that this function is called from within a try-block
// Assume further that REFERENCE is a global constant with some predetermined value
void inputData (int &thisNumber, string &thisName)
{
  cout << ”Enter ID Number:”
  cin>> thisNumber; getchar();
  if (thisNumber < REFERENCE) throw (“Invalid IDNumber”);
  cout << endl;
  cout<<”Enter Name:”;
  cin >> thisName; char firstBytes[3] = (thisName.substr(0,2)).c_str();
  if (!isalpha(firstBytes[0])) throw (“Invalid Name entered…”);
}
```

12.3 Catching Exceptions

When an exception is thrown, the function that throws the exception immediately terminates and returns control to the calling statement. Failure to handle the exception will result in abnormal termination of your program. To handle a thrown exception, you must include at least one **catch-block** in your program. The **catch-block** must immediately follow a **try-block.** The **catch-block** outlines a graceful way to recover from the thrown exception. This often simply involves displaying a message to the user. It has the following syntax structure:

```
catch (<type> thisArg)
{
  // … catch-block containing instruction(s) to handle the exception
}
```

Here are some general guidelines:

1. Every thrown exception must be caught and processed by a **catch-block**.

2. There may (therefore) be several **catch-block** for a given try block.

3. The type of the exception determines which **catch** statement is used. The exception itself is caught (stored) in the object **thisArg**. Typically, this is a string, but it could be otherwise.

4. When your program is executed, if there are no exceptions, the try block(s), and catch block(s) are ignored.

5. After a **catch** statement executes, program control continues with the statement following the catch. When an exception has been thrown, control passes to the (corresponding) **catch** statement and the **try-block** terminates.

There may be occasions on which you want to catch all exceptions that might have been thrown. C++ provides a special catch-all usage of the catch statement. The format is shown below:

```
catch (…)
{
  // … Action to be taken if any exception is thrown
}
```

Putting this all together, figure 12-3a provides the basic syntactical structure for C++ exception handling. Here is the essence of the internal workings: A **try-block** executes code that throws at least one exception; for each exception thrown, there is a **catch-block** that handles recovery from the exception.

Figure 12-3a: Syntactical Structure of C++ Exception Handling

```
try {
// Code from which an exception may be raised; multiple exceptions may be raised
}
// . . .
catch (<Parameter for the exception>)
{
  // Exception-handling code
}
// . . .
catch (<Parameter for the exception>)
{
  // Exception-handling code; there is typically a catch-block for each exception raised in a preceding try-block
}
```

Figure 12-3b provides a partial program listing that illustrates how you may handle exceptions in an elegant manner. A global array called **errMsg** is used to store all error messages that the program may need to display. The function called **initialize()** sets up values for this array. The function called **inputData(…)** prompts the user for an ID number and a name. If either is invalid, an integer, which is the actual index to the appropriate message in **errMsg** is thrown. In the **main(…)** function, **inputData(…)** is called, and if an exception is raised, the appropriate message is retrieved from **errMsg** and displayed for the user. This example outlines an approach to exception handling that you may use in any program.

Figure 12-3b: Using try-block and catch-block

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <ctype.h>

const int REFERENCE = 1980000;
string errMsg[10]; // This array contains all error messages displayed by the program

int main(int argc, char *argv[])
{
   int thisNumber;
   string thisName;
   initialize(); // Initialize the message array called errMsg
   // …
   try {
     inputData (thisNumber, thisName);
     // …
   }
   catch (int errNum)
  {
    cout  << errMsg[errNum] << end;
   }

   // … Rest of the program
}  // End of main
//…
// Other Functions
void inputData (int &thisNumber, string &thisName) // The inputData Function
{
   cout << ”Enter ID Number:”
   cin>> thisNumber; getchar();
   if (thisNumber < REFERENCE) throw (0);
   cout << endl;
   cout<<”Enter Name:”;
   cin >> thisName; char firstBytes[3] = (thisName.substr(0,2)).c_str();
   if (!isalpha(firstBytes[0])) throw (1);
   // …
}
void initialize () // Initialization Method
{
   // Initialize errMsg array…
   errMsg[0] = “Invalid ID Number”;
   errMsg[1] = “Invalid Name”;
   errMsg[2] = “Invalid Date of Birth”;
  // …
} // End of Initialization Method
```

12.4 Aborting a Program

If an exception occurs from which your program cannot recover, C++ provides two functions to force immediate program termination, they are **exit ()** and **abort ()**
The abort () function requires no argument and returns no information. It simply forces an immediate program termination. It should be used as a last resort.

The **exit ()** function has prototype

```
void exit (int status);
```

Like **abort** (), it forces immediate program termination. However, the value of the status code is returned as an exit code to the called process. By convention, a value 0 indicates a successful termination; any other value indicates on abnormal termination. Typically, **exit ()** is called with an arbitrary integer value. Both functions belong to the heading file **<cstdlib.h>** header file.

12.5 Handling Exception Thrown by the new Operator

The **new** operator was introduced earlier in the course. As you are aware, it is useful when allocating memory for pointers. The **new** operator may return NULL, in which case an exception called the *bad-alloc* (short for bad allocation) exception is raised. It is therefore a good habit to use a **try-block** followed by a **catch-block** when using the **new** operator. Figure 12-4 provides a simple example.

Figure 12-4: Checking for Exception Caused by the new Operator

```
int main(int argc, char *argv[])
{
  char* testString;
  // …
  try
  {
    testString  = new char[30]; //allocate memory for 30-byte string
  }
  catch (bad_alloc xa)
  {
    cout « "Allocation failure. Please try later or change your program.\n";
    return 1;
  }
  // ….

return 0;
}
```

12.6 Summary and Concluding Remarks

In short, exception handling involves throwing and catching exceptions. When an exception is thrown, the function that throws it terminates, and control is returned to the statement responsible for invoking the function in the first place. Recovery from the exception must therefore be done in the function responsible for calling the function that throws the exception. This is done via a **try-block** followed by a **catch-block**. If the exception is to be caught, then **throw** must be executed either from within a **try-block**, or from a function called within the **try-block**.

As in Java, C++ allows you to catch an exception and re-throw it for any number of invocation levels. However, eventually, you must code an appropriate recovery from the exception.

C++ gives you the flexibility of throwing any valid object when an exception is raised. It also provides the flexibility of checking for any exception that might have been thrown.

12.7 Recommended Readings

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapter 16.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 18.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 16.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 9.