# 1.2 Exception Handling

Sam has just finished a large project and, at his professor's insistence, needs to retrofit error handling to his code. This is proving to be much more difficult than he envisioned. It is one thing to catch an error in a function, it is another thing entirely to let the caller of that function know what happened. This is forcing Sam to rethink his entire program! If only he thought about error handling from the beginning.

### Objectives

By the end of this chapter, you will be able to:

- List and define the components to C++ exceptions
- Write the code necessary to throw and catch an exception within a single function
- Write the code necessary to throw an exception in one function and catch it in another

### Prerequisites

Before reading this chapter, please make sure you are able to:

- Be able to create and use a #define (Procedural Programming in C++, Chapter 2.1)
- Author the Error Handling section of a design document (Chapter 1.0)
- Write the code necessary to detect a file, user and internal error (Chapter 1.1)



## What are exceptions and why you should care

In all but the simplest problems, programs are expected to gracefully recover from errors originating from a wide variety of sources. Some errors may come from the user such as improperly formed input, some may come from the system such as resources no longer being accessible, and some may come from the program itself when a logical error is encountered. In each of these cases, the user expects the program to continue to function normally.

There are three main ways that errors are handled in a program: error flags, error IDs, and exception handling. Of these three, exception handling is the only mechanism built into C++ specifically to facilitate the handling of errors. Exception handling affords the programmer an easy, efficient, and standard way to work with errors. Though the syntax may initially appear awkward and unusual, it serves to clearly delineate the part of a program dedicated to normal operation and the part tasked with recovering from errors.

**Sue's Tips**

Understanding exception handling is a required skill for all programmers. Though you will certainly encounter it many times in your career, there are other ways to handle errors. However, the more tools you have in your tool-bag of programmer tricks, the more effective you will be. This is one more thing that makes our job as programmers that much easier.

# Error flags

Possibly the easiest way to propagate an error is to have a function return a `bool`: `true` means the function succeeded in performing its task and `false` means that it failed.

```
bool readBoard(const string & fileName, char board[][3], bool & xTurn);
```

This methodology is called an "error flag." In the context of programming, a flag is a Boolean variable that contains two-state data. In many ways, this is similar to a signal flag used to communicate between ships in years past.

Back to our `readBoard()` example, consider the following code:

```
bool readBoard(const string & fileName, char board[][3], bool & xTurn)
{
   assert(fileName.length() != 0);

   // open the file
   ifstream fin(fileName.c_str());
   if (fin.fail())                        // always check for errors when
   {                                      //   opening a file
      cout << "Unable to open file "
           << fileName << " for reading.\n";
      return false;
   }

... code removed for brevity ...

   // determine whose turn it is by the xOverO count
   xTurn = (numXoverO == 0 ? true : false);
   if (numXoverO != 0 && numXoverO != 1)  // there must be an equal number of
   {
      cout << "Invalid board in file "    //    X's and O's or there must be
           << fileName << ".\n";          //    one more X.
      fin.close();
      return false;
   }

   // close the file
   fin.close();
   return true;
}
```

In this example, the function returns `false` if one of many conditions are met: we failed to open the file, we failed to read a number from the file, or if the board is invalid. Only if everything goes as expected does the function return `true`.

There are several problems with error flags. The first is that we only know that an error has occurred. No information is passed back to the caller indicating what the error was. Thus the caller is forced to treat all errors the same.

The second problem is that it is up to the caller to handle the error. If the caller cannot recover from the error (by re-prompting the user for a new file name in this example), then the caller will be forced to return `false` as-well. In other words, the caller must propagate the error. This means the error information must be passed down to the caller's caller if it expected the function to succeed.

The complete solution is available at 1-2-ticTacToe-flags.html or:

```
/home/cs165/examples/1-2-ticTacToe-flags.cpp
```

# Error ID

An error-ID (EID) works much the same as an error flag with one important difference: EIDs are integers and are able to differentiate between a wide number of errors. Consider the following `#defines`.

```
#define EID_NONE      0
#define EID_NO_FILE   1
#define EID_CORRUPT   2
#define EID_INVALID   3
```

With these codes, we can re-write our file reading function from the previous example:

```
int readBoard(const string & fileName, char board[][3], bool & xTurn)
{
   // open the file
   ifstream fin(fileName.c_str());
   if (fin.fail())
      return EID_NO_FILE;                 // send NO-FILE error
... code removed for brevity ...

   if (numXoverO != 0 && numXoverO != 1)
   {
      fin.close();                        // send INVALID error because
      return EID_INVALID;                 //    the game does not make sense
   }

   // close the file
   fin.close();
   return EID_NONE;                       // success!
}
```

The complete solution is available at 1-2-ticTacToe-eid.html or:

```
/home/cs165/examples/1-2-ticTacToe-eid.cpp
```

Observe how much more detailed the error message is with this example than with the flag example. This detail also yields an increase in complexity: both the caller and the callee must share the same EID vocabulary. That complexity is present in the function `interact()` which calls `readBoard()`:

```
switch(readBoard(fileName, board, xTurn))
{
   case EID_NO_FILE:
      cout << "Error: Unable to open file "
           << fileName << ".\n";
      break;
   case EID_CORRUPT:
      cout << "Error: The file "
           << fileName << " contains unexpected data.\n";
      clearBoard(board, xTurn);
      break;
   case EID_INVALID:
      cout << "Error: The file "
           << fileName << " does not contain a valid game.\n";
      clearBoard(board, xTurn);
}
```

In other words, an additional EID used by the callee `readBoard()` must be understood by `interact()`. This is a form of control coupling. A second problem with the EID method is that, like with the error flags, it is the responsibility of the caller to either handle the error or propagate it. This can be a source of programming errors if not done correctly.

# Exception handling

With the inherit shortcomings of the error flag method and the EID method of error handling, C++ developed a specialized mechanism to work with errors: exceptions. An **exception** is mechanism to alter the flow of a program in the case of a problem to special-purpose error-handling code. All C++ exceptions have the same three components: the `throw`, the `try`, and the `catch`.

*An exception is a special error-handling mechanism built into the C++ language*

## Throw

When an error is detected, an exception is thrown. If, for example, I wish to handle the case where an array variable consists of a `NULL` pointer, I could throw a c-string indicating the error state:

```
{
   // check if the array is invalid
   if (array == NULL)
      throw "Invalid pointer address";             // the exception is thrown from here

   // use the array variable now that it is safe
   array[0] = 42;
}
```

In this example, the c-string `"Invalid pointer address"` will be thrown if the pointer `array` consists of the `NULL` address. This means that the code after the `throw` statement (`array[0] = 42;` in this case) will not be executed. The syntax of the throw statement is:

```
throw ("down");
```

Expression to be thrown.
Must match the data type in the `catch` statement.

It is possible to throw any data type you like, though it is most common to throw a c-string, a `string` object, or an integer. It is also possible to throw more than one type of exception in a given function. For example, one statement could throw an `int` and another a `float`.

## Try

The second part of exception handling is the `try` statement. The purpose of the `try` statement is to delineate the part of the code where an exception could be thrown. Since the `try` statement combined with the curly braces delineate a block of code wherein an exception may be thrown, we most commonly call the region a try-block. Back to our example of the `NULL` pointer detection, the code may be:

```
try                                                 // the try block indicates that an
{                                                   //     exception may be thrown here
   // check if the array is invalid
   if (array == NULL)
      throw "Invalid pointer address";

   // use the array variable now that it is safe
   array[0] = 42;
}
```

### Sam's Corner

Exception handling code is heavily optimized for the non-error case. This means that you pay very little performance penalty for using exceptions in the case when it is not thrown. It also means that a thrown exception is rather expensive. C++ exceptions are designed for exceptional circumstances, not main-stream ones. They should only be used in the case of an unusual error.

## Catch

The final part of the exception handling mechanism is the catch statement. The purpose of `catch` is to receive the error message sent by `throw`. In other words, execution of the program jumps from `throw` to `catch` in the case where an exception is thrown. Thus, in the following example, if `array == NULL`, the statement `array[0] = 42;` will be skipped when the `"Invalid pointer address"` exception is thrown.

```
{
   try
   {
      // check if the array is invalid
      if (array == NULL)
         throw "Invalid pointer address";

      // use the array variable now that it is safe
      array[0] = 42;
   }
   catch (const char * message)                    // in the case of an error, the
   {                                               //     program will jump from the
      cout << "Error: " << message << endl;        //     throw to this code
   }
}
```

The data type in the `catch` statement must match the data type in the `throw` statement for the thrown exception to be caught. In the above example, a c-string constant is thrown and the variable `message` in the catch statement is of the same data type.

In the case when more than one data type is thrown, multiple `catch` statements may be required. Consider the following code:

```
{
   try
   {
      if (text == NULL)
         throw string("NULL pointer in text variable");   // throw a string object
      if (text[0] == '\0')
         throw 0;                                         // throw an integer
      cout << text << endl;
   }
   catch (const string message)                           // catch the string
   {
      cout << message << endl;
   }
   catch (int value)                                      // catch the integer
   {
      cout << "text contains an empty string!\n";
   }
}
```

With two types of exceptions thrown (a `string` object and an integer), two catches are required. There is also a special type of catch statement akin to the `default` statement in a SWITCH-CASE. This `catch` statement will catch any type exception, regardless of the data type. This is called the catch-all:

```
catch (...)                          // catch-all
{
   cout << "Some exception was thrown, I just don't know which one!\n";
}
```

Note that exception handling may occur within a function or between functions. Although the mechanisms are similar, there are subtle differences between these cases.

# Exceptions within a function

One kind of exception handling occurs when an exception is thrown and caught in the same function. Back to the `readBoard()` example used earlier:

```
void readBoard(const string & fileName, char board[][3], bool & xTurn)
{
   assert(fileName.length() != 0);

   try
   {
      // open the file
      ifstream fin(fileName.c_str());
      if (fin.fail())                              // throw if we fail to open
         throw "Unable to open file";              //    the file for any reason

      // read the contents of the file
      int numXoverO = 0;
      for (int row = 0; row < 3; row++)
         for (int col = 0; col < 3; col++)
         {
            // read the item from the board
            fin >> board[row][col];

            // check for failure.
            if (fin.fail())                        // throw if we fail to read data
               throw "Error reading file ";        //    from the file

            if (board[row][col] == 'X')
               numXoverO++;
            else if (board[row][col] == 'O')
               numXoverO--;
            else if (board[row][col] != '.')       // throw if the symbol we read
               throw "Unexpected symbol";          //    is not a X O or .
         }

      // determine whose turn it is by the xOverO count
      xTurn = (numXoverO == 0 ? true : false);
      if (numXoverO != 0 && numXoverO != 1)        // throw if the board does not
         throw "Invalid board";                    //    make sense
   }
   catch (const char * message)                    // all errors are sent here
   {                                               //    so we only need to
      cout << "Error: "   << message               //    construct the error
           << " in file " << fileName              //    message once
           << ".\n";
   }

   // close the file
   fin.close();                                    // this is called regardless of
}                                                  //    whether there was an error
```

In this example, no code in the `catch` block will get executed if an exception is not thrown. However, if one of the four exceptions is thrown, control of the program will be redirected to the `catch` block immediately.

The complete solution is available at 1-2-ticTacToe-exception1.html or:

```
/home/cs165/examples/1-2-ticTacToe-exception1.cpp
```

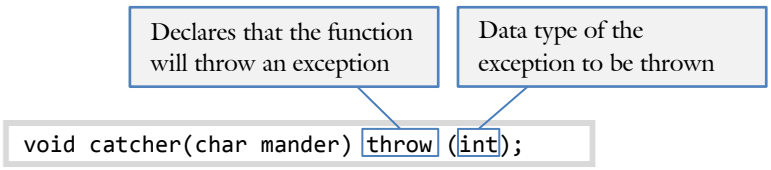
## Exceptions between functions

It is common to want to throw an exception in one function and catch it in another. To accomplish this, three things are needed. First, the function throwing the exception needs to advertise the types of exceptions that are thrown in a **throw list**. Thus the prototype of `readBoard()` becomes:

*A throw list is a list of all the possible un-caught exceptions a function may throw*

```
void readBoard(const string & fileName,
               char board[][3],
               bool & xTurn) throw (const char *);
```

The syntax for a function prototype including the throw list is:




The second part of the syntax is to throw the exception in the function itself. Because `readBoard()` will not be catching its own exception, there is no `try`-block and there is no `catch`-block.

```
void readBoard(const string & fileName,
               char board[][3],
               bool & xTurn) throw (const char *)
{
   // open the file
   ifstream fin(fileName.c_str());
   if (fin.fail())                          // throw if we fail to open
      throw "Unable to open file";          //    the file for any reason
... code removed for brevity ...
   if (numXoverO != 0 && numXoverO != 1)    // throw if the board does not
      throw "Invalid board";                //    make sense

   // close the file
   fin.close();                             // this is called regardless of
}
```

The final part of the syntax is to catch the exception in the caller. In this case, the caller is `interact()`:

```
... code removed for brevity ...
         case 'r':
            fileName = getText("What file would you like to read the board from? ");
            if (fileName.length() != 0)
            {
               try
               {
                  readBoard(fileName, board, xTurn);
                  displayBoard(board);
               }
               catch (const char * message)         // catch the exception
               {                                    //    thrown in the
                  cout << "Error: "   << message     //    readBoard() function
                       << " in file " << fileName
                       << ".\n";
               }
            }
            break;
... code removed for brevity ...
```

The complete solution is available at 1-2-ticTacToe-exception2.html or:

```
/home/cs165/examples/1-2-ticTacToe-exception2.cpp
```

## Example 1.2 – Multiple exceptions

**Demo**

The purpose of this demo is three-fold. First, it is meant to demonstrate how multiple exception types can be thrown in a single function. Next, the exceptions are to be caught in a different function than they are thrown from. Finally, a catch-all will demonstrate how to catch un-expected exceptions.

**Solution**

The function throwing the exceptions:

```
void exceptionalFunction() throw (int, float, string, char)
{
   Switch (prompt())
   {
      case 1:
         throw 0;
      case 2:
         throw float(0.0);
      case 3:
         throw string("zero");
      case 4:
         break;
      default:
         throw '0';
   }
   cout << "End of the exceptional function\n";
}
```

Next, `main()` which will catch the exceptions:

```
int main()
{
   try
   {
      exceptionalFunction();
      cout << "No exception was thrown\n";
   }
   catch (int integer)                                  // for throw 0;
   {
      cout << "An integer was thrown!\n";
   }
   catch (float floatingPoint)                          // for throw float(0.0);
   {
      cout << "A floating point number was thrown!\n";
   }
   catch (string text)                                  // for throw string("zero");
   {
      cout << "Text was thrown!\n";
   }
   catch (...)                                          // catch-all
   {
      cout << "Error! Unexpected exception was thrown!\n";
   }
   return 0;
}
```

**See Also**

The complete solution is available at 1-2-multipleThrows.html or:

```
/home/cs165/examples/1-2-multipleThrows.cpp
```

## Review 1

Write an assert to verify that the piece in a chess board (`board[row][col]`) is one of the following: `{K, Q, R, B, N, P, space}`

*Please see page* 50 *for a hint.*

## Review 2

Modify the above code from Review 1 so the entire board is checked to make sure it is one of the valid pieces.

*Please see page* 50 *for a hint.*

## Problem 3

What exceptions are thrown from the following function?

```
double funky(string text) throw (int, bool)
```

*Please see page* 61 *for a hint.*

## Problem 4

What is the most correct function definition for the following code?

```
{
   cout << "Which item would you like to edit? "
   int index;
   cin >> index;

   if (cin.error())
      throw "Non-digit entered";
   if (index < 0)
      throw index;
   return index;
}
```

*Please see page* 61 *for a hint.*

## Problem 5

What is the output when the user enters the value 0?

```
try
{
   int value;
   cin >> value;

   if (value < 0)
      throw "Negative!";
   if (value == 0)
      throw 0;
}
catch (char *message)
{
   cout << message << endl;
}
catch (...)
{
   cout << "Unhandled exception!";
}
catch (int num)
{
   cout << num;
}
```

*Please see page* 59 *for a hint.*

## Problem 6

Up to this point, we call the new function to allocate memory with the following code:

```
double * list = new(no_throw) double[10];
```

It turns out that `new` throws the `bad_alloc` exception. Write some code to allocate 10 `doubles` and display an error message if the allocation failed by catching the `bad_alloc` exception.

```
{

      double * list = new double[10];


}
```

*Please see page* 60 *for a hint.*

## Problem 7

Match the scenario on the left with the best error handling technique on the right

| Scenario | Technique |
|---|---|
| User responds to a prompt with data that is outside the expected range | Throw an exception |
| The file you are attempting to read from does not exist | Use an assert |
| The code depends on a pointer being initialized, but you want to check it just to make sure | While loop and try again |
| The user enters a file-name that has illegal characters | Display an error message and re-prompt |
| In a grades program, the user entered -100% as an earned grade | Exit the program immediately |

*Please see page* 39 *for a hint.*