# 1.1 Defensive Programming

Sue is working on a design document for a project and is stumped by the "Error Handling" section. While she understands the importance of properly handling all types of errors to her program, she is unsure exactly how that is to work. Sue needs a set of tools to help her identify and deal with file, user, and internal errors.

### Objectives

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

- Use asserts to catch internal errors
- Catch user errors with `cin.fail()`
- Recover from user errors with `cin.ignore()` and `cin.clear()`

### Prerequisites

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

- Describe the purpose of the Error Handling section of a Design Document (chapter 1.0)
- Make logical assertions use Boolean algebra (Procedural Programming in C++, chapter 1.5)
- Accept user input with `cin` (Procedural Programming in C++, chapter 1.2)
- Be able to use the `string` class (Procedural Programming in C++, chapter 4.2)



## What is defensive programming and why you should care

Defensive driving is the process of driving your car in such a way to be constantly ready to handle the worst possible conditions. This is an important skill for keeping the occupants of your car safe at all points in time. Defensive programming is much the same way. This is about writing code so there is "no way possible" for the program to malfunction, regardless of the input of the user or other system faults that may occur. While this is clearly an impossible goal, the closer we get the more reliable our code becomes. A well written program should never malfunction regardless of the environment it is put in. It should resist failure to any combination of user input, be that malicious, ignorant, or accidental. When software falls short of this goal, the following eventualities may result:

- **Instability**: The user may perceive the software to be unreliable and untrustworthy.
- **Incompatibility**: The software may work on less systems than the user needs.
- **Insecurity**: The software may be vulnerable to malicious attacks compromising the user's system or his confidential data.

Each of these eventualities is clearly undesirable. Though the sources of these defects are legion, the most common include: file errors, user errors, and internal errors.

**Sue's Tips**

The earlier you start thinking about errors, the easier they will be to catch. This is why the Error Handling section exists in a design document. It is well worth your time to brainstorm about the most likely sources of errors in your program. In many cases, significant sources of errors can be mitigated in the design process long before actual code is written.

# File errors

When reading from a file, several things could happen causing a program to malfunction:

- **Missing file**: The requested file might not exist. It is common for the user to misspell a file name or specify a file that does not exist. A program should handle these errors gracefully.
- **Insufficient permissions**: The user might not have sufficient privileges to access the file.
- **Corrupt file**: The file may not be in the expected format.

In each case, checks must be in place to catch the errors. These are typically accomplished with extensive use of the `fail()` method. Back to our `readBoard()` function from the Tic-Tac-Toe example:

```
void readBoard(const string & fileName, char board[][3], bool & xTurn)
{
   // 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;                                   // see chapter 1.2 for ways to report
   }                                            //   these errors to the caller

   // 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())                        // a failure can also occur in the middle
         {                                      //    of reading a file.  Check them too!
            cout << "Error reading file "      // tell the user that the failure
                 << fileName << ".\n";          //    happened, and why!
            fin.close();                        // don't forget to close the file!
            return;                             // again, report these errors
         }

         if (board[row][col] == 'X')
            numXoverO++;
         else if (board[row][col] == 'O')
            numXoverO--;
         else if (board[row][col] != '.')       // make sure the file only consists
            cout << "Unexpected symbol "        //   of X, O, and . symbols.
                 << board[row][col]
                 << "in file: " << fileName << ".\n";
      }

   // 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.

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

Observe how the added code verifies that the file exists and the data in the file is in the expected format. We need to do all we can to detect every possible misconfiguration of the file.

# User errors

One must always assume that the user will enter the worst possible input to a given prompt. While most users are not malicious, ignorant or careless users often find ways to make programs malfunction. The two most common form of user errors are out-of-bounds errors, buffer overrun errors, and unexpected input.

# Bounds checking

A program prompting the user for a row on a Tic-Tac-Toe board expects the value to be between 1 and 3, and a column value to be between 'A' and 'C'. Thus, if the user enters a value outside that range, the user must be re-prompted. The process of verifying that user input is within a pre-defined range is called "bounds checking."

Back to our Tic-Tac-Toe example, one of the bugs with the original program is that the bounds checking is missing. To address this issue, the following code was added:

```
void getCoordinates(int & row, int & col)
{
   char colLetter;

   do
   {
      // prompt with instructions
      cout << "Please specify the coordinates: ";
      cin >> colLetter >> row;                    // an error may occur here too!
   }
   while (colLetter < 'A' || colLetter > 'C' ||
          row < 1 || row > 3);

   // convert for a letter to a number
   col = colLetter - 'A';

   // convert from 1's based to 0's based
   row--;
}
```

Observe how the added DO-WHILE loop serves to catch the case where user input is outside the valid range. This guarantees that the variable `row` and `col` are in the expected range.

**Sue's Tips**

As a general rule, every time the user is given the opportunity to input a number, the program should verify that the number is in the valid range before the value is used. It is therefore a very common pattern to use a DO-WHILE loop in a prompt function.

## Buffer overruns

A **buffer** is like an array, a block of memory set aside for a task. If the programmer sets aside ten slots of memory, what happens when eleven are used? What happens when the programmer uses memory that is not reserved?

*A buffer is an array of data to be filled with user input*

```
{
   int array[10];                  // ten integers

   for (int i = 0; i < 20; i++)    // what happens when twenty numbers are put
     array[i] = 0;                 //    into an array that can hold ten?
}
```

The answer to this question is: bad things will happen. It is akin to building an Olympic sized swimming pool on a small city lot: the pool will extend into your neighbor's land and he will not be happy! With a computer program, a buffer overrun like this will probably cause the program to crash.

Often, however, array bounds bugs are more difficult to find. Observe that c-strings are arrays of characters with a fixed length (say 256 characters). Thus the following code from our Tic-Tac-Toe game has a buffer overrun bug.

```
void getText(const char * prompt, char * input)
{
   cout << prompt;
   cin  >> input;                  // ERROR: could be longer than 255!
}
```

In this case, the assumption is that the user will not input more than 255 characters for his file name. While few individuals would enter such a long name, a malicious user may attempt an especially long string. This defect can be fixed by using the string class rather than c-strings.

```
string getText(const char * prompt)
{
   cout << prompt;
   string input;                  // notice that we do not specify the size here!
   cin >> input;                  // the string object "input" will grow to accommodate
   return input;                  //    as much text as the user provides
}
```

### Sam's Corner

The `string` class is able to handle any-sized input because of the way it stores data in its buffer. Initially, the buffer size of `text` is small. As the user enters more data into the buffer, the `string` object checks to make sure the buffer size is not exceeded. When it is exceeded, a new buffer twice the size is allocated and the old data is copied into the new data. This process is continued as long as more data is entered into the buffer. Thus, as long as there is enough memory in the computer, it is impossible to exceed the buffer size of a `string` object.

# Unexpected input

What happens when the user enters a non-digit into a variable that expects numbers? Consider, for example the following code:

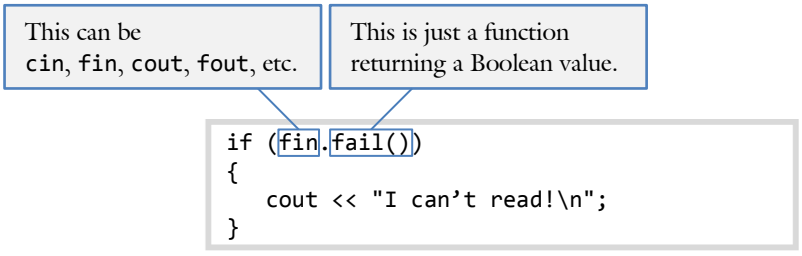
```
void getCoordinates(int & row, int & col)
{
   char colLetter;

   do
   {
      // prompt with instructions
      cout << "Please specify the coordinates: ";
      cin >> colLetter >> row;                    // ERROR! User could enter a letter
   }                                              //    for the row variable
   while (colLetter < 'A' || colLetter > 'C' ||
          row < 1 || row > 3);

   // convert to letter to a number
   col = colLetter - 'A';

   // convert from 1's based to 0's based
   row--;
}
```

What will be the output if the user enters a letter instead of a digit? To see an example of this, try compiling the program `/home/cs165/examples/1-0-ticTacToe.cpp` and enter "`zz`" as a coordinate.

The problem here is that `cin` enters an error state when a non-digit appears at the beginning of the input stream. We need to detect this state (for example when a letter is put into an integer), clear the state, and handle the error (skip the invalid data).

## Detecting errors

The first step to recovering from user input errors is to detect that an error has occurred. The stream variables (such as `cin`, `cout`, `fin`, and `fout`) contain a member variable representing the error state of the stream. For a file stream, these errors could include missing file, no permission to open a file, unable to write to a file, or a host of other sources. Each of these errors sets the error variable in the stream object. We can detect these errors with the `fail()` method:




Console streams also have error member variables that can be set due to a variety of error events. One of these events occurs when a non-integer is read into an integer variable. We can detect this class of errors with:

```
if (cin.fail())
   ...
```

## Clearing the error state

The final stage in handing stream errors is to clear the error state. Recall that each stream object as an error member variable. Once an error is encountered, the error member variable remains set until it is explicitly cleared. This can be accomplished through the `clear()` method:

```
cin.clear();
```

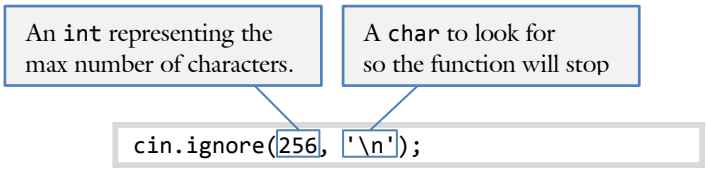
After this has been called, normal stream operations can continue and `cin.fail()` function will return `false`.

## Handling errors

Once it has been determined that the input stream contains at least one erroneous character, it is necessary to skip over them so normal user input can be accepted. If, for example, the program prompts the user for an integer and the user enters "five" into the input stream, the stream will look like:

| 'f' | 'i' | 'v' | 'e' | '\n' | |
|---|---|---|---|---|---|

From here, we need to skip over the text "`five`" and move the insertion stream pointer to the newline character. This can be accomplished through the `ignore()` method. By default, `ignore()` will skip over one character in the input stream. You can also configure `ignore()` to skip over any number of characters until a given character is encountered in the input stream. For example, the following code will skip up to 256 characters until a newline is encountered:




## Using cin.fail(), cin.clear(), and cin.ignore()

In order to make our `getCoordinates()` function correctly handle errors, we need to carefully incorporate error detection (`cin.fail()`), error recovery (`cin.ignore()`), and clearing the error state (`cin.clear()`). To bring it all together:

```
void getCoordinates(int & row, int & col)
{
   char colLetter;

   do
   {
      // prompt with instructions
      cout << "Please specify the coordinates: ";
      cin >> colLetter >> row;

      if (cin.fail())                    // detect we are in an error state
      {
         cin.clear();                    // clear the error state
         cin.ignore(256, '\n');          // ignore the rest of the characters
      }
   }
   while (colLetter < 'A' || colLetter > 'C' ||
          row < 1 || row > 3);

   // convert to a letter to a number
   col = colLetter - 'A';

   // convert from 1's based to 0's based
   row--;
}
```

# Internal errors

When writing a program, we often make a ton of assumptions. We assume that a function was able to perform its task correctly; we assume the parameters in a function are set up correctly; and we assume our data-structures are correctly configured. A diligent programmer would check all these assumptions to make sure his code is robust. Unfortunately, the vast majority of these checks are redundant and, to make matters worse, can be a drain on performance. A method is needed to allow a programmer to state all his assumptions, get notified when the assumptions are violated, and have the checks not influence the speed or stability of the customer's program. Assertions are designed to fill this need.

## Overview of asserts

An **assert** is a check placed in a program representing an assumption the developer thinks is always true. In other words, the developer does not believe the assumption will ever be proven false and, if it does, definitely wants to be notified. An assert is expressed as a Boolean expression where the true evaluation indicates the assumption proved correct and the false assumption indicates violation of the assumption. Asserts are evaluated at run-time verifying the integrity of an assumption with each execution of the check.

*An assert is a check placed in a program representing an assumption the developer thinks is always true*

An assert is said to fire when, during the execution of the program, the Boolean expression evaluates to false. In most cases, the firing of an assert triggers termination of the program. Typically the assert will tell the programmer where the assert is located (program name, file name, function, and line number) as well as the assumption that was violated.

Assertions have several purposes:

- **Identify logical errors**: While writing a program, assertions can be helpful for the developer to identify errors in the program due to invalid assumptions. Though many of these can be found through more thorough investigation of the algorithm, the use of assertions can be a time saver.
- **Find special-case bugs**: Testers can help find assumption violations while testing the product when their copy of the software has embedded asserts. Typically, developers love this class of bugs because the assert will tell the developer where to start looking for the cause of the bug.
- **Highlight integration issues**: During component integration activities or when enhancements are being made, well-constructed assertions can help prevent problems and speed development time.

Assertions are not designed for:

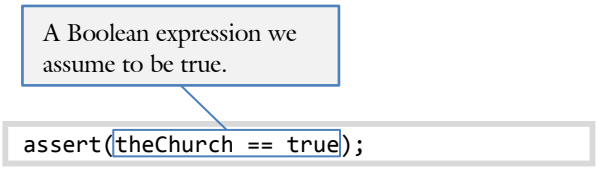
- **User error**: The user should never see an assert fire. Asserts are designed to detect internal errors, not invalid input provided by the user.
- **File errors**: Like user-errors, a program must gracefully recover from file errors without asserts firing.

## Syntax

Asserts in C++ are in the `cassert` library. You can include asserts with:

```
#include <cassert>
```

Since asserts are simply C++ statements (more precisely, they are function calls), they can be put in just about any location in the code. An example `assert` ensuring an index is not negative would be:




If this assert were in a file called `testimony.cpp` as part of a program called grades in the function called `templeInterview`, then the following output would appear if the assumption proved to be invalid because the Boolean variable was set incorrectly:

```
interview: testimony.cpp:164: bool templeInterview(std:: string&): Assertion `theChurch
== true' failed.
Aborted
```

It is important that the client never sees a build of the product containing asserts. Fortunately, it is easy to remove all the asserts in a product by defining the `NDEBUG` macro. Since asserts are defined with pre-processor directives, the `NDEBUG` macro will effectively remove all assert code from the product at compilation time. This can be achieved with the following compiler switch:

```
g++ -DNDEBUG file.cpp
```

## When to use asserts

As a general rule, an assert should be added every time an assumption is made in the code. Typically, a program should consist of about 50% asserts. Common places to put asserts include: at the beginning of functions, before a function is called, and after a function is called.

### Beginning of a function

Asserts should be placed at the beginning of a function to verify that the passed parameters are valid. If the assert fires, the problem is not to be found in the function, but rather in the caller. This is very useful information for debugging a program. For example, the `convertSymbol()` function in the Tic-Tac-Toe function will convert `'.'` into `' '` for display purposes. The code with an assert would be:

```
char convertSymbol(char letter)
{
   assert(letter == '.' || letter == 'O' || letter == 'X');
   return (letter == '.') ? ' ' : letter;
}
```

## End of a function

Asserts should be placed at the end of a function to varify that the data passed back to the caller is valid. If the assert fires, it is not the caller's fault but the function's fault. For example, the `editSquare()` function needs place an X or O in a square. We can add an assert to verify this:

```
void editSquare(char board[][3], bool & xTurn)
{
   int row;
   int col;

   // get the coordinates
   getCoordinates(row, col);
   if (board[row][col] != '.')
      return;

   // change the state of the board
   board[row][col] = (xTurn ? 'X' : 'O');
   xTurn = !xTurn;

   assert(board[row][col] == 'X' || board[row][col] == 'O');
}
```

Note that the `assert` in the above example may seem redundant. After all, we just set `board[row][col]` to a valid value! However, what would happen if the function was changed by another programmer who does not fully understand the `board[][]` data-structure? This `assert` is not meant to catch bugs that currently exist in the function, but rather bugs that may be introduced in the future.

## After a function is called

When a function is called, the caller needs an assurance that the provided data was as expected. It is therefore common to the return values of a function. Back to the `editSquare()` function:

```
void editSquare(char board[][3], bool & xTurn)
{
   int row;
   int col;

   // get the coordinates
   getCoordinates(row, col);
   assert(row >= 0 && row <= 2);
   assert(col >= 0 && col <= 2);
   if (board[row][col] != '.')
      return;

   // change the state of the board
   board[row][col] = (xTurn ? 'X' : 'O');
   xTurn = !xTurn;

   assert(board[row][col] == 'X' || board[row][col] == 'O');
}
```

## Other instances

There are many other instances that an `assert` may come in handy. This includes any time an assumption is made in a program. Common examples are: valid pointers (Assume that a passed pointer is not `NULL` before dereferencing it), valid indices (assume that the index to an array is within an acceptable range), and well-formed data (assume that the data (say our Tic-Tac-Toe board) is in the expected format). A programmer should never hesitate to add an `assert`. Even situations where the assumption "could not possibly be violated" can prove to be incorrect.

## Example 1.1 – Get Index

**Demo**

This program will demonstrate how to fully integrate asserts and user error handling into a function that prompts the user for an index.

**Problem**

Write a program to prompt the user for an index and display the results. The program should function normally regardless of the input the user enters.

```
Please enter the index.  The acceptable range is 1 <= index <= 10.
> 0
ERROR: value is outside the accepted range
> five
ERROR: non-digit entered
> 5
The user's index is: 5
```

**Solution**

```
int getIndex(int min, int max)
{
   // before we do anything, validate the input.
   assert(min <= max);
   bool done = false;
   int index = min - 1;  // some invalid state.

   // instructions
   cout << "Please enter the index.  The acceptable range is "
        << min << " <= index <= " << max << ".\n";

   do
   {
      // we should be all clear at this point
      assert(cin.good());
      cout << "> ";
      cin  >> index;

      // check for a value that is not an integer
      if (cin.fail())
      {
         cout << "ERROR: non-digit entered\n";
         cin.clear();           // clear the error state
         cin.ignore(256, '\n'); // ignore all the characters in the buffer
      }

      // check we are within range
      else if (index < min || index > max)
         cout << "ERROR: value is outside the accepted range\n";
      else
         done = true;
   }
   while (!done);

   // ensure we are good before we even think of leaving
   assert(index >= min && index <= max);
   return index;
}
```

**See Also**

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

```
/home/cs165/examples/1-1-getIndex.cpp
```

## Example 1.1 – Tic-Tac-Toe

**Demo**

This program will demonstrate how to fully integrate asserts and user error handling into project. Specifically, the Tic-Tac-Toe project from Chapter 1.0

**Solution**

All three types of errors are handled in Tic-Tac-Toe:

### File Errors

Please see `readBoard()`. Observe how failure-to-open failures are handed as well as invalid data in the board. All the conditions of the Error Handling section of the design document are checked:

| Error | Condition | Handling |
|---|---|---|
| Unexpected token | Token in file that is not `X`, `O`, or `.` | Error message and stop loading file |
| Too few tokens | Reach EOF before 9 tokens are read | Error message and stop loading file |
| Impossible game | There should be the same number of `X`s and `O`s or one more `X` than `O` | Error message and stop loading file |

### User Errors

Users errors are checked in the `interact()` function for the times when the user specified an invalid option, `getCoordinates()` when the user is prompted to change the game, and in the file functions when the user specifies a filename. Thus all the conditions of the Error Handing section of the design document are checked.

| Error | Condition | Handling |
|---|---|---|
| Invalid option | User input something other than `r`,`s`, or `q` | Re-prompt and error message |
| Invalid coordinate | Coordinate specification that is not `[a..c][1..3]` | Re-prompt |
| Invalid filename | Unable to open file with specified filename | Re-prompt and error message |

### Internal Errors

Seven asserts are placed in the project to ensure the filename is correctly formed, the row & column variables are valid, and that the board itself has the expected tokens in it. Thus all the conditions of the Error Handling section are checked:

| Error | Condition | Handling |
|---|---|---|
| Malformed coordinate | `coord.x > 2` or `coord.x < 0` or `coord.y > 2` or `coord.y < 0` | Assert |
| Malformed board | Any value in the board that is not `X`, `O`, or `.` | Assert |

**See Also**

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

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

## Review 1

List the three types of errors that frequently need to be described in the Error Handling section of the Design Document.

- _____
- _____
- _____

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

## Review 2

For each of the three error types listed in Review 1, how should the program handle these errors?

- _____
- _____
- _____

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



## Problem 3

Consider the following function, add asserts to catch as many bugs as possible:

```
bool isLeapYear(int year)
{
   if (year % 4)
      return false;

   if (year % 100)
      return true;

   if (year % 400)
      return false;

   return true;
}
```

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

## Problem 4

Write a function to prompt the user for his GPA. Make sure the function will always return valid data and will catch all errors. The stub function is:

```
float getGPA
{



}
```

*Please see page* 45 *for a hint.*