7.12 — Handling errors, cerr and exit

BY ALEX ON OCTOBER 22ND, 2007 | LAST MODIFIED BY ALEX ON MARCH 30TH, 2018

When writing programs, it is almost inevitable that you will make mistakes. In this section, we will talk about the different kinds of errors that are made, and how they are commonly handled.

Errors fall into two categories: syntax and semantic errors.

Syntax errors

A **syntax error** occurs when you write a statement that is not valid according to the grammar of the C++ language. For example:

```
if 5 is not equal to 6 then write "not equal";
```

Although this statement is understandable by humans, it is not valid according to C++ syntax. The correct C++ statement would be:

```
1 if (5 != 6)
2     std::cout << "not equal";
```

Syntax errors are almost always caught by the compiler and are usually easy to fix. Consequently, we typically don't worry about them too much.

Semantic errors

A **semantic error** occurs when a statement is syntactically valid, but does not do what the programmer intended. For example:

```
1 for (int count=0; count <= 3; ++count)
2     std::cout << count << " ";
```

The programmer may have intended this statement to print `0 1 2`, but it actually prints `0 1 2 3`.

Semantic errors are not caught by the compiler, and can have any number of effects: they may not show up at all, cause the program to produce the wrong output, cause erratic behavior, corrupt data, or cause the program to crash.

It is largely the semantic errors that we are concerned with.

Semantic errors can occur in a number of ways. One of the most common semantic errors is a logic error. A **logic error** occurs when the programmer incorrectly codes the logic of a statement. The above for statement example is a logic error. Here is another example:

```
1 if (x >= 5)
2     std::cout << "x is greater than 5";
```

What happens when x is exactly 5? The conditional expression evaluates to true, and the program prints "x is greater than 5". Logic errors can be easy or hard to locate, depending on the nature of the problem.

Another common semantic error is the violated assumption. A **violated assumption** occurs when the programmer assumes that something will be either true or false, and it isn't. For example:

```
1 std::string hello = "Hello, world!";
2 std::cout << "Enter an index: ";
3
4 int index;
5 std::cin >> index;
6
7 std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
```

See the potential problem here? The programmer has assumed that the user will enter a value between 0 and the length of "Hello, world!". If the user enters a negative number, or a large number, the array index will be out of bounds. In this case, since we are just reading a value, the program will probably print a garbage letter. But in other cases, the erroneous statement might corrupt other variables or cause the program to crash.

Defensive programming is a form of program design that involves trying to identify areas where assumptions may be violated, and writing code that detects and handles any violation of those assumptions so that the program reacts in a predictable way when those violations do occur.

Detecting assumption errors

As it turns out, we can catch almost all assumptions that need to be checked in one of three locations:

- When a function has been called, the caller may have passed the function parameters that are incorrect or semantically meaningless.
- When a function returns, the return value may indicate that an error has occurred.
- When program receives input (either from the user, or a file), the input may not be in the correct format.

Consequently, the following rules should be used when programming defensively:

- At the top of each function, check to make sure any parameters have appropriate values.
- After a function has returned, check the return value (if any), and any other error reporting mechanisms, to see if an error occurred.
- Validate any user input to make sure it meets the expected formatting or range criteria.

Let's take a look at examples of each of these.

Problem: When a function is called, the caller may have passed the function parameters that are semantically meaningless.

```
1  void printString(const char *cstring)
2  {
3      std::cout << cstring;
4  }
```

Can you identify the assumption that may be violated? The answer is that the caller might pass in a null pointer instead of a valid C-style string. If that happens, the program will crash. Here's the function again with code that checks to make sure the function parameter is non-null:

```
1  void printString(const char *cstring)
2  {
3      // Only print if cstring is non-null
4      if (cstring)
5          std::cout << cstring;
6  }
```

Problem: When a function returns, the return value may indicate that an error has occurred.

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello = "Hello, world!";
7      std::cout << "Enter a letter: ";
8
9      char ch;
10     std::cin >> ch;
11
12     int index = hello.find(ch);
13     std::cout << ch << " was found at index " << index << '\n';
14
15     return 0;
16 }
```

Can you identify the assumption that may be violated? The answer is that the user may enter a character that isn't in the string. If that happens, the find() function will return an index of -1, which will get printed.

Here's a new version with error checking:

```
1  #include <iostream>
2  #include <string>
3
```

```
int main()
{
    std::string hello = "Hello, world!";
    std::cout << "Enter a letter: ";

    char ch;
    std::cin >> ch;

    int index = hello.find(ch);
    if (index != -1) // handle case where find() failed to find the character in the string
        std::cout << ch << " was found at index " << index << '\n';
    else
        std::cout << ch << " wasn't found" << '\n';

    return 0;
}
```

Problem: When program receives input (either from the user, or a file), the input may not be in the correct format. Here's the sample program you saw previously that illustrates this flaw:

```
#include <iostream>
#include <string>

int main()
{
    std::string hello = "Hello, world!";
    std::cout << "Enter an index: ";

    int index;
    std::cin >> index;

    std::cout << "Letter #" << index << " is " << hello [index] << std::endl;

    return 0;
}
```

And here's the version that checks the user input to make sure it is valid:

```
#include <iostream>
#include <string>

int main()
{
    std::string hello = "Hello, world!";
    int index;

    do
    {
        std::cout << "Enter an index: ";
        std::cin >> index;

        //handle case where user entered a non-integer
        if (std::cin.fail())
        {
            std::cin.clear(); // reset any error flags
            std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
            index = -1; // ensure index has an invalid value so the loop doesn't terminate
            continue; // this continue may seem extraneous, but it explicitly signals an intent to term
inate this loop iteration
        }

    } while (index < 0 || index >= hello.size()); // handle case where user entered an out of range int
eger

    std::cout << "Letter #" << index << " is " << hello [index] << std::endl;

    return 0;
```

```
}
```

Note that this last example has two-levels of error checking: first, we need to make sure the user's input was properly read into variable index. Then we needed to ensure index was within range of the array.

Handling assumption errors

Now that you know where assumption errors typically occur, let's talk about different ways to handle them when they do occur. There is no best way to handle an error -- it really depends on the nature of the problem and whether the problem can be fixed or not.

Here are some typical methods:

1) Quietly skip the code that depends on the assumption being valid:

```
void printString(const char *cstring)
{
    // Only print if cstring is non-null
    if (cstring)
        std::cout << cstring;
}
```

In the above example, if cstring is null, we don't print anything. We have skipped the code that depends on cstring being non-null. This can be a good option if the statement being skipped isn't critical and doesn't impact the program logic. The primary challenge with doing so is that the caller or user have no way to identify that something went wrong.

2) If we are in a function, return an error code back to the caller and let the caller deal with the problem.

```
#include <array>

int getArrayValue(const std::array<int, 10> &array, int index)
{
    // use if statement to detect violated assumption
    if (index < 0 || index >= array.size())
       return -1; // return error code to caller

    return array[index];
}
```

In this case, the function returns -1 if the caller passes in an invalid index. Returning an enumerated value would be even better.

3) If we want to terminate the program immediately, the **exit** function that lives in <cstdlib> can be used to return an error code to the operating system:

```
#include <cstdlib> // for exit()
#include <array>

int getArrayValue(const std::array<int, 10> &array, int index)
{
    // use if statement to detect violated assumption
    if (index < 0 || index >= array.size())
       exit(2); // terminate program and return error number 2 to OS

    return array[index];
}
```

If the caller passes in an invalid index, this program will terminate immediately (with no error message) and pass error code 2 to the operating system.

4) If the user has entered invalid input, ask the user to enter the input again.

```
#include <iostream>
#include <string>

int main()
{
    std::string hello = "Hello, world!";
    int index;
```

```
    do
    {
        std::cout << "Enter an index: ";
        std::cin >> index;

        //handle case where user entered a non-integer
        if (std::cin.fail())
        {
            std::cin.clear(); // reset any error flags
            std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
            index = -1; // ensure index has an invalid value so the loop doesn't terminate
            continue; // this continue may seem extraneous, but it explicitly signals an intent to term
inate this loop iteration...
        }

        // ...just in case we added more stuff here later

    } while (index < 0 || index >= hello.size()); // handle case where user entered an out of range int
eger

    std::cout << "Letter #" << index << " is " << hello [index] << std::endl;

    return 0;
}
```

5) cerr is another mechanism that is meant specifically for printing error messages. **cerr** is an output stream (just like cout) that is defined in <iostream>. Typically, cerr writes the error messages on the screen (just like cout), but it can also be individually redirected to a file.

```
void printString(const char *cstring)
{
    // Only print if cstring is non-null
    if (cstring)
        std::cout << cstring;
    else
        std::cerr << "function printString() received a null parameter";
}
```

In the above example, we not only skip the bad line, we also log an error so the user can later determine why the program didn't execute as expected.

6) If working in some kind of graphical environment (eg. MFC, SDL, QT, etc…), it is common to pop up a message box with an error code and then terminate the program.

The specific details of how to do this depend on the environment.

Note: Because this lesson was getting long, discussions of assert have been moved to the next lesson.


7.12a -- Assert and static_assert


Index


7.11 -- Recursion

Share this: