

7.12 — Handling errors, cerr and exit

BY ALEX ON OCTOBER 22ND, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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] << '\n';

```

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{ static_cast<int>(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
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{ static_cast<int>(hello.find(ch)) };
13     if (index != -1) // handle case where find() failed to find the character in the string
14         std::cout << ch << " was found at index " << index << '\n';
15     else
16         std::cout << ch << " wasn't found" << '\n';
17
18     return 0;
19 }

```

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:

```

1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello{ "Hello, world!" };
7      std::cout << "Enter an index: ";
8
9      int index;
10     std::cin >> index;
11
12     std::cout << "Letter #" << index << " is " << hello [index] << '\n';
13
14     return 0;
15 }

```

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

```

1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello{ "Hello, world!" };
7      int index;
8
9      do

```

```

10     {
11         std::cout << "Enter an index: ";
12         std::cin >> index;
13
14         //handle case where user entered a non-integer
15         if (std::cin.fail())
16         {
17             std::cin.clear(); // reset any error flags
18             std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
19             index = -1; // ensure index has an invalid value so the loop doesn't terminate
20             continue; // this continue may seem extraneous, but it explicitly signals an inten
21         }
22
23     } while (index < 0 || index >= static_cast<int>(hello.size())); // handle case where user
24
25     std::cout << "Letter #" << index << " is " << hello [index] << '\n';
26
27     return 0;
28 }

```

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:

```

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

```

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.

```

1 #include <array>
2
3 // assume the array only holds positive values
4 int getArrayValue(const std::array<int, 10> &array, int index)
5 {
6     // use if statement to detect violated assumption
7     if (index < 0 || index >= static_cast<int>(array.size()))
8         return -1; // return error code to caller
9
10    return array[index];
11 }

```

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

This method only works if you can find a return value that can't naturally be in set of normal return values for that function. For example, if the array above could hold negative numbers, then array might return -1 normally. This precludes using that value to indicate an error.

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:

```
1  #include <cstdlib> // for std::exit()
2  #include <array>
3
4  int getArrayValue(const std::array<int, 10> &array, int index)
5  {
6      // use if statement to detect violated assumption
7      if (index < 0 || index >= static_cast<int>(array.size()))
8          std::exit(2); // terminate program and return error number 2 to OS
9
10     return array[index];
11 }
```

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.

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello{ "Hello, world!" };
7      int index;
8
9      do
10     {
11         std::cout << "Enter an index: ";
12         std::cin >> index;
13
14         //handle case where user entered a non-integer
15         if (std::cin.fail())
16         {
17             std::cin.clear(); // reset any error flags
18             std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
19             index = -1; // ensure index has an invalid value so the loop doesn't terminate
20             continue; // this continue may seem extraneous, but it explicitly signals an inten
21         }
22
23         // ...just in case we added more stuff here later
24
25     } while (index < 0 || index >= static_cast<int>(hello.size())); // handle case where user
26
27     std::cout << "Letter #" << index << " is " << hello [index] << '\n';
28
29     return 0;
30 }
```

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.

```
1  void printString(const char *cstring)
2  {
3      // Only print if cstring is non-null
```

```
4     if (cstring)
5         std::cout << cstring;
6     else
7         std::cerr << "function printString() received a null parameter";
8 }
```

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

 [C++ TUTORIAL](#) |  [PRINT THIS POST](#)

82 comments to 7.12 — Handling errors, cerr and exit

[« Older Comments](#) [1](#) [2](#)

Gacrux

[November 12, 2019 at 11:54 pm · Reply](#)

The hello world example needs `static_cast<size_t>(index)` or won't compile here. Weird.



C:\Users\admin\Documents\cbprojects\7.12\main.cpp | 24 | error: comparison of integer expressions of different signedness: 'int' and 'std::__cxx11::basic_string<char>::size_type' {aka 'long long unsigned int'} [-Werror=sign-compare] |

C:\Users\admin\Documents\cbprojects\7.12\main.cpp | 24 | error: conversion to 'std::__cxx11::basic_string<char>::size_type' {aka 'long long unsigned int'} from 'int' may change the sign of the result [-Werror=sign-conversion] |

C:\Users\admin\Documents\cbprojects\7.12\main.cpp | 27 | error: conversion to 'std::__cxx11::basic_string<char>::size_type' {aka 'long long unsigned int'} from 'int' may change the sign of the result [-Werror=sign-conversion] |



nascardriver

November 13, 2019 at 3:56 am · Reply.

Some lessons were written without warnings about conversions. I updated this lesson, thanks for pointing it out!



Samira Ferdi

September 26, 2019 at 5:56 pm · Reply.

Hi, Alex and Nascardriver! How to handle if the argument that we passed to the function printString() is const or non-const char pointer?

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



nascardriver

September 27, 2019 at 12:21 am · Reply.

I don't understand your question. Can you provide example calls and an explanation what you want to happen?



Samira Ferdi

September 29, 2019 at 4:50 pm · Reply.

```
1 void printString(const char *cstring)
2 {
3     // Only print if cstring is non-null
4     if (cstring)
5         std::cout << cstring;
6 }
7
8 int main()
9 {
10     char ch{ 'a' };
11     char *chPtr{ &ch };
12
13     printString(chPtr); //This print something strange.
14
15 }
```

**nascar driver**

September 30, 2019 at 2:43 am · Reply

`std::cout<<`` prints everything until it finds a null-terminator. A single ``char`` doesn't have a zero-terminator, so `std::cout<<`` continues reading from the memory after ``ch``, causing undefined behavior. You can't detect this. Your function name and documentation should clarify what the function does with its parameters.

**Samira Ferdi**

September 26, 2019 at 5:39 pm · Reply

Hi, Alex and Nascar driver!

I wanna ask help to you to clarify my understanding about Alex's code

```

1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string hello = "Hello, world!";
7      int index;
8
9      do
10     {
11         std::cout << "Enter an index: ";
12         std::cin >> index;
13
14         //handle case where user entered a non-integer
15         if (std::cin.fail())
16         {
17             std::cin.clear(); // reset any error flags
18             std::cin.ignore(32767, '\n'); // ignore any characters in the input buffer
19             index = -1; // ensure index has an invalid value so the loop doesn't terminate
20             continue; // this continue may seem extraneous, but it explicitly signals an in
21         }
22
23     } while (index < 0 || index >= hello.size()); // handle case where user entered an out
24
25     std::cout << "Letter #" << index << " is " << hello [index] << std::endl;
26
27     return 0;
28 }
```

I try to understand what this part means

```

1  index = -1; // ensure index has an invalid value so the loop doesn't terminate
2  continue; // this continue may seem extraneous, but it explicitly signals an intent to termi
```

So, my understanding is like this: so, what Alex basically did is to strongly enforce the user to enter an integer. If user enter non-integers, the user must enter again and the consequences of this enforcement is the while loop part is never executed if user enters non-integers and that's basically mean we don't have to do useless non-integers comparing because it's useless in this program context.

So, they are basically like Alex said, two steps of checking: enter an integer first, and checking and handle case where user entered an out of range integer

Is my understanding correct?

**nascardriver**

September 27, 2019 at 12:20 am · Reply

`continue` doesn't skip the condition check. This code would do the same if you removed line 20.

Alex uses `index = -1`, because if extraction fails, `index` will be set to `0`. If `index` is `0` in line 23, the loop will stop, which is not what we want. By setting `index` to `-1`, Alex makes sure that the condition is `true`.

If the loop's condition is changed, this trick might not work anymore. I'd use a `while (true)` loop and `break` if `index` is valid.

**Samira Ferdi**

October 1, 2019 at 4:33 pm · Reply

Hi, Nascardriver! Thanks for your great explanation!

So, using while loop is like this:

```

1  #include <iostream>
2  #include <string>
3  #include <limits>
4
5  int main()
6  {
7      std::string text{ "Hello world!" };
8      constexpr int minIndex{ 0 };
9      const int maxIndex{ static_cast<int>( text.length() ) };
10
11     while(true)
12     {
13         std::cout << "Enter an index: ";
14         int index{};
15         std::cin >> index;
16
17         if(std::cin.fail())
18         {
19             std::cin.clear(); // reset any error flags
20
21             // ignore any characters in the input buffer
22             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
23             std::cout << "\n[ERROR]: Extraction failed! "
24                       << "You enter not an integer! Please, enter an integer!"
25                       << "\n\n";
26         }
27
28         //Bound checking
29         else if(index < minIndex || index > maxIndex)
30         {
31             std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
32             std::cout << "\n[ERROR]: The index is outside the bound! "
33                       << "Please enter an index that in the bound!\n\n";
34         }
35
36         else
37         {
38             std::cout << "Letter #" << index << " is "
39                       << text[static_cast<unsigned int>(index)] << '\n';
40             break;
41         }

```

```
42  
43     }  
44  
45     return 0;  
46 }
```

So, there is no need of using continue and index = -1 here.



nascardriver

October 2, 2019 at 4:41 am · Reply.

Yes. Now the code isn't as easy to read anymore. Adding a `getInteger()` and a `getInteger(int min, int max)` increases readability and increases reusability.

[« Older Comments](#)

1

2