7.15 — Detecting and handling errors

ALEX MARCH 30, 2021

In lesson 7.14 -- Common semantic errors in C++, we covered many types of common C++ semantic errors that new C++ programmers run into with the language. If an error is the result of a misused language feature or logic error, the error can simply be corrected.

But most errors in a program don't occur as the result of inadvertently misusing language features -- rather, most errors occur due to faulty assumptions made by the programmer and/or a lack of proper error detection/handling.

For example, in a function designed to look up a grade for a student, you might have assumed:

- The student being looked up will exist.
- All student names will be unique.
- The class uses letter grading (instead of pass/fail).

What if any of these assumptions aren't true? If the programmer didn't anticipate these cases, the program will likely malfunction or crash when such cases arise (usually at some point in the future, well after the function has been written).

There are three key places where assumption errors typically occur:

- When a function returns, the programmer may have assumed the called function was successful when it was not.
- When program receives input (either from the user, or a file), the programmer may have assumed the input was in the correct format and semantically valid when it was not.
- When a function has been called, the programmer may have assumed the parameters would be semantically valid when they were not.

Many new programmers write code and then only test thehappy path: only the cases where there are no errors. But you should also be planning for and testing your sad paths, where things can and will go wrong. In lesson 3.10 -- Finding issues before they become problems, we defined defensive programming as the practice of trying to anticipate all of the ways software can be misused, either by end-users, or by developers (either the programmer themselves, or others). Once you've anticipated (or discovered) some misuse, the next thing to do is handle it.

In this lesson, we'll talk about error handling strategies (what to do when things go wrong) inside a function. In the subsequent lessons, we'll talk about validating user input, and then introduce a useful tool to help document and validate assumptions.

Handling errors in functions

Functions may fail for any number of reasons -- the caller may have passed in an argument with an invalid value, or something may fail within the body of the function. For example, a function that opens a file for reading might fail if the file cannot be found.

When this happens, you have quite a few options at your disposal. 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.

There are 4 general strategies that can be used:

- Handle the error within the function
- Pass the error back to the caller to deal with
- Halt the program
- Throw an exception

Handling the error within the function

If possible, the best strategy is to recover from the error in the same function in which the error occurred, so that the error can be contained and corrected without impacting any code outside the function. There are two options here: retry until successful, or cancel the operation being executed.

If the error has occurred due to something outside of the program's control, the program can retry until success is achieved. For example, if the program requires an internet connection, and the user has lost their connection, the program may be able to display a warning and then use a loop to periodically recheck for internet connectivity. Alternatively, if the user has entered invalid input, the program can ask the user to try again, and loop until the user is successful at entering valid input. We'll show examples of handling invalid input and using loops to retry in the next lesson (7.16 -- std::cin and handling invalid input).

An alternate strategy is just to ignore the error and/or cancel the operation. For example:

```
1 void printDivision(int x, int y)
  {
      if (y != 0)
2         std::cout << static_cast<double>(x) /
3 y;
4 }
```

In the above example, if the user passed in an invalid value for `y`, we just ignore the request to print the result of the division operation. The primary challenge with doing this is that the caller or user have no way to identify that something went wrong. In such case, printing an error message can be helpful:

```
1 void printDivision(int x, int y)
  {
      if (y != 0)
2         std::cout << static_cast<double>(x) / y;
3     else
4         std::cerr << "Error: Could not divide by
  zero\n";
  }
```

However, if the calling function is expecting the called function to produce a return value or some useful side-effect, then just ignoring the error may not be an option.

Passing errors back to the caller

In many cases, the error can't reasonably be handled in the function that detects the error. For example, consider the following function:

```
1 double doDivision(int x, int y)
  {
2     return static_cast<double>(x) /
3 y;
  }
```

If `y` is `0`, what should we do? We can't just skip the program logic, because the function needs to return some value. We shouldn't ask the user to enter a new value for `y` because this is a calculation function, and introducing input routines into it may or may not be appropriate for the program calling this function.

In such cases, the best option can be to pass the error back to the caller in hopes that the caller will be able to deal with it.

How might we do that?

If the function has a void return type, it can be changed to return a Boolean that indicates success or failure. For example, instead of:

```
void printDivision(int x, int y)
{
    if (y != 0)
        std::cout << static_cast<double>(x) / y;
    else
        std::cerr << "Error: Could not divide by
zero\n";
}
```

We can do this:

```
bool printDivision(int x, int y)
{
    if (y == 0)
    {
        std::cerr << "Error: could not divide by
zero\n";
        return false;
    }

    std::cout << static_cast<double>(x) / y;

    return true;
}
```

That way, the caller can check the return value to see if the function failed for some reason.

If the function returns a normal value, things are a little more complicated. In some cases, the full range of return values isn't used. In such cases, we can use a return value that wouldn't otherwise be possible to occur normally to indicate an error. For example, consider the following function:

```
// The reciprocal of x is
1/x
double reciprocal(double
x)
{
    return 1.0 / x;
}
```

The reciprocal of some number `x` is defined as `1/x`, and a number multiplied by its reciprocal equals 1.

However, what happens if the user calls this function as `reciprocal(0.0)`? We get a `divide by zero` error and a program crash, so clearly we should protect against this case. But this function must return a double value, so what value should we return? It turns out that this function will never produce `0.0` as a legitimate result, so we can return `0.0` to indicate an error case.

```
// The reciprocal of x is 1/x, returns 0.0 if
x=0
double reciprocal(double x)
{
    if (x == 0.0)
       return 0.0;

    return 1.0 / x;
}
```

However, if the full range of return values are needed, then using the return value to indicate an error will not be possible (because the caller would not be able to tell whether the return value is a valid value or an error value). In such a case, an `out parameter` (covered in lesson 11.3 -- Passing arguments by reference) might be a viable choice.

Fatal errors

If the error is so bad that the program can not continue to operate properly, this is called a non-recoverable error (also called a fatal error). In such cases, the best thing to do is terminate the program. If your code is in `main()` or a function called directly from `main()`, the best thing to do is let `main()` return a non-zero status code. However, if you're deep in some nested subfunction, it may not be convenient or possible to propagate the error all the way back to `main()`. In such a case, a `halt statement` (such as `std::exit()`) can be used.

For example:

```
double doDivision(int x, int y)
{
    if (y == 0)
    {
        std::cerr << "Error: Could not divide by
zero\n";
        std::exit(1);
    }
    return static_cast<double>(x) / y;
}
```

Exceptions

Because returning an error from a function back to the caller is complicated (and the many different ways to do so leads to inconsistency, and inconsistency leads to mistakes), C++ offers an entirely separate way to pass errors back to the caller: `exceptions`.

The basic idea is that when an error occurs, an exception is "thrown". If the current function does not "catch" the error, the caller of the function has a chance to catch the error. If the caller does not catch the error, the caller's caller has a chance to catch the error. The error progressively moves up the call stack until it is either caught and handled (at which point execution continues normally), or until main() fails to handle the error (at which point the program is terminated with an exception error).

We cover exception handling in chapter 20 of this tutorial series.

Next lesson
7.16 std::cin and handling invalid input

Back to table of contents

Previous lesson
7.14 Common semantic errors in C++

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest