9.4 — Detecting and handling errors

learncpp.com/cpp-tutorial/detecting-and-handling-errors/

In lesson 9.3 -- 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 a 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 arguments would be semantically valid when they were not.

Many new programmers write code and then only test the **happy 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 (9.5 -- std::cin and handling invalid input).

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

```
// Silent failure if y=0
void printIntDivision(int x, int y)
{
    if (y != 0)
        std::cout << x / y;
}
```

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:

```
void printIntDivision(int x, int y)
{
    if (y != 0)
        std::cout << x / y;
    else
        std::cout << "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:

```
int doIntDivision(int x, int y)
{
    return x / 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 `bool` that indicates success or failure. For example, instead of:

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

We can do this:

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

    std::cout << 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;
}
```

A **sentinel value** is a value that has some special meaning in the context of a function or algorithm. In our `reciprocal()` function above, `0.0` is a sentinel value indicating that the function failed. The caller can test the return value to see if it matches the sentinel value -- if so, then the caller knows the function failed.

However, if the full range of return values can be produced by the function, then using a sentinel value to indicate an error is problematic (because the caller would not be able to tell whether the return value is a valid value or an error value).

Related content

In such a case, returning a `std::optional` would be a good choice. We cover `std::optional` in lesson 12.15 -- std::optional.

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 doIntDivision(int x, int y)
{
    if (y == 0)
    {
        std::cout << "Error: Could not divide by zero\n";
        std::exit(1);
    }
    return 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 27 of this tutorial series.

When to use `std::cout` vs `std::cerr` vs logging

You may be wondering when you should be using `std::cerr` vs `std::cout` vs logging to a text file.

By default, both `std::cout` and `std::cerr` print text to the console. However, modern OSes provide a way to redirect output streams to files, so that the output can be captured for review or automated processing later.

For this discussion, it is useful to differentiate two types of applications:

- **Interactive applications** are those that the user will interact with after running. Most standalone applications, like games and music apps, fall into this category.
- **Non-interactive applications** are applications that do not require user interaction to operate. The output of these programs may be used as input for another application

Within non-interactive applications, there are two types:

- **Tools** are non-interactive applications that are typically launched in order to produce some immediate result, and then terminate after producing such a result. An example of this is Unix's grep command, which is a utility that searches text for lines that match some pattern.
- **Services** are non-interactive applications that typically run silently in the background to perform some ongoing function. An example of this would be a virus scanner.

Here's some rules of thumb:

- Use `std::cout` for all conventional, user-facing text.
- For an interactive program, use `std::cout` for normal user-facing error messages (e.g. "Your input was invalid"). Use `std::cerr` or a logfile for status and diagnostic information that may be helpful for diagnosing issues but probably isn't interesting for normal users. This can include technical warnings and errors (e.g. bad input to function x), status updates (e.g. successfully opened file x, failed to connect to internet service x), percentage completion of long tasks (e.g. encoding 50% complete), etc…
- For a non-interactive program (tool or service), use `std::cerr` for error output only (e.g. could not open file x). This allows errors to be displayed or parsed separately from normal output.
- For any application type that is transactional in nature (e.g. one that processes specific events, such as an interactive web browser or non-interactive web server), use a logfile to produce a transactional log of events that can be reviewed later.

For example, outputting which file it's processing right now, percentage of completion, timestamps of when it started a certain stage of computing, warning, and error messages.