9.6 — Assert and static_assert

ALEX DECEMBER 28, 2023

In a function that takes parameters, the caller may be able to pass in arguments that are syntactically valid but semantically meaningless. For example, in the previous lesson (9.4 -- Detecting and handling errors (https://www.learncpp.com/cpp-tutorial/detecting-and-handling-errors/)), we showed the following sample function:

```
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";
}
```

This function does an explicit check to see if `y` is `0`, since dividing by zero is a semantic error and will cause the program to crash if executed.

In the prior lesson, we discussed a couple of ways to deal with such problems, including halting the program, or skipping the offending statements.

Both of those options are problematic though. If a program skips statements due to an error, then it is essentially failing silently. Especially while we are writing and debugging programs, silent failures are bad, because they obscure real problems. Even if we print an error message, that error message may be lost among the other program output, and it may be non-obvious where the error message is being generated or how the conditions that triggered the error message occurred. Some functions may be called tens or hundreds of times, and if only one of those cases is generating a problem, it can be hard to know which one.

If the program terminates (via `std::exit`) then we will have lost our call stack and any debugging information that might help us isolate the problem. `std::abort` is a better option for such cases, as typically the developer will be given the option to start debugging at the point where the program aborted.

Preconditions, invariants, and postconditions

In programming, a **precondition** is any condition that must be true prior to the execution of some section of code (typically the body of a function). In the prior example, our check that `y != 0` is a precondition that ensures `y` has a non-zero value before dividing by `y`.

Preconditions for a function are best placed at the top of a function, using an early return to return back to the caller if the precondition isn't met. For example:

```
    void printDivision(int x, int y)
    {
        if (y == 0) // handle
        {
            std::cerr << "Error: Could not divide by zero\n";
            return; // bounce the user back to the caller
        }

        // We now know that y != 0
        std::cout << static_cast<double>(x) / y;
    }
```

This is sometimes known as the “bouncer pattern”.

An **invariant** is a condition that must be true while some section of code is executing. This is often used with loops, where the loop body will only execute so long as the invariant is true.

Similarly, a **postcondition** is something that must be true after the execution of some section of code. Our function doesn’t have any postconditions.

Assertions

Using a conditional statement to detect an invalid parameter (or to validate some other kind of assumption), along with printing an error message and terminating the program, is such a common method of detecting problems that C++ provides a shortcut method for doing this.

An **assertion** is an expression that will be true unless there is a bug in the program. If the expression evaluates to `true`, the assertion statement does nothing. If the conditional expression evaluates to `false`, an error message is displayed and the program is terminated (via `std::abort`). This error message typically contains the expression that failed as text, along with the name of the code file and the line number of the assertion. This makes it very easy to tell not only what the problem was, but where in the code the problem occurred. This can help with debugging efforts immensely.

Key insight

When an assertion evaluates to false, your program is immediately stopped. This gives you an opportunity to use debugging tools to examine the state of your program and determine why the assertion failed. Working backwards, you can then find and fix the issue.

Without an assertion to detect an error and fail, such an error would likely cause your program to malfunction later. In such cases, it can be very difficult to determine where things are going wrong, or what the root cause of the issue actually is.

In C++, runtime assertions are implemented via the **assert** preprocessor macro, which lives in the <cassert> header.

```
#include <cassert> // for assert()
#include <cmath> // for std::sqrt
#include <iostream>

double calculateTimeUntilObjectHitsGround(double initialHeight, double gravity)
{
  assert(gravity > 0.0); // The object won't reach the ground unless there is positive gravity.

  if (initialHeight <= 0.0)
  {
    // The object is already on the ground. Or buried.
    return 0.0;
  }

  return std::sqrt((2.0 * initialHeight) / gravity);
}

int main()
{
  std::cout << "Took " << calculateTimeUntilObjectHitsGround(100.0, -9.8) << " second(s)\n";

  return 0;
}
```

When the program calls `calculateTimeUntilObjectHitsGround(100.0, -9.8)`, `assert(gravity > 0.0)` will evaluate to `false`, which will trigger the assert. That will print a message similar to this:

```
dropsimulator: src/main.cpp:6: double calculateTimeUntilObjectHitsGround(double, double): Assertion 'gravit
y > 0.0' failed.
```

The actual message varies depending on which compiler you use.

Although asserts are most often used to validate function parameters, they can be used anywhere you would like to validate that something is true.

Although we told you previously to avoid preprocessor macros, asserts are one of the few preprocessor macros that are considered acceptable to use. We encourage you to use assert statements liberally throughout your code.

Making your assert statements more descriptive

Sometimes assert expressions aren't very descriptive. Consider the following statement:

```
assert(found);
```

If this assert is triggered, the assert will say:

```
Assertion failed: found, file C:\\VCProjects\\Test.cpp, line 34
```

What does this even mean? Clearly `found` was `false` (since the assert triggered), but what wasn't found? You'd have to go look at the code to determine that.

Fortunately, there's a little trick you can use to make your assert statements more descriptive. Simply add a string literal joined by a logical AND:

```
    assert(found && "Car could not be found in database");
```

Here's why this works: A string literal always evaluates to Boolean `true`. So if `found` is `false`, `false && true` is `false`. If `found` is `true`, `true && true` is `true`. Thus, logical AND-ing a string literal doesn't impact the evaluation of the assert.

However, when the assert triggers, the string literal will be included in the assert message:

```
Assertion failed: found && "Car could not be found in database", file C:\\VCProjects\\Test.cpp, line 34
```

That gives you some additional context as to what went wrong.

Asserts vs error handling

Assertions and error handling are similar enough that their purposes can be confused, so let's clarify:

The goal of an assertion is to catch programming errors by documenting something that should never happen. If that thing does happen, then the programmer made an error somewhere, and that error can be identified and fixed. Assertions do not allow recovery from errors (after all, if something should never happen, there's no need to recover from it), and the program will not produce a friendly error message.

On the other hand, error handling is designed to gracefully handle cases that could happen (however rarely) in release configurations. These may or may not be recoverable, but one should always assume a user of the program may encounter them.

Best practice

Use assertions to document cases that should be logically impossible.

Assertions are also sometimes used to document cases that were not implemented because they were not needed at the time the programmer wrote the code:

```
    assert(moved && "Need to handle case where student was just moved to another classroom");
```

That way, if a future user of the code does encounter a situation where this case is needed, the code will fail with a useful error message, and the programmer can then determine how to implement that case.

NDEBUG

The `assert` macro comes with a small performance cost that is incurred each time the assert condition is checked. Furthermore, asserts should (ideally) never be encountered in production code (because your code should already be thoroughly tested). Consequently, many developers prefer that asserts are only active in debug builds. C++ comes with a way to turn off asserts in production code. If the macro `NDEBUG` is defined, the assert macro gets disabled.

Some IDEs set `NDEBUG` by default as part of the project settings for release configurations. For example, in Visual Studio, the following preprocessor definitions are set at the project level: `WIN32;NDEBUG;_CONSOLE`. If you're using Visual Studio and want your asserts to trigger in release builds, you'll need to remove `NDEBUG` from this setting.

If you're using an IDE or build system that doesn't automatically define `NDEBUG` in release configuration, you will need to add it in the project or compilation settings manually.

Some assert limitations and warnings

There are a few pitfalls and limitations to asserts. First, the assert itself can be improperly written. If this happens, the assert will either report an error where none exists, or fail to report a bug where one does exist.

Second, your asserts should have no side effects -- that is, the program should run the same with and without the assert. Otherwise, what you are testing in a debug configuration will not be the same as in a release configuration (assuming you ship with NDEBUG).

Also note that the `abort()` function terminates the program immediately, without a chance to do any further cleanup (e.g. close a file or database). Because of this, asserts should be used only in cases where corruption isn't likely to occur if the program terminates unexpectedly.

()static_assert (#static_assert)

C++ also has another type of assert called `static_assert`. A **static_assert** is an assertion that is checked at compile-time rather than at runtime, with a failing `static_assert` causing a compile error. Unlike assert, which is declared in the <cassert> header, static_assert is a keyword, so no header needs to be included to use it.

A `static_assert` takes the following form:

```
static_assert(condition, diagnostic_message)
```

If the condition is not true, the diagnostic message is printed. Here's an example of using static_assert to ensure types have a certain size:

```
static_assert(sizeof(long) == 8, "long must be 8 bytes");
static_assert(sizeof(int) >= 4, "int must be at least 4 bytes");

int main()
{
	return 0;
}
```

On the author's machine, when compiled, the compiler errors:

```
1>c:\consoleapplication1\main.cpp(19): error C2338: long must be 8 bytes
```

A few useful notes about `static_assert`:

- Because `static_assert` is evaluated by the compiler, the condition must be a constant expression.
- `static_assert` can be placed anywhere in the code file (even in the global namespace).
- `static_assert` is not compiled out in release builds.

Prior to C++17, the diagnostic message must be supplied as the second parameter. Since C++17, providing a diagnostic message is optional.

Next lesson
9.x Chapter 9 summary and quiz

Back to table of contents

Previous lesson
9.5 std::cin and handling invalid input