7.12a — Assert and static_assert

BY ALEX ON MAY 14TH, 2017 | LAST MODIFIED BY ALEX ON SEPTEMBER 2ND, 2017

Using a conditional statement to detect a violated assumption, along with printing an error message and terminating the program, is such a common response to problems that C++ provides a shortcut method for doing this. This shortcut is called an **assert**.

An **assert statement** is a preprocessor macro that evaluates a conditional expression at runtime. If the conditional expression is true, the assert statement does nothing. If the conditional expression evaluates to false, an error message is displayed and the program is terminated. This error message contains the conditional expression that failed, along with the name of the code file and the line number of the assert. 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.

The assert functionality lives in the <cassert> header, and is often used both to check that the parameters passed to a function are valid, and to check that the return value of a function call is valid.

```
#include <cassert> // for assert()

int getArrayValue(const std::array<int, 10> &array, int index)
{
    // we're asserting that index is between 0 and 9
    assert(index >= 0 && index <= 9); // this is line 6 in Test.cpp

    return array[index];
}
```

If the program calls getArrayValue(array, -3), the program prints the following message:

```
Assertion failed: index >= 0 && index <=9, file C:\\VCProjects\\Test.cpp, line 6
```

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 something wasn't found, but what? 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 C-style string description joined with a logical AND:

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

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

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

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

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

NDEBUG and other considerations

The assert() function 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: #define NDEBUG.

```
#define NDEBUG

// all assert() calls will now be ignored to the end of the file
```

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.

Do note that the exit() function and assert() function (if it triggers) terminate the program immediately, without a chance to do any further cleanup (e.g. close a file or database). Because of this, they should be used judiciously (only in cases where corruption isn't likely to occur if the program terminates unexpectedly).

Static_assert

C++11 adds another type of assert called **static_assert**. static_assert takes the form

```
static_assert
```

Unlike assert, which operates at runtime, static_assert is designed to operate at compile time, causing the compiler to error if the condition is not true. If the condition is false, 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 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 notes. Because static_assert is evaluated by the compiler, the conditional part of a static_assert must be able to be evaluated at compile time. Because static_assert is not evaluated at runtime, static_assert statements can also be placed anywhere in the code file (even in global space).

In C++11, a diagnostic message must be supplied as the second parameter. In C++17, providing a diagnostic message is optional.

Exceptions

C++ provides one more method for detecting and handling errors known as exception handling. The basic idea is that when an error occurs, the error 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 stack until it is either caught and handled, or until main() fails to handle the error. If nobody handles the error, the program typically terminates with an exception error.

Exception handling is an advanced C++ topic, and we cover it in much detail in chapter 14 of this tutorial.

 **7.13 -- Command line arguments**