7.12a — Assert and static_assert

BY ALEX ON MAY 14TH, 2017 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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\\Tes
```

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**

 **Index**

 **7.12 -- Handling errors, cerr and exit**

C++ TUTORIAL | PRINT THIS POST

50 comments to 7.12a — Assert and static_assert

Ged
December 11, 2019 at 6:30 am · Reply

1. For some reason the assert still works. I'm using the newest Visual Studio.

```
#include <iostream>
#include <cassert>

#define NDEBUG // Should turn assert off

int main()
{
    std::cout << "Testing" << '\n';
    int x{ -1 };
```

```
    assert(x > 0 && "x is less than 0");
    return 0;
}
```

2. Just remembered a problem that happened a while back when I was writing the "blackjack game". Something was under my function and I got an error and then realized that it should be over it. But the error message was not really understandable. So I just wanna ask what is the best order to write all the things that we covered outside the main function?

For example:

#include <EXAMPLE>

Global variables (1)
Enum (2)
Struct (3)
Function (4)

int main()
{

}

nascardriver
December 11, 2019 at 6:38 am · Reply

1.
`NDEBUG` needs to be defined before you include to disable assertions.

2.
There's no general rule. What I do, and what works well

```
#include "local_files"
#include "in_alphabetical_order"

#include <system_files>
#include <in_alphabetical_order>

types and type aliases

// (global variables, shouldn't exist)

functions

main
```

NXPY
March 24, 2019 at 8:10 am · Reply

I'm having a bit of trouble with assert.
When I use assert , the program crashes(exe has stopped working) and the command prompt window shows the following line :

'This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information. '

The process then returns 3.

Alex

March 24, 2019 at 9:02 pm · Reply

That's bizarre. What OS and compiler are you using?

NXPY
March 24, 2019 at 10:52 pm · Reply

OS - Windows 7 Ultimate 32-bit
Compiler - GNU GCC
IDE - Code:: Blocks 17.12

Alex
March 26, 2019 at 12:30 pm · Reply

Can you share the code that is crashing for you?

hellmet
October 26, 2019 at 10:32 am · Reply

I'm guessing that's because of the ..erm emulation (if that's what it is) that GCC (or Cygwin) on windows uses. Native builds work fine (MSVC and Clang).

Also this previous comment here (similar issue)
https://www.learncpp.com/cpp-tutorial/7-12a-assert-and-static_assert/comment-page-1/#comment-294033

Asgar
February 18, 2019 at 5:03 pm · Reply

Hi,

One thing about this line:
"An assert statement is a preprocessor macro that evaluates a conditional expression at runtime."

Isn't preprocessor macro supposed to guide the compiler to do something at compile-time? Does the preprocessor replace the assert statement with something that tells the compiler that the following piece of code should be evaluated at run-time and that the compiler should ignore it?

nascardriver
February 19, 2019 at 8:33 am · Reply

The preprocessor replaces ever use of @assert with C++ code, that is then compiled by the compiler along all the other code.

hellmet
October 26, 2019 at 10:36 am · Reply

Wait, I thought preprocessor directives are directly evaluated by the preprocessor itself (ex: ifdef, ifndef etc...) and since the preprocessor can check the condition during this stage, I'm led to think a failed static_assert causes the 'compiler' to not even go past the preprocessor stage (consequently no machine code is generated implying no 'compilation' occurs).

nascardriver
October 27, 2019 at 1:19 am · Reply

`static_assert` is no macro, the preprocessor doesn't do anything with it.
`assert` has to be a macro to access the __LINE__ etc. macros in the right place.
The preprocessor expands it, but it doesn't do any checks.

hellmet
October 27, 2019 at 3:05 am · Reply

Whoops my bad!
Okay the second one makes sense!
Thank you!

Jeroen P. Broks
February 18, 2019 at 11:23 am · Reply

On static assert I see some possibilities, since I prefer to work open source (almost all the code I wrote the past few years can be found on github.com), I deem it possible that (although I do use either make files or compilation shell-scripts to 'force' people to do it right), some issues with config can come up, and not everybody downloading and building from source is always well versed in programming in general, and some errors can be easier to help with if you know what went wrong.
For example can static_assert() be used to determine if you are compiling with C++11 and thus throw a compiler error when older (or newer) versions of C++ are used?

Something like this ;)

```
g++ -o mybin mysource.cpp
1>/Users/Jeroen/code/myapp/mysource.cpp(19): error C2338: compiling requires C++11, make sur
```

I am also thinking that if a .cpp file contains for Windows only and should not be included in Mac or Linux builds such an assertion can also be useful.

```
g++ -o mybin mysource.cpp
1>/Users/Jeroen/code/myapp/mysource.cpp(19): error C2338: This source file has been exclusiv
```

I mean, that will make sense to everybody, even those who know nothing at all about coding. An error saying certain windows only dependencies have not been found will only make sense to those who know their stuff. XD
(Of course, I am quite often surprised how many obvious things don't make sense to some people, but I guess 100% fool-proof doesn't exist as fools are so ingenious, but to most people, it should make sense).

And of course, I could also get on with all the flags you can give to the compiler...

nascardriver
February 19, 2019 at 7:29 am · Reply

You should CMake to do this. You can define which version of the standard you need and change compilation settings per OS (or deny compilation at all).
Most compilers predefine macros which could be used to detect the OS, but that's not standard.

nascardriver

January 4, 2019 at 5:58 am · Reply

Hi Alex!

Just a heads up that contracts will be added in C++20. This lesson and at least the lessons using the Fraction class should be updated.

Alex
January 6, 2019 at 1:29 pm · Reply

Yup. Quite a few lessons will be impacted by C++20. If the modules functionality makes it in (and I hope it does), then pretty much every lesson will be impacted!

Soham Kar
May 4, 2019 at 9:30 pm · Reply

Hey Alex,

I've heard a lot about modules being added to C++20, and I've used modules in Java and Python before, but I'm not clear on what the benefits of modules are vs the #include preprocessor directive. Do you have any thoughts regarding the matter or can you link to any resources regarding this?

nascardriver
May 4, 2019 at 11:40 pm · Reply

Hi Soham!

Headers are compiled in every file they're included in. Modules are compiled only once. That's faster and you no longer need to worry about include guards. Modules can't break each other by changing the import order. Headers were able to break each other by using #defines, since headers are entirely handled by the preprocessor.

Headers might expose entities that you intended only for internal use (In the situation of sharing libraries). Modules have full control over what is visible to the importer.

Including a header gives you everything from that header. Modules can be imported partially, so you only get what you need. This prevents cluttered autocomplete and potential name collisions.

This is what I remember, I haven't worked with modules yet. What I said might be wrong or incomplete.

Soham Kar
May 5, 2019 at 12:54 am · Reply

Thanks a lot!

Alan
November 2, 2018 at 12:14 am · Reply

Hi Alex,

I was writing a piece of code which states

```
1 // this function will print out your message in different colors.
```

```
// only blue, green, red, yellow and white are valid. If not specified, it is in white.
void colorPrint(std::string color = "white", std::string message = "")
{
    bool invalidColor = (color == "blue" || color == "green" || color == "red" || color == "
    assert(invalidColor && ("Unrecognized color: " + color).c_str());
...
```

While I was testing this piece of code by writing

```
colorPrint("pink", "this is some test message");
```

the console prints 'Assertion failed: invalidColor && ("Unrecognized color: " + color).c_str()'.

What I want to ask is that, is there a way I can make the assert message alter based on different parameter contents in the function call?

In this example, the ideal output on the console I would like to see is 'Assertion failed: invalidColor && "Unrecognized color: pink"'.

nascardriver
November 2, 2018 at 1:06 am · Reply

@assert is a compile-time macro, it's text content cannot be dynamically created.
You can throw an exception (Chapter 14) instead

```
void colorPrint(std::string color = "white", std::string message = {})
{
  bool validColor{ color == "blue" || color == "green" || color == "red" || color ==

  if (!validColor)
  {
    throw std::invalid_argument{ "Unrecognized color: " + color };
  }

  // ...
}
```

Alan
November 2, 2018 at 1:19 am · Reply

Thanks @nascardriver, that saves my day. I really appreciate it!

nascardriver
November 2, 2018 at 1:21 am · Reply

I updated my previous post to use @std::invalid_argument instead of @std::runtime_error

Davis
June 3, 2018 at 5:38 pm · Reply

Hey Alex,
Could you explain why #define NDEBUG has to be written before #include <cassert> ?
#define NDEBUG didn't work for my complier when I wrote it after #include <cassert>

nascardriver
June 4, 2018 at 6:26 am · Reply

Hi Davis!

When you include a file it's content is copied into the including file

```
#include <cassert>

int main() { }
```

turns into

```
/*
Contents of <cassert>, this contains something along the lines
#if !defined(NDEBUG)
#define assert ...
#endif
*/

int main() { }
```

Now, when you define NDEBUG after the include to <cassert>, the preprocessor has already checked whether or not it should enable asserts, that's why you need to define it before including <cassert>.

Davis
June 4, 2018 at 8:32 am · Reply

Oh. Makes perfect sense now. Thanks!

prince GUPTA
January 30, 2018 at 7:02 am · Reply

HI ALEX!
AS STATED IN 7.12a UNDER STATIC-ASSERT:
Because static_assert is not evaluated at runtime, static_assert statements can also be placed anywhere in the code file (even in global space).

WHY IS IT SO? AND IF THEY WERE EVALUATED AT RUNTIME WHAT PROBLEM MAY OCCUR PUTTING THEM IN GLOBAL SPACE..AND WHAT IS NOT BEEN FACED IN GLOBAL SPACE AS THEY ARE EVALAUTED AT COMPILE TIME?

Alex
January 31, 2018 at 4:28 pm · Reply

It is so because that's how it was designed: The goal is to give programmers a way to flag things that you expect to be true at compile time rather than at runtime.

For example, if your code has an assumption that integers are at least 4 bytes, previously you could assert this, but it wouldn't fail until you ran the program. With static_assert, you can now fail your program at compile time, saving you from compiling something that won't run anyway.

Code execution always starts at the top of main() and then continues from there into whatever subfunctions are called. At no point is code in the global space ever executed.

I don't understand your last question.

prince GUPTA
February 1, 2018 at 6:39 am · Reply

you said,"static_assert is evaluated at compile time so we can put it in global space comfortably". I want to ask what problem maybe caused if we put it in global space if it would evaluate at runtime.

nascardriver
February 1, 2018 at 6:46 am · Reply

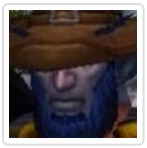
Code outside of functions won't be executed at runtime. All code needs to be in a function of some sort. static_assert can be outside of functions and is executed at compile time.
(There's also constexpr but I don't want to overcomplicate things)

prince GUPTA
February 2, 2018 at 12:35 pm · Reply

THANKS ! NASCARDRIVER
yes plz explain 'constexp' i saw it in 8.11 i don't know what does this keyword do?

nascardriver
February 3, 2018 at 3:53 am · Reply

constexpr values and functions are evaluated at compile-time.
constexpr value are covered in lesson 2.9, there's no tutorial for constexpr functions yet.
You don't need to worry about constexpr functions just yet, learn the basics first.



David
September 1, 2017 at 4:30 pm · Reply

Typo: "If the program calls GetArrayValue(-3),"
The function requires two arguments, but only one is provided. Also, the G is capitalized.

Alex
September 2, 2017 at 9:39 pm · Reply

Thanks, fixed!

CrazyL
July 29, 2017 at 2:43 am · Reply

Hello Alex,
the first time I tried "#define NDEBUG" I used it after "#include <cassert>" and wondered why it didn't switch off all assert() afterwards:

```
1
2  #include <iostream>
```

```
#include <cassert>    // includes assert.h that defines the assert()-macro, because NDEBUG

#define NDEBUG

int main()
{
    int i;
    std::cout << "Enter an int: ";
    std::cin >> i;

    assert((i > 0) && "i must be positive");    // assert()-macro was defined and is active
    return 0;
}
```

I guess that's a really stupid mistake to make, but to avoid confusion, it might be a good idea to change the snippet in "NDEBUG and other considerations" into the following lines:

```

#define NDEBUG
#include <cassert>

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

Anyway, thanks for the "to-the point" introduction of assert() and to use it effectively!

Aaron
August 3, 2018 at 8:42 pm · Reply

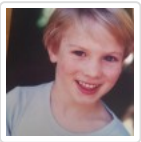
Hey there CrazyL!

This answer was answered already before. But you might not have seen it, so here it comes!

As said by nascardriver, when you include a file, the file is copied to the uncompiled code. The contents of cassert are something like this:

```
#if !defined(NDEBUG)
#define assert
//Insert assert here.
#endif
```

So, before you define NDEBUG, the if statement is false, and assert, is defined.
Don't credit me. Credit nascardriver.



James Ray
July 17, 2017 at 10:26 pm · Reply

"Exception handling is an advanced C++ topic, and we cover it in much detail in chapter 15 of this tutorial."

It's in chapter 14.

nikos-13
July 15, 2017 at 3:41 am · Reply

Could you explain me the terms of "production code" and "debug builds" ?

Alex
July 15, 2017 at 12:05 pm · Reply

This is covered in lesson 0.6a. Have a read there.

Curiosity
June 13, 2017 at 2:10 am · Reply

What Do You Mean By this? :-
Because static_assert is not evaluated at run-time, static_assert statements can also be placed anywhere in the code file (even in global space).

Plz Explain !
And YES, I found that non-const vars don't work in static assertions.

Curiosity
June 13, 2017 at 2:50 am · Reply

Oh Okay ! I saw that static_assert works in global space while assert doesn't.
But Now, My question is that why run-time variables etc. don't work in global space?

Alex
June 13, 2017 at 2:32 pm · Reply

Global and static variables get initialized when the program starts up. Then execution begins at the top of main and continued to the bottom. Code in global space would never get executed.

Alex
June 13, 2017 at 2:21 pm · Reply

When code is executed at runtime, the execution path starts at the top of main() and continued to the bottom of main() (branching out for called functions). So for normal asserts, those need to be put inside a function that gets called in order to be executed at runtime.

However, static_assert is evaluated by the compiler when it's compiling the program. This means static_assert can be placed anywhere in the program, even outside of functions, since it doesn't have to be within the path of execution.

Curiosity
June 13, 2017 at 10:50 pm · Reply

So It means that because compilation starts from the top of program(not from main()) that's why static assert is seen by it but because asserts are evaluated at run-time they are not seen as execution starts from the top of main()(not from the top of program)
Tell Me If I'm Right !

Alex
June 14, 2017 at 10:07 am · Reply

Yup.

C++ Learner
May 31, 2017 at 5:17 am · Reply

can you explain what do you mean by this sentence? :)
In C++11, a diagnostic message must be supplied as the second parameter(it is written just above "Exceptions")

Alex
May 31, 2017 at 1:56 pm · Reply

The diagnostic message is the second parameter provided to the static_assert function that gets printed if the condition isn't true. For example, "long must be 8 bytes" is a diagnostic message.

John
May 30, 2017 at 12:18 pm · Reply

Hi,
I'm running the code below, but I don't get the message "Assertion failed: index >= 0 && index <=9, file...", rather, the program just crashes with the usual message box showing up. Is something wrong in the code?

```
1  // Compiled and liked with MinGW[GCC 6.3.0] distro nuwen.net v14.1
2
3  #include <iostream>
4  #include <array>
5  #include <cassert> // for assert()
6
7  int getArrayValue(const std::array<int, 10> &array, int index)
8  {
9      // we're asserting that index is between 0 and 9
10     assert(index >= 0 && index <= 9); // this is line 6 in Test.cpp
11
12     return array[index];
13 }
14
15 int main()
16 {
17     std::array<int, 10> myArray={3,2,5,4,2,8,4,6,2,1};
18     getArrayValue(myArray, -3);
19
20 return 0;
21 }
```

Alex
May 30, 2017 at 3:43 pm · Reply

Nothing is wrong in the code. My best guess is that you're running your code with a release configuration, and that's setting NDEBUG, which turns assertions off. Try recompiling and running your code using a debug configuration instead.

John
May 31, 2017 at 9:14 pm · Reply

Ok, I found the problem, it's compiler specific. MinGW is not directing the the diagnostic message to std::cout, but to a pop-up window. To see this pop up window the code has to be compiled with the mwindow switch, i.e. g++ -mwindow main.cpp, then it works as shown in the example.