B.3 — Introduction to C++17

BY ALEX ON MARCH 31ST, 2018 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

What is C++17?

In September of 2017, the **ISO (International Organization for Standardization)** approved a new version of C++, called C++17. C++17 contains a fair amount of new content

New improvements in C++17

For your interest, here's a list of the major improvements that C++17 adds. Note that this list is not comprehensive, but rather intended to highlight some of the key improvements of interest.

- __has_include preprocessor identifier to check if optional header files are available (no tutorial yet)
- if statements that resolve at compile time (no tutorial yet)
- Initializers in if statements and switch statements (no tutorial yet)
- inline variables (**6.8 -- Global constants and inline variables**)
- Fold expressions (no tutorial yet)
- Nested namespaces can now be defined as namespace X::Y (**6.2 -- User-defined namespaces**)
- Removal of std::auto_ptr and some other deprecated types
- static_assert no longer requires a diagnostic text message parameter (**7.12a -- Assert and static_assert**)
- std::any (no tutorial yet)
- std::byte (no tutorial yet)
- std::filesystem (no tutorial yet)
- std::optional (no tutorial yet)
- std::shared_ptr can now manage C-style arrays (but std::make_shared can't create them yet) (**15.6 -- std::shared_ptr**)
- std::size (**6.2 -- Arrays (Part II)**)
- std::string_view (**6.6a -- An introduction to std::string_view**)
- Structured binding declarations (**7.4a -- Returning values by value, reference, and address**, but could use a full lesson)
- Template deduction for constructors (no tutorial yet)
- Trigraphs have been removed
- typename can now be used (instead of class) in a template template parameter
- UTF-8 (u8) character literals (no tutorial yet)

C.1 -- The end?

Index

B.2 -- Introduction to C++14

C++ TUTORIAL | PRINT THIS POST

27 comments to B.3 — Introduction to C++17

hellmet
February 4, 2020 at 12:13 pm · Reply

Hello again!

So, I've programmed pretty much so far in a single thread. Now, I'd like to wet my toes in threaded programming. I have an application that logs some stuff by printing to std::cerr, but I want to pull that out into a separate library. I know there are excellent alternatives and I don't need to reinvent the wheel, but this is a great learning experience, and a brush up on my OS course.

The idea is this. The main thread calls the log constructor to start the logging thread. At the end of main, just before return 0, it is Flush()ed and the destructor is called. (An explicit flush so that I don't throw exceptions in the destructor, hopefully). The main() starts some threads to process some stuff in parallel and joins them appropriately.

So I created a log class with a static thread instance, so that a new thread is spawned on first construction of the class. Subsequent calls to the constructor do nothing, and prompt the user that multiple constructions have taken place (i.e, a runtime error). This class has a push method, to push a message onto a queue. This is protected by a mutex lock. I also have a while loop in the logging thread that flushes the queue if it's not empty and suspends the thread (condition variables). So far, I'm hoping the sync right.

Each thread gets its own stack, so I can't catch the exception of one thread in another in the naïve way (the usual try-catch around std::thread t{} ... t.join()). Say in a thread, an exception was thrown. This exception propagates in that thread's stack. Just as I have a try..catch around main, I'll need to wrap these threads too, if I am to terminate it safely. Also, I am guaranteed to note all the messages generated by this thread just before it crashes, since .push() is synchronous, aren't I? I'm hoping the debugger can find exceptions in threaded scenarios too, but yeah, need to test it.

I hope threads are added here soon! I sure am looking forward to it! It's a rabbit hole in its own right!

PS: Are questions like this okay (long, drawn out)?

hellmet
February 5, 2020 at 11:13 am · Reply

Never having programmed complex threaded programs, I'm having trouble wrapping my head around thread sync. This is what I'm trying to do, as an exercise. Alternatively print odd and even numbers from alternate threads in a strictly alternate manner.

This is my code

```
1  #includes ...
2
3  std::mutex m;
4  std::condition_variable cv;
5
6  void PrintNum(const int thread_num) {
7
8      auto lock{ std::unique_lock(m, std::defer_lock) };
9
10     for (int i = thread_num; i < 20; i += 2) {
11         lock.lock();
12         std::cerr << "Thread [" << thread_num << "]: " << i << '\n';
13         lock.unlock();
14         cv.notify_one();
15         cv.wait(lock);
16     }
17 }
18
19 int main() {
20
21     // Assume t1 always starts first.. just assume for now.
22     std::thread t1(PrintNum, 0);
23     std::thread t2(PrintNum, 1);
24
25     t1.join();
26     t2.join();
27 }
```

This is my thought process. Thread t1 starts and wins on owning the lock (assume), meaning t2 is waiting to acquire the lock (blocked at lock.lock()). t1 continues, prints, unlocks, notifies the only waiting thread and stops as it is blocked in wait. In the meantime, t2 now acquires the lock, prints, unlocks and notifies the only waiting thread, t1, and the process repeats.

But I either seem to get a deadlock or an assertion failure from MSVC (unlock of unowned mutex).

I moved the unlock to the last line in the for-loop, and it works! But this makes 0 sense to me. Say I'm in a thread and just finished printing (meaning I own the lock), shouldn't waiting cause a deadlock?

nascardriver
February 6, 2020 at 9:05 am · Reply

I fail to see the question in your first comment, and I don't understand the question with the push. Can you try to shorten it?

Please don't remove the includes, post examples that can be compiled without modifications.

Both of your threads can be in line 14 at the same time (t1 unlocks and idles. t2 locks and unlocks, now both are unlocked). This leads to a deadlock.

My threading is too rusty, I couldn't solve this seemingly simple problem yet. If you solve it before me, please let me know how.

hellmet

February 6, 2020 at 9:50 am · Reply

The first comment, I meant to ask if my approach was correct. Forgot to put in there hahhe my bad.

About the push. I'm trying to write a logging library that logs asynchronously. The IO is done in a separate thread, and different threads push log messages into a queue owned by the logging thread. The logging thread keeps dumping these to the disk or wherever it's configured to. My question is, if a thread crashes, what is a non-intrusive way to catch that exception and send it to the logging thread? I was thinking of some constexpr functions that compile away to nothing in release mode, but some things (such as logging the line number and file name) are pushing me towards using macros.

Here's a pastebin of the second comment's code. https://pastebin.com/dHckgbiN

If both are unlocked, one or the other should win over the lock in the next line right?

I went back to my days in the OS course and did this problem in C, albeit with semaphores. There, it was more or less straightforward. C++ doesn't seem to have the concept of semaphores, and instead relies on condition_variable. This is from my reading so far.

If I solve it, I'll let you know! But I'm struggling to find decent resources to shed some light on C++ threading. Too bad most books don't cover this topic. And damn if they aren't expensive AF here in my place.

I'm really looking forward to seeing some tutorials here!

hellmet
February 7, 2020 at 7:33 am · Reply

I think I'm getting it.

This is the code that works, and then deadlocks at the end. I'll need to fix that last bit.

```cpp
#include <iostream>
#include <thread>
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;

void PrintNum(const int thread_num) {

    auto lock{ std::unique_lock(m) };

    for (int i = thread_num; i < 20; i += 2) {
        cv.notify_one();
        std::cerr << "Thread [" << thread_num << "]: " << i << '\n';
        cv.wait(lock);
    }
    // This works too (but use a deferred lock) like so
    // auto lock{ std::unique_lock(m, std::defer_lock) };
    for (int i = thread_num; i < 20; i += 2) {
        lock.lock();
        std::cerr << "Thread [" << thread_num << "]: " << i << '\n';
        cv.notify_one();
        cv.wait(lock);
        lock.unlock();
    }
}
```

```

int main() {

    std::cerr << "Started main\n";

    std::thread t1(PrintNum, 0);
    std::thread t2(PrintNum, 1);

    t1.join();
    t2.join();
}
```

This is how it goes. Say thread 0 starts and wins the lock, printing "Thread [0]: 0\n" to the terminal. Parallelly, thread2 is blocked waiting to get the lock. t1, after printing, notifies the other thread to wake up, but t2 doesn't wake up yet, as the lock is still held by t1. t1 goes to sleep by calling wait()*, releasing the lock after which t2 can actually acquire the lock, goes on to print, notifies the other, unlocks by calling wait. This dance of notifying one-another is the other and interleaved waits is the reason why we can print alternatively.

I'm yet to figure out how to get the last iteration fixed. When I do, I'll post my findings here.

* wait() releases a lock when a thread is put to sleep and reacquires the lock when woken up (atomically, duh, else there is no point of a 'mutex' :)).

nascardriver
February 7, 2020 at 8:16 am · Reply

> wait() releases a lock
That's what I was missing. I kept running into the need of notifying and waiting atomically.

> I was thinking of some constexpr functions that compile away to nothing in release mode
Doesn't sound like it makes sense. There will be `std::source_location` in C++20, until then, you have to use macros for file names/line numbers.

> if a thread crashes, what is a non-intrusive way to catch that exception and send it to the logging thread?
Without a `try` in the crashing thread, I don't know if there's a standard way of catching another thread's exception.

> Too bad most books don't cover this topic
If you're looking for performance improvement, look into execution policies. There will be async/await in C++20. That should be asynchronous programming a lot easier.

> I'm yet to figure out how to get the last iteration fixed
The last loop cycle calls `wait`, but is never notified. Only call `wait` if you're not in the last cycle (i < 19).

hellmet
February 7, 2020 at 9:17 am · Reply

> That's what I was missing.
Yeah me too! I was reading this amazing book here (it's free!, free as in free beer) [http://pages.cs.wisc.edu/~remzi/OSTEP/]. Perhaps it would be

nice to add this as a prerequisite (at the least, the concurrency part) if and when threads are added here. It really puts things into perspective.

> There will be `std::source_location` in C++20
Wow that's great! How does it work though? Does it require compiler support? From my cursory reading, it seems that this requires compiler support. For now, what if, I do this

```
namespace my_logger {/* ... */}
/* some code */
// Some useful defines
#define my_logger::get_line() __LINE__
#define my_logger::get_file() __FILE__
// ... and so on ?
```

Is that too dirty :D

> way of catching another thread's exception
Yeah. Other than using exception_ptr, which seems convoluted and intrusive to use. I was thinking, what if I write a macro that starts the thread with some code around it (basically a try-catch) when in debug mode, but the macro just boils down to the normal code when in release mode. Or, I could just ask the user of my library to set one of my functions as the terminate handler. The latter seems appropriate.

I also looked into std::terminate_handler and that seems to work, so I can try to print the thread that crashed in the log. But hmmm... how can I add more useful info. I could try reading the stack, but that becomes OS specific quickly (unless C++ has more tricks up its sleeve). But this has been really fun, but scary too. So much to learn, so little time.

> look into execution policies
Alright!

> Only call `wait` if you're not in the last cycle
Yeah that fixes it! I was thinking of some way to make it as general as possible, without this condition. Doesn't seem like it's going to be possible though. Also, I need to figure out a way to start the right thread first, always. Shouldn't be too hard now that the basics are falling into place. Thanks for the exchange!

nascardriver
February 8, 2020 at 1:23 am · Reply

> Does [std::source_location] require compiler support?
Yep. You'll have to wait for your standard library and compiler to support it. According to https://en.cppreference.com/w/cpp/compiler_support#C.2B.2B2a_library_features gcc libstdc++ is the only compiler/stdlib combo that supports it right now, and it's still experimental.

> Is that too dirty
It's dirty, but there's no way around it now. As a general recommendation, don't make your macros look like functions, they're not functions. If I assume your macro is a function because it looks like one, I could try to use it as a default argument, which won't do the same as a function.

> what if I write a macro
You like macros too much. You only need a macro to check for debug mode. If you have to use ugly code, hide it behind good code.

```
static constexpr bool k_bDebug
{
#if defined(NDEBUG)
  false
#else
  true
#endif
};
```

That's the only preprocess magic you need, the rest is pure C++.

I wrapped it in a class, use a function if you like to, the idea is the same.
Please excuse the formatting, the code was too much for clang-format.

```
class CSafeThread : public std::thread
{
public:
  template <class T, class... Parameters>
  CSafeThread(T fn, Parameters&&... args)
      : std::thread{
          k_bDebug ? std::thread{ [fn](Parameters&&... args)
            try
            {
              fn(std::forward<Parameters>(args)...);
            }
            catch (...)
            {
              std::cerr << "oops\n";
            }
          }, std::forward<Parameters>(args)... }
                  : std::thread{ fn, std::forward<Parameter
        }
  {
  }
};
```

We extend `std::thread`, making `CSafeThread` a drop-in replacement.
If we're in debug mode, we wrap the function that was passed to the constructor in another function. That function calls the original function, but guarded by a `try`.
If we're not in debug mode, we simply forward the arguments to `std::thread`.

Read up on parameter packs, the code is easier than it looks.

Since you're in debug mode, why don't you just use a debugger and/or core dumps?

hellmet
February 8, 2020 at 7:05 am · Reply

Yep, I'm looking at that page almost everyday! I can't wait to get my hands on the latest clang release! Boy most of the header stuff will just disappear! It'll make code more pythonic while maintaining performance of C++. I can't wait!

> You like macros too much. As a general recommendation ... don't!
Hahaha, perhaps... I was just playing around with it and yeah, it's misleading AF. I tripped myself with that haha.

> CSafeThread
How about member access then? I'll probably need to add a .get() method to return a reference to the thread so that normal std::thread methods can be called.

nascardriver
February 8, 2020 at 7:31 am

`CSafeThread` _is a_ `std::thread`, you don't need to add anything, everything is there.

hellmet
February 8, 2020 at 7:54 am

Ohh right! Inheritance! Doh forgot that was there. My bad!

Thanks!

hellmet
December 31, 2019 at 6:36 am · Reply

Hello Alex and Nascardriver! I have an unrelated doubt.

In cmake, I have a directory structure like so

```
Sources
├──Render
│    ├──Debug
│    ├──Device
│    ├──Instance
│    ├──Queues
│    ├──Surface
│    ├──SwapChain
│    └──VulkanInclude
└──Window
```

Adding a cmake file for each of the subfolders and adding libraries for each (ex: in each of Device, Instance etc...) and linking them into a RenderLib would be one way. The other way is to just glob all of the files under Render to build a single RenderLib. Which is a better approach? I was thinking, with the first approach, compilation would be streamlined, but then again, cmake knows which files change and can compile only those units required. So I don't think there is any performance advantage in glob vs lib-per-module. What would you suggest?

nascardriver
December 31, 2019 at 6:45 am · Reply

If you're writing the files in Render/ (ie. they're still changing), compile them (Either in a separate CMakeLists or in one for the entire project). If Render/ is a project on its own, compile it to a library link to it.
As you said, the files will only be compiled once either way. If Render/ is a project on its own and you copy the files to a different project to use them, you might end up with several copies of the source files each with some minor differences you made along the way.

hellmet
December 31, 2019 at 7:03 am · Reply

Yeah I'm writing all of them. It's not a project itself though, but I could make it one, so it's reusable in anything else I make... Hmm... For now, I'm not making it a project by itself. So I thought, I'll just glob all the files under Render and export a RenderLib. That's fine right?

nascardriver
December 31, 2019 at 7:04 am · Reply

That's fine

Geneva S.Flores
November 13, 2019 at 3:12 am · Reply

Good

Simon Allais
February 10, 2019 at 9:30 am · Reply

There is also `std::variant` which is a C++17 template class representing type-safe union (useful when you simply want to use a "sum" type).

yoyos
January 3, 2019 at 11:00 pm · Reply

what will happen in C++20 and more? wil they continue to create new versions and keep people learning?

nascardriver
January 4, 2019 at 5:31 am · Reply

Hopefully, yes.
You can see what's likely to be added to C++20 on wikipedia
https://en.wikipedia.org/wiki/C%2B%2B20

Soumya
November 23, 2018 at 8:42 pm · Reply

Hi Alex,

I am a great fan of your website. But recently I saw in reddit that they discourage your site for learning C++ [https://www.reddit.com/r/learnprogramming/wiki/index#wiki_discouraged_resources].

Actually I am worried and need your response to this. It would cleanse doubts in me and many others.

Thanks.

nascardriver
November 24, 2018 at 7:12 am · Reply

Hi Soumya!

This has been discussed on said subreddit several times, I couldn't find a statement of a wiki maintainer.

Summing it up:
Some people don't like that Alex uses "wrong" names to explain certain topics. Though, throwing technical words and descriptions at a beginner is discouraging. As long as there's a transfer to the proper names once the basics are set, I don't see a problem.
Some people don't like the order of the chapters/lessons, others do like the current order.
There were and still are inconsistencies across all lessons, because the lessons weren't all written on the same day obviously. Alex fixes these inconsistencies once they are pointed out. One of the complaints was about @std::endl and '\n' being mixed. This has been addressed by Alex and mostly, if not entirely, been fixed.
You'll have to go through a lot of lessons before you get to the topics that separate C and C++. Again, this can be considered good or bad.

My opinion (I haven't read the lessons (apart from 2 or 3), I just reply to comments and give suggestions to learners):
Learncpp is dated. Alex appears to be busy irl, so I can't blame him for not yet having lessons about C++14, 17 and 20. But there is too few content about C++11, and the content that is covered falls short compared to the pre C++11 content. eg. 9 lessons about C-style arrays and only 3 about @std::vector and @std::array, @std::string not being properly discussed until chapter 17!
A lot of quizzes involve writing algorithms, but Alex rarely mentions the @std functions that could be used instead (eg. @std::sort, @std::accumulate, @std::end, etc.). The @std library is too important to be left out.
The first couple of lessons teach bad practice, some of which stretches throughout the entire tutorials (eg. std::endl and not using uniform initialization). All lessons should use the recommended features. The discouraged features should be discussed briefly, stating the difference to their alternatives and why they shouldn't be used.
Adding rules but not following them in the tutorials causes readers the forget about the rules and don't follow them either.

Despite these problem, I think of learncpp as the best place to learn C++.
learncpp is actively maintained, questions get answered and errors get fixed. There are quizzes every couple of lessons which help understanding the lessons. C++ is a difficult language and Alex does a great job explaining it.

Resources
-
https://www.reddit.com/r/learnprogramming/comments/4wbeg5/is_learncppcom_still_considered_bad/
-
https://www.reddit.com/r/learnprogramming/comments/6z8en4/why_is_learncppcom_a_discouraged_resource/
-
https://www.reddit.com/r/learnprogramming/comments/3ncs7q/c_learncppcom_has_been_updated_is_it_a_reliable/

Piyush
January 1, 2019 at 6:52 am · Reply

The active forum is the major cause of my love to this website. :)

Rahul
September 6, 2019 at 1:14 am · Reply

Things have been explained in a simple and easy way at this site. I have referred this website to many interns and young graduates in my team for a while, each one has come back that now they understand C++ better. I highly recommend this website to get foundation of C++ fundamentals right.

P-z
October 3, 2018 at 8:51 am · Reply

2nd line:

ISO (International Organization for Standardization)

Shouldn't Organization and Standardization be flipped? Or IOS instead of ISO?

nascardriver
October 3, 2018 at 9:01 am · Reply

"ISO is not an acronym. The organization adopted ISO as its abbreviated name in reference to the Greek word isos (ίσος, meaning "equal"),[4] as its name in the three official languages would have different acronyms. "
https://en.wikipedia.org/wiki/International_Organization_for_Standardization#Name_and_abbreviations

nascardriver
May 1, 2018 at 6:20 am · Reply

"Nested namespaces can now be defined as namespace X:Y"
is missing a colon
"Nested namespaces can now be defined as namespace X::Y"

Alex
May 6, 2018 at 9:11 pm · Reply

Fixed. Thanks for pointing that out!