BARTŁOMIEJ FILIPEK

C++17 IN DETAIL

LEARN THE EXCITING FEATURES OF THE NEW C++ STANDARD!

(BF)
C++ STORIES

BFILIPEK.COM

C++17 in Detail

Learn the Exciting Features of The New C++ Standard!

Bartłomiej Filipek

This book is for sale at http://leanpub.com/cpp17indetail

This version was published on 2019-09-12

for Wiola and Mikołaj

Contents

About the Author

Bartłomiej (Bartek) Filipek is a C++ software developer with more than 12 years of professional experience. In 2010 he graduated from Jagiellonian University in Cracow, Poland with a Masters Degree in Computer Science.

Bartek currently works at Xara, where he develops features for advanced document editors. He also has experience with desktop graphics applications, game development, large-scale systems for aviation, writing graphics drivers and even biofeedback. In the past, Bartek has also taught programming (mostly game and graphics programming courses) at local universities in Cracow.

Since 2011 Bartek has been regularly blogging at bfilipek.com. Initially, the topics revolved around graphics programming, but now the blog focuses on core C++. He's also a co-organiser of the C++ User Group in Cracow. You can hear Bartek in one @CppCast episode where he talks about C++17, blogging and text processing.

Since October 2018, Bartek has been a C++ Expert for the Polish National Body which works directly with ISO/IEC JTC 1/SC 22 (C++ Standardisation Committee). In the same month, Bartek was awarded his first MVP title for the years 2019/2020 by Microsoft.

In his spare time, he loves collecting and assembling Lego models with his little son.

Technical Reviewer

Jacek Galowicz is a Software Engineer with roughly a decade of professional experience in C++. He got his master of science degree in electrical engineering at RWTH Aachen University in Germany.

Jacek co-founded the Cyberus Technology GmbH in early 2017 and works on products around low-level cybersecurity, virtualization, microkernels, and advanced testing infrastructure. At former jobs, he implemented performance- and security-sensitive microkernel operating systems for Intel x86 virtualization at Intel and FireEye in Germany. In general, he gained experience with kernel driver development, 3D graphics programming, databases, network communication, physics simulation, mostly in C or C++.

In his free time, Jacek maintains a little C++ blog, which has seen some lack of love while he wrote the C++17 STL Cookbook. He is a regular visitor of the C++ Usergroups in Hannover and Braunschweig. In order to do meta programming and generic programming better, he also learned and uses Haskell, which in turn sparked his interest to generally bring the advantages of purely functional programming to C++.

Additional Reviewers & Supporters

Without the support of many good people, this book would have been far less than it is. It is a great pleasure to thank them. A lot of people read drafts, found errors, pointed out confusing explanations, suggested different wording, tested programs, and offered support and encouragement. Many reviewers generously supplied insights and comments that I was able to incorporate into the book. Any mistakes that remain are, of course, my own.

Thanks especially to the following reviewers, who either commented on large sections of the book, smaller parts or gave me a general direction for the whole project.

Patrice Roy - Patrice has been playing with C++, either professionally, for pleasure or (most of the time) both for over 20 years. After a few years doing R&D and working on military flight simulators, he moved on to academics and has been teaching computer science since 1998. Since 2005, he's been involved more specifically in helping graduate students and professionals from the fields of real-time systems and game programming develop the skills they need to face today's challenges. The rapid evolution of C++ in recent years has made his job even more enjoyable.

Jonathan Boccara - Jonathan is a passionate C++ developer working on large codebase of financial software. His interests revolve around making code expressive. He created and regularly blogs on Fluent C++[1], where he explores how to use the C++ language to write expressive code, make existing code clearer, and also about how to keep your spirits up when facing unclear code.

Łukasz Rachwalski - Software engineer - founder C++ User Group Krakow.

Michał Czaja - C++ software developer and network engineer. Works in telecommunication industry since 2009.

Arne Mertz - Software Engineer from Hamburg, Germany. He is a C++ and clean code enthusiast. He's the author of the Simplify C++[2] blog.

JFT - Has been involved with computer programming and "computer techy stuff" for over 45 years - including OS development and teaching c++ in the mid 1990's.

[1]https://www.fluentcpp.com/
[2]https://arne-mertz.de/

Victor Ciura - Senior Software Engineer at CAPHYON and Technical Lead on the Advanced Installer team[3]. For over a decade, he designed and implemented several core components and libraries of Advanced Installer. He's a regular guest at Computer Science Department of his Alma Mater, University of Craiova, where he gives student lectures & workshops on "Using C++STL for Competitive Programming and Software Development". Currently, he spends most of his time working with his team on improving and extending the repackaging and virtualization technologies in Advanced Installer IDE, helping clients migrate their Win32 desktop apps to the Windows Store (MSIX).

Karol Gasiński - Tech Lead on macOS VR in Apple's GPU SW Architecture Team. Previously Senior Graphics Software Engineer at Intel. As a member of KHRONOS group, he contributed to OpenGL 4.4 and OpenGL ES 3.1 Specifications. Except for his work, Karol's biggest passion is game development and low-level programming. He conducts research in the field of VR, new interfaces and game engines. In game industry is known from founding WGK conference. In the past, he was working on mobile versions of such games as Medieval Total War, Pro Evolution Soccer and Silent Hill.

Marco Arena - Software Engineer and C++ Specialist building mission critical and high performance software products in the Formula 1 Industry. Marco is very active in the C++ ecosystem as a blogger, speaker and organizer: he founded the Italian C++ Community in 2013, he joined isocpp.org[4] editors in 2014 and he has been involved in community activities for many years. Marco has held the Microsoft MVP Award since 2016. Discover more at marcoarena.com[5].

Konrad Jaśkowiec - C++ expert with almost 8 years of professional experience at the time with prime focus on system design and optimization. You can find his profile at Linkedin[6].

[3]http://www.advancedinstaller.com
[4]http://isocpp.org/
[5]http://marcoarena.com/
[6]https://www.linkedin.com/in/konrad-ja%C5%9Bkowiec-84585159/

Daniel Khoshnoudirad - a passionate C++ Developer. Daniel graduated in 2016 with the PhD in Computer Science from Université Paris-Est, under the direction of Pr. Hugues Talbot. Daniel also holds a Master's degree in Mathematical Engineering from Université de Bordeaux. He is a proud reviewer of the French version of Effective Modern C++ by Dr Scott Meyers. He has experience in Qt, Python, Fortran, Machine Learning and teaching. Daniel is also interested in Java, Rust, JavaScript, HTML, networks, and many other technologies. You can follow him @DanielKhoshnoud[7]

Rob Stewart - started programming in high school on a Commodore VIC-20. He taught himself BASIC, 6502 machine code, Forth, C, C++, JavaScript, Python, and other programming languages. (He also took a course on Fortran in college.) He has been using C++ for 30 years for things like cockpit simulators, commercial real estate tools, web browsing accelerators, computer desktop alternatives, financial trading, network communications, and more, while working for the US Air Force, startups, and Susquehanna International Group. He actually writes documentation (!) for his own libraries and has helped with the documentation for numerous Boost libraries. He has helped with several well-known C++ books. He has taught C++ classes and mentored many developers. He was a founding member of the Boost Steering Committee. He and his wife of 33 years have nine children.

[7]https://twitter.com/DanielKhoshnoud

Revision History

- 10th August 2018 - the first release!
- 31st August 2018 - new sections: nested namespaces,using statement in pack expansion
  - An example of `std::visit` with multiple variants and about `overloaded`
  - Improved "Code Contracts With nodiscard and "Refactoring with optional"
- 28th September 2018, New chapters String Conversions and Searchers
  - Added notes about `gcd, lcm, clamp` in the Other STL Changes Chapter
- 3rd October 2018 - hot fixes and clarifications in String Conversions
- 4th November 2018 - Parallel Algorithms was rewritten and is 3X larger, new examples and descriptions
- 21st December 2018 - New chapter - How to Parallelise CSV Reader
- 18th January 2019 - the book is 99% ready!
  - Filesystem chapter was rewritten and is 5X larger, new examples and descriptions
- 1st February 2019 - additions to Filesystem
- 15th February 2019 - updates in "Structured Binding" and "if constexpr"
- 1st March 2019 - the book is 100% ready!
  - Added `scoped_lock`, `std::iterator` deprecation and polymorphic memory allocator sections
- 21st June 2019 - book format, foreword by Herb Sutter, smaller updates
  - Changes book format from 21.6 x 27.9cm (US Letter) into 17.8 x 23.1cm (Technical)
  - common code style, add code titles in most of the places
  - updated Constexpr Lambda, added capturing `[*this]`
- 9th August 2019 - wording, layout for print version, extracted deprecated Lib features chapter, notes for GCC 9.1
- 7th September 2019 - print version ready! improved section about dynamic memory allocation.

Foreword

If you've ever asked "what's in C++17 and what does it mean for me and my code?" — and I hope you have — then this book is for you.

Now that the C++ standard is being released regularly every three years, one of the challenges we have as a community is learning and absorbing the new features that are being regularly added to the standard language and library. That means not only knowing what those features are, but also how to use them effectively to solve problems. Bartlomiej Filipek does a great job of this by not just listing the features, but explaining each of them with examples, including a whole Part 3 of the book about how to apply key new C++17 features to modernize and improve existing code — everything from upgrading `enable_if` to the new `if constexpr`, to refactoring code by applying the new `optional` and `variant` vocabulary types, to writing parallel code using the new standard parallel algorithms. In each case, the result is cleaner code that's often also significantly faster too.

The point of new features isn't just to know about them for their own sake, but to know about how they can let us express our intent more clearly and directly than ever in our C++ code. That ability to directly "say what we mean" to express our intent, or to express "what" we want to achieve rather than sometimes-tortuous details of "how" to achieve it through indirect mechanisms, is the primary thing that determines how clean and writable and readable — and correct — our code will be. For C++ programmers working on real-world projects using reasonably up-to-date C++ compilers, C++17 is where it's at in the industry today for writing robust production code. Knowing what's in C++17 and how to use it well is an important tool that will elevate your day-to-day coding, and more likely than not reduce your day-to-day maintenance and debugging chores.

If you're one of the many who have enjoyed Barteks's blog (bfilipek.com, frequently cited at isocpp.org), you'll certainly also enjoy this entertaining and informative book. And if you haven't enjoyed his blog yet, you should check it out too… and then enjoy the book.

Herb Sutter, herbsutter.com

Preface

After the long-awaited C++11, the C++ Committee has made changes to the standardisation process, and we can now expect a new language standard every three years. In 2014 the ISO Committee delivered C++14. Now it's time for C++17, which was published at the end of 2017. As I am writing these words, in the middle of 2019, the C++20 draft is feature ready and prepared for the final review process.

As you can see, the language and the Standard Library evolves quite fast! Since 2011 you've got a set of new library modules and language features every three years. Thus, staying up to date with the whole state of the language has become quite a challenging task, and that is why this book will help you.

This book describes all the significant changes in C++17 and will give you the essential knowledge to stay current with the latest features. What's more, each section contains lots of practical examples and also compiler-specific notes to provide you with a more comfortable start.

It's a pleasure for me to write about new and exciting things in the language and I hope you'll have fun discovering C++17 as well!

Best regards,

Bartek

About the Book

C++11 was a major update for the language. With all the modern features like lambdas, constexpr, variadic templates, threading, range-based for loops, smart pointers and many more powerful elements, it signalled enormous progress for the language. Even now, in 2019, many teams struggle to modernise their projects to leverage all the modern features. Later there was a minor update - C++14, which improved some things from the previous Standard and added a few smaller elements.

Although C++17 is not as big as C++11, it's larger than C++14 and brings many exciting additions and improvements. And this book will guide through all of them!

The book brings you exclusive content about C++17 and draws from the experience of many articles that have appeared at bfilipek.com. The material was rewritten from the ground-up and updated with the latest information. All of that equipped with lots of new examples and practical tips. Additionally, the book provides insight into the current implementation status, compiler support, performance issues and other relevant knowledge to boost your current projects.

Who This Book is For

This book is intended for all C++ developers who have at least essential experience with C++11/14.

The principal aim of the book is to equip you with practical knowledge about C++17. After reading the book, you'll be able to move past C++11 and leverage the latest C++ techniques in your day to day tasks.

Please don't worry if you're not an expert in C++11/14. This book provides the necessary background, so you'll get the information in a proper context.

Overall Structure of the Book

C++17 brings a lot of changes to the language and the Standard Library. In this book, all the new features were categorised into a few segments, so that they are easier to comprehend.

As a result, the book has the following sections:

- Part One - C++17 Language Features
  – Fixes and Deprecation
  – Language Clarification
  – General Language Features
  – Templates
  – Attributes
- Part Two - C++17 The Standard Library
  – std::optional
  – std::variant
  – std::any
  – std::string_view
  – String Operations
  – Filesystem
  – Parallel STL
  – Other Changes
- Part Three - More Examples and Use Cases
  – Refactoring with std::optional and std::variant
  – Enforcing Code Contracts With [[nodiscard]]
  – Replacing enable_if with ifconstexpr
  – How to Parallelise CSV Reader
- Appendix A - Compiler Support
- Appendix B - Resources and Links

Part One, about the language features, is shorter and will give you a quick run over the most significant changes. You can read it in any order you like.

Part Two, describes a set of new types and utilities that were added to the Standard Library. The helper types create a potential new vocabulary for C++ code: like when you use `optional`, `any`, `variant` or `string_view`. And what's more, you have new powerful capabilities, especially in the form of parallel algorithms and the standard filesystem. A lot of examples in this part will use many other features from C++17.

Part Three brings together all of the changes in the language and shows examples where a lot of new features are used alongside. You'll see discussions about refactoring, simplifying code with new template techniques or working with parallel STL and the filesystem. While the first and the second part can also be used as a reference for individual changes, the third part shows more of larger C++17 patterns that join many features.

A considerable advantage of the book is the fact that with each new feature you'll get information about the compiler support and the current implementation status. That way you'll be able to check if a particular version of the most popular compilers (MSVC, GCC or Clang) implements it or not. The book also gives practical hints on how to apply new techniques in your current codebase.

Reader Feedback

If you spot an error, a typo, a grammar mistake… or anything else (especially logical issues!) that should be corrected, then please send your feedback to bartlomiej.filipek AT bfilipek.com.

You can also use those two places to leave your feedback:

- Leanpub Book's Feedback Page[8]
- GoodReads Book's Page[9]

Example Code

You can find the ZIP package with all the example on the book's website:

cppindetail.com/data/cpp17indetail.zip[10]

The same ZIP package should also be attached with the ebook.

[8]https://leanpub.com/cpp17indetail/feedback
[9]https://www.goodreads.com/book/show/41447221-c-17-in-detail
[10]https://www.cppindetail.com/data/cpp17indetail.zip

Many examples in the book are relatively short. You can copy and paste the lines into your favourite compiler/IDE and then run the code snippet.

Code License

The code for the book is available under the Creative Commons License.

Compiling

To use C++17 make sure you provide a proper flag for your compiler:

- for GCC (at least 7.1 or 8.0 or newer): use `-std=c++17` or `-std=c++2a`
- for Clang (at least 4.0 or newer): use `-std=c++17` or `-std=c++2a`
- for MSVC (Visual Studio 2017 or newer): use `/std:c++17` or `/std:c++latest` in `project options -> C/C++ -> Language -> C++ Language Standard`

Formatting

The code is presented in a monospace font, similarly to the following example:

For longer examples with a corresponding `cpp` file:

ChapterABC/example_one.cpp

```
#include <iostream>

int main() {
    std::string text = "Hello World";
    std::cout << text << '\n';
}
```

Or shorter snippets (without a corresponding file):

```
int foo() {
    return std::clamp(100, 1000, 1001);
}
```

Snippets of longer programs were usually shortened to present only the core mechanics. In that case, you'll find their full version in the separate ZIP package that comes with the book.

The corresponding file for the code snippet is mentioned in the title above the frame:

`Chapter ABC/example_one.cpp`

Usually, source code uses full type names with namespaces, like `std::string`, `std::clamp`, `std::pmr`. However, to make code compact and present it nicely on a book page the namespaces sometimes might be removed, so they don't use space. Also, to avoid line wrapping, longer lines might be manually split into two. In some case, the code in the book might skip `include` statements.

Syntax Highlighting Limitations

The current version of the book might show some Pygments syntax highlighting limitations.

For example:

- `if constexpr` - Link to Pygments issue: #1432 - C++ if constexpr not recognized (C++17)[11]
- The first method of a class is not highlighted - #1084 - First method of class not highlighted in C++[12]
- Template method is not highlighted #1434 - C++ lexer doesn't recognize function if return type is templated[13]
- Modern C++ attributes are sometimes not recognised properly

Other issues for C++ and Pygments: issues C++[14].

Online Compilers

Instead of creating local projects to play with the code samples, you can also leverage online compilers. They offer a basic text editor and usually allow you to compile only one source file (the code that you edit). They are convenient if you want to play with code samples and check the results using various compilers.

For example, many of the code samples for this book were created using Coliru Online and Wandbox compilers and then adapted for the book.

[11]https://bitbucket.org/birkenfeld/pygments-main/issues/1432/c-if-constexpr-not-recognized-c-17
[12]https://bitbucket.org/birkenfeld/pygments-main/issues/1084/first-method-of-class-not-highlighted-in-c
[13]https://bitbucket.org/birkenfeld/pygments-main/issues/1434/c-lexer-doesnt-recognize-function-if
[14]https://bitbucket.org/birkenfeld/pygments-main/issues?q=c%2B%2B

Here's a list of some of the useful services:

- Coliru[15] - uses GCC 8.2.0 (as of July 2019), offers link sharing and a basic text editor, it's simple but very effective.
- Wandbox[16] - offers a lot of compilers, including most Clang and GCC versions, can use boost libraries; offers link sharing and multiple file compilation.
- Compiler Explorer[17] - offers many compilers, shows compiler output, can execute the code.
- CppBench[18] - runs simple C++ performance tests (using google benchmark library).
- C++ Insights[19] - a Clang-based tool for source to source transformation. It shows how the compiler sees the code, for example by expanding lambdas, auto, structured bindings or range-based for loops.

There's also a helpful list of online compilers gathered on this website: List of Online C++ Compilers[20].

[15]http://coliru.stacked-crooked.com/
[16]https://wandbox.org/
[17]https://gcc.godbolt.org/
[18]http://quick-bench.com/
[19]https://cppinsights.io/
[20]https://arnemertz.github.io/online-compilers/

Part 1 - Language Features

We can say that C++ comes in two parts: The Language and The Standard Library. The first element, The Language, focuses on the expressive code and conscience syntax. The second element gives you tools, utilities and algorithms. For example, in C++11, we got lambdas that simplified writing short function objects. C++14 allowed ‘auto‘ type deduction for function return types which also shorten code and simplified templated code.

C++17, as a major update to the Standard, brings many amazing language elements that generally, make the language clearer and more straightforward. For instance, you can reduce the need to use `enable_if` and tag dispatching techniques by leveraging `if constexpr`. You can treat tuples like first class language citizens thanks to structured bindings, rely on and understand expression evaluation order, write code that naturally uses copy-elision mechanism, and much, much more!

In this part you’ll learn:

- What was removed from the language and what is deprecated
- How the language is more precise: for example, thanks to expression evaluation order guarantees
- What are new features of templates: like `if constexpr` or fold expressions
- What are the new standard attributes
- How you can write cleaner and more expressive code thanks to structured binding, inline variables, compile-time if or template argument deduction for classes

1. Quick Start

To make you more curious about the new Standard, below, there are a few code samples that present combined language features.

Don't worry if you find the examples confusing or too complicated. All the new features are individually explained in depth in the coming chapters.

Working With Maps

Part I/demo_map.cpp

```
 1  #include <iostream>
 2  #include <map>
 3
 4  int main() {
 5      std::map<std::string, int> mapUsersAge { { "Alex", 45 }, { "John", 25 } };
 6
 7      std::map mapCopy{mapUsersAge};
 8
 9      if (auto [iter, wasAdded] = mapCopy.insert_or_assign("John", 26); !wasAdded)
10          std::cout << iter->first << " reassigned...\n";
11
12      for (const auto& [key, value] : mapCopy)
13          std::cout << key << ", " << value << '\n';
14  }
```

The code will output:

```
John reassigned...
Alex, 45
John, 26
```

The above example uses the following features:

- Line 7: Template Argument Deduction for Class Templates - `mapCopy` type is deduced from the type of `mapUsersAge`. No need to declare `std::map<std::string, int> mapCopy{...}`.

- Line 9: New inserting member function for maps - `insert_or_assign`.
- Line 9: Structured Bindings - captures a returned pair from `insert_or_assign` into separate names.
- Line 9: init if statement - `iter` and `wasAdded` are visible only in the scope of the surrounding `if` statement.
- Line 12: Structured Bindings inside a range-based for loop - we can iterate using `key` and `value` rather than `pair.first` and `pair.second`.

Debug Printing

Part I/demo_print.cpp

```cpp
#include <iostream>

template<typename T> void linePrinter(const T& x)  {
    if constexpr (std::is_integral_v<T>)
        std::cout << "num: " << x << '\n';
    else if constexpr (std::is_floating_point_v<T>) {
        const auto frac = x - static_cast<long>(x);
        std::cout << "flt: " << x << ", frac " << frac << '\n';
    }
    else if constexpr(std::is_pointer_v<T>) {
        std::cout << "ptr, ";
        linePrinter(*x);
    }
    else
        std::cout << x << '\n';
}

template<typename ... Args> void printWithInfo(Args ... args) {
    (linePrinter(args), ...); // fold expression over the comma operator
}

int main () {
    int i = 10;
    float f = 2.56f;
    printWithInfo(&i, &f, 30);
}
```

The code will output:

```
ptr, num: 10
ptr, flt: 2.56, frac 0.56
num: 30
```

Here you can see the following features:

- Line 4, 6, 10: `if constexpr` - to discard code at compile-time, used to match the template parameter.
- Line 4, 6, 10: `_v` variable templates for type traits - no need to write `std::trait_-name<T>::value`.
- Line 19: Fold Expressions inside `printWithInfo`. This feature simplifies variadic templates. In the example, we invoke `linePrinter()` over all input arguments.

Let's Start!

The code you've seen so far is just the tip of the iceberg! Continue reading to see much more: fixes in the language, clarifications, removed things (like `auto_ptr`), and of course the new stuff: `constexpr` lambda, `if constexpr`, fold expressions, structured bindings, `template<auto>`, inline variables, template argument deduction for class templates and much more!

2. Removed or Fixed Language Features

The C++17 Standard contains over 1600 pages, growing over 200 pages compared to C++14[1]! Fortunately, the language specification was cleaned up in a few places, and some old or potentially harmful features could be cleared out. This short chapter lists several language elements that were removed or fixed. See the Removed And Deprecated Library Features Chapter for a list of changes in the Standard Library.

In this chapter, you'll learn:

- What was removed from the language like the `register` keyword or `operator++` for `bool`
- What was fixed, notably the auto type deduction with brace initialisation.
- Other improvements like for `static_assert` and range-based for loops.

[1]For example the draft from July 2017 N4687 compared to C++14 draft N4140

Removed Elements

One of the core concepts behind each iteration of C++ is its compatibility with previous versions. We'd like to have new things in the language, but at the same time, our old projects should still compile. However, sometimes, there's a chance to remove parts that are wrong or rarely used.

This section briefly explains what was removed from the Standard.

Removing the `register` Keyword

The `register` keyword was deprecated in 2011 (C++11), and it has had no meaning since then. It was removed in C++17. The keyword is reserved and might be repurposed in future revisions of the Standard (for example `auto` keyword was reused and now is an entirely new and powerful feature).

If you use `register` to declare a variable:

```
register int a;
```

You might get the following warning (GCC8.1 below)

```
warning: ISO C++17 does not allow 'register' storage class specifier
```

or error in Clang (Clang 7.0)

```
error: ISO C++17 does not allow 'register' storage class specifier
```

Extra Info

The change was proposed in: P0001R1[2].

[2] https://wg21.link/p0001r1

Removing Deprecated `operator++(bool)`

The increment operator for `bool` has been already deprecated for a very long time! The committee recommended against its use back in 1998 (C++98), but they only now finally agreed to remove it from the language. Note that `operator--` was never enabled for `bool`.

If you try to write the following code:

```
bool b;
b++;
```

You should get a similar error like this from GCC (GCC 8.1):

```
error: use of an operand of type 'bool' in 'operator++' is forbidden in C++17
```

Extra Info

The change was proposed in: P0002R1[3].

Removing Deprecated Exception Specifications

In C++17, exception specification will be part of the type system (as discussed in the next chapter about Language Clarification). However, the standard contains old and deprecated exception specification that appeared to be impractical and unused.

For example:

```
void fooThrowsInt(int a) throw(int) {
   printf_s("can throw ints\n");
   if (a == 0)
      throw 1;
}
```

Pay special attention to that `throw(int)` part.

The above code has been deprecated since C++11. The only practical exception declaration is `throw()` which means - this code won't throw anything. Since C++11 it's been advised to use `noexcept`.

[3] https://wg21.link/p0002r1

For example in clang 4.0 you'll get the following error:

```
error: ISO C++1z does not allow dynamic exception specifications
[-Wdynamic-exception-spec] note: use 'noexcept(false)' instead
```

Extra Info

The change was proposed in: P0003R5[4].

Removing Trigraphs

Trigraphs are special character sequences that could be used when a system doesn't support 7-bit ASCII (like ISO 646). For example `??=` generated `#`, `??-` produced ~. (All of C++'s basic source character set fits in 7-bit ASCII). Today, trigraphs are rarely used, and by removing them from the translation phase, the compilation process can be more straightforward. See a table below with all the trighraps that were declared until C++17:

Trigraph	Replacement
??=	#
??(	[
??<	{
??/	\
??)	]
??>	}
??'	ˆ
??!	|
??-	~

Extra Info

The change was proposed in: N4086[5].

[4]http://wg21.link/p0003r5
[5]https://wg21.link/n4086

Fixes

We can argue what is a fix in a language standard and what is not. Below there are three things that might look like a fix for something that was missing or not working in the previous rules.

New auto rules for direct-list-initialisation

Since C++11 there's been a strange problem where:

```
auto x { 1 };
```

Is deduced as `std::initializer_list<int>`. Such behaviour is not intuitive as in most cases you should expect it to work like `int x { 1 };`.

Brace initialisation is the preferred pattern in modern C++, but such exceptions make the feature weaker.

With the new Standard, we can fix this so that it will deduce `int`.

To make this happen, we need to understand two ways of initialisation - copy and direct:

```
// foo() is a function that returns some Type by value
auto x = foo();   // copy-initialisation
auto x{foo()};    // direct-initialisation, initializes an
                  // initializer_list (until C++17)

int x = foo();    // copy-initialisation
int x{foo()};     // direct-initialisation
```

For the direct initialisation, C++17 introduces new rules:

- For a braced-init-list with a single element, auto deduction will deduce from that entry.
- For a braced-init-list with more than one element, auto deduction will be ill-formed.

For example:

```
auto x1 = { 1, 2 };   // decltype(x1) is std::initializer_list<int>
auto x2 = { 1, 2.0 }; // error: cannot deduce element type
auto x3{ 1, 2 };      // error: not a single element
auto x4 = { 3 };      // decltype(x4) is std::initializer_list<int>
auto x5{ 3 };         // decltype(x5) is int
```

Extra Info

The change was proposed in: N3922[6] and N3681[7]. The compilers fixed this issue quite early, as the improvement is available in GCC 5.0 (Mid 2015), Clang 3.8 (Early 2016) and MSVC 2015 (Mid 2015). Much earlier than C++17 was approved.

`static_assert` With no Message

This feature adds a new overload for `static_assert`. It enables you to have the condition inside `static_assert` without passing the message.

It will be compatible with other asserts like `BOOST_STATIC_ASSERT`. Programmers with boost experience will now have no trouble switching to C++17 `static_assert`.

```
static_assert(std::is_arithmetic_v<T>, "T must be arithmetic");
static_assert(std::is_arithmetic_v<T>); // no message needed since C++17
```

In many cases, the condition you check is expressive enough and doesn't need to be mentioned in the message string.

Extra Info

The change was proposed in: N3928[8].

[6]http://wg21.link/n3922
[7]http://wg21.link/n3681
[8]https://wg21.link/n3928

Different `begin` and `end` Types in Range-Based For Loop

C++11 added range-based for loops:

```
for (for-range-declaration : for-range-initializer)
    statement;
```

According to the C++14 standard that loop is equivalent to the following code:

```
auto && __range = for-range-initializer;
for ( auto __begin = begin-expr, __end = end-expr;
          __begin != __end;
          ++__begin ) {
    for-range-declaration = *__begin;
    statement;
}
```

As you can see, `__begin` and `__end` have the same type. This works nicely but is not scalable enough. For example, you might want to iterate until some sentinel value with a different type than the start of the range.

In C++17 range-based for loops are defined as equivalent to the following code:

```
auto && __range = for-range-initializer;
auto __begin = begin-expr;
auto __end = end-expr;
for ( ; __begin != __end; ++__begin ) {
    for-range-declaration = *__begin;
    statement;
}
```

The types of `__begin` and `__end` might be different; only the comparison operator is required. That change has no effect on existing for loops but it provides more options for libraries. For example, this little change allows Range TS (and Ranges in C++20) to work with the range-based for loop.

Extra Info

The change was proposed in: P0184R0[9].

[9] https://wg21.link/p0184r0

Compiler Support

Feature	GCC	Clang	MSVC
Removing `register` keyword	7.0	3.8	VS 2017 15.3
Remove Deprecated `operator++`(bool)	7.0	3.8	VS 2017 15.3
Removing Deprecated Exception Specifications	7.0	4.0	VS 2017 15.5
Removing trigraphs	5.1	3.5	VS 2010
New auto rules for direct-list-initialisation	5.0	3.8	VS 2015
`static_assert` with no message	6.0	2.5	VS 2017
Different begin and end types in range-based for	6.0	3.6	VS 2017

3. Language Clarification

C++ is a challenging language to learn and fully understand, and some parts might be confusing for programmers. One of the reasons for the lack of clarity might be the freedom given to the implementation/compiler. For example, some parts of the language are left vague to allow for more aggressive optimisations. Other difficulties can arise from the requirement to be compatible with C. C++17 addresses some of the most common "holes" in the language.

In this chapter, you'll learn:

- What Evaluation Order is and why it might generate unexpected results
- Copy elision guarantees in the language
- Exceptions specifications as part of the type system
- Memory allocations for (over)aligned data

Stricter Expression Evaluation Order

Until C++17 the language hasn't specified any evaluation order for function parameters. **Period**.

For example, that's why in C++14 `make_unique` is not just syntactic sugar, but it guarantees memory safety:

Consider the following examples:

```
foo(unique_ptr<T>(new T), otherFunction()); // first case
```

And with explicit `new`.

```
foo(make_unique<T>(), otherFunction()); // second case
```

Considering the first case, in C++14, we only know that `new T` is guaranteed to happen before the `unique_ptr` construction, but that's all. For example, `new T` might be called first, then `otherFunction()`, and then the constructor `unique_ptr` is invoked.

For such evaluation order, when `otherFunction()` throws, then `new T` generates a leak (as the unique pointer is not yet created).

When you use `make_unique`, as in the second case, the leak is not possible as you wrap memory allocation and creation of unique pointer in one call.

C++17 addresses the issue shown in the first case. Now, the evaluation order of function arguments is "practical" and predictable. In our example, the compiler won't be allowed to call `otherFunction()` before the expression `unique_ptr<T>(new T)` is fully evaluated.

The Changes

In an expression:

```
f(a, b, c);
```

The order of evaluation of `a`, `b`, `c` is still unspecified, but any parameter is fully evaluated before the next one is started. It's especially crucial for complex expressions like this:

```
f(a(x), b, c(y));
```

If the compiler chooses to evaluate `a(x)` first, then it must evaluate `x` before processing `b`, `c(y)` or `y`.

This guarantee fixes the problem with `make_unique` vs `unique_ptr<T>(new T())`. A given function argument must be fully evaluated before other arguments are evaluated.

Consider the following case:

Chapter Clarification/chain_order.cpp

```cpp
#include <iostream>

class Query {
public:
    Query& addInt(int i) {
        std::cout << "addInt: " << i << '\n';
        return *this;
    }

    Query& addFloat(float f) {
        std::cout << "addFloat: " << f << '\n';
        return *this;
    }
};

float computeFloat() {
    std::cout << "computing float... \n";
    return 10.1f;
}

float computeInt() {
    std::cout << "computing int... \n";
    return 8;
}

int main() {
  Query q;
  q.addFloat(computeFloat()).addInt(computeInt());
}
```

You probably expect that using C++14 `computeInt()` happens after `addFloat`. Unfortunately, that might not be the case. For instance here's an output from GCC 4.7.3:

```
computing int...
computing float...
addFloat: 10.1
addInt: 8
```

The chaining of functions is already specified to work from left to right (thus `addInt()` happens after `addFloat()`), but the order of evaluation of the inner expressions can differ. To be precise:

> The expressions are indeterminately sequenced with respect to each other.

With C++17, function chaining will work as expected when they contain inner expressions, i.e., they are evaluated from left to right:

In the expression:

```
a(expA).b(expB).c(expC)
```

`expA` is evaluated before calling `b()`.

Compiling the previous example with a conformant C++17 compiler, yields the following result:

```
computing float...
addFloat: 10.1
computing int...
addInt: 8
```

Another result of this change is that when using operator overloading, the order of evaluation is determined by the order associated with the corresponding built-in operator.

For example:

```
std::cout << a() << b() << c();
```

The above code contains operator overloading and expands to the following function notation:

```
operator<<(operator<<(operator<<(std::cout, a()), b()), c());
```

Before C++17, `a()`, `b()` and `c()` could be evaluated in any order. Now, in C++17, `a()` will be evaluated first, then `b()` and then `c()`.

Here are more rules described in the paper P0145R3[1]:

> the following expressions are evaluated in the order a, then b:
>
> 1. `a.b`
> 2. `a->b`
> 3. `a->*b`
> 4. `a(b1, b2, b3) // b1, b2, b3 - in any order`
> 5. `b @= a // '@' means any operator`
> 6. `a[b]`
> 7. `a << b`
> 8. `a >> b`

If you're not sure how your code might be evaluated, then it's better to make it simple and split it into several clear statements. You can find some guides in the Core C++ Guidelines, for example ES.44[2] and ES.44[3].

Extra Info

The change was proposed in: P0145R3[4].

[1]https://wg21.link/p0145r3
[2]http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-order
[3]http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es44-dont-depend-on-order-of-evaluation-of-function-arguments
[4]https://wg21.link/p0145r3

Guaranteed Copy Elision

Copy Elision is a common optimisation that avoids creating unnecessary temporary objects.

For example:

Chapter Clarification/copy_elision.cpp

```
#include <iostream>

struct Test {
    Test() { std::cout << "Test::Test\n"; }
    Test(const Test&) { std::cout << "Test(const Test&)\n"; }
    Test(Test&&) { std::cout << "Test(Test&&)\n"; }
    ~Test() { std::cout << "~Test\n"; }
};

Test Create() {
    return Test();
}

int main() {
    auto n = Create();
}
```

In the above call, you might assume a temporary copy is used - to store the return value of `Create`. In C++14, most compilers recognise that the temporary object can be optimised easily, and they can create `n` "directly" from the call of `Create()`. So you'll probably see the following output:

```
Test::Test // create n
~Test // destroy n when main finishes
```

In its basic form, the copy elision optimisation is called Return Value Optimisation (**RVO**).

As an experiment, in GCC you can add a compiler flag `-fno-elide-constructors` and use `-std=c++14` (or some earlier language standard). In that case you'll see a different output:

```
// compiled as "g++ CopyElision.cpp -std=c++14 -fno-elide-constructors"
Test::Test
Test(Test&&)
~Test
Test(Test&&)
~Test
~Test
```

In this case, we have two extra copies that the compiler uses to pass the return value into `n`;

Compilers are even smarter, and they can elide in cases when you return a named object - it's called **Named Return Value Optimisation - NRVO**:

```
Test Create() {
    Test t;
    // several instruction to initialize 't'...
    return t;
}

auto n = Create(); // temporary will be usually elided
```

Currently, the Standard allows eliding in cases like:

- When a temporary object is used to initialise another object (including the object returned by a function, or the exception object created by a throw-expression)
- When a variable that is about to go out of scope is returned or thrown
- When an exception is caught by value

However, it's up to the compiler/implementation to elide or not. In practice, all the constructors' definitions are required.

With C++17, we get clear rules on when elision has to happen, and thus constructors might be entirely omitted. In fact, instead of eliding the copies the compiler defers the "materialisation" of an object.

Why might this be useful?

- To allow returning objects that are not movable/copyable - because we could now skip copy/move constructors
- To improve code portability since every conformant compiler supports the same rule

- To support the *“return by value”* pattern rather than using output arguments
- To improve performance

Below you can see an example with a non-movable/non-copyable type, based on P0135R0:

Chapter Clarification/copy_elision_non_moveable.cpp

```
struct NonMoveable {
    NonMoveable(int x) : v(x) { }
    NonMoveable(const NonMoveable&) = delete;
    NonMoveable(NonMoveable&&) = delete;

    std::array<int, 1024> arr;
    int v;
};

NonMoveable make(int val) {
    if (val > 0)
        return NonMoveable(val);

    return NonMoveable(-val);
}

int main() {
    auto largeNonMoveableObj = make(90); // construct the object
    return largeNonMoveableObj.v;
}
```

The above code wouldn’t compile under C++14 as it lacks copy and move constructors. But with C++17 the constructors are not required - because the object `largeNonMovableObj` will be constructed in place.

Please notice that you can also use many return statements in one function and copy elision will still work.

Moreover, it’s important to remember, that in C++17 copy elision works only for unnamed temporary objects, and Named RVO is not mandatory.

To understand how mandatory copy elision/deferred temporary materialisation is defined in the C++ Standard, we must understand value categories which are covered in the next section.

Updated Value Categories

In C++98/03, we had two basic categories of expressions:

- `lvalue` - an expression that can appear on the left-hand side of an assignment
- `rvalue` - an expression that can appear only on the right-hand side of an assignment

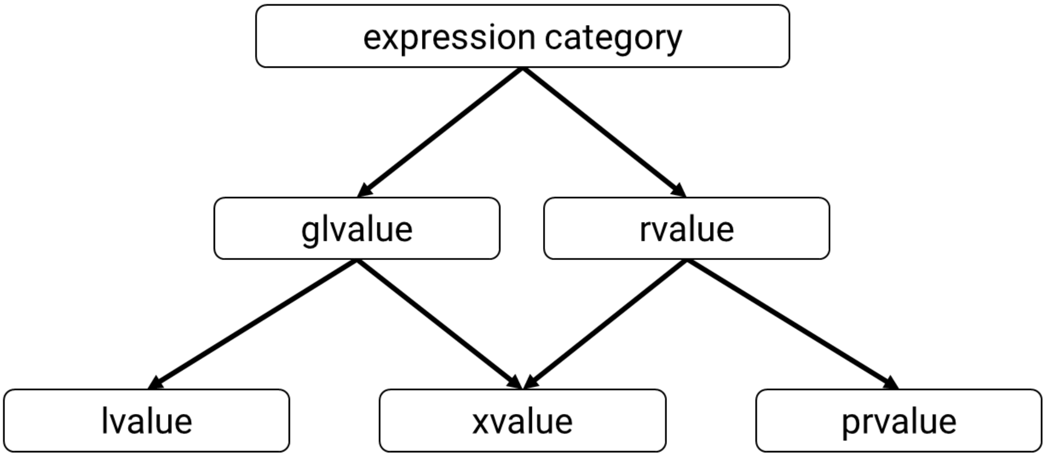
C++11 extended this taxonomy (due to the move semantics), with three more categories:

- `xvalue` - an eXpiring `lvalue`
- `prvalue` - a pure `rvalue`, an `xvalue`, a temporary object or subobject, or a value that is not associated with an object.
- `glvalue` - a generalised `lvalue`, which is an `lvalue` or an `xvalue`

Examples:

```
std::string str;
str;            // lvalue
42;             // prvalue
str + "10"      // prvalue
std::move(str); // xvalue
```

Here's a diagram that shows how the categories are related:



Value Categories

There are three core categories (below with colloquial "definitions"):

- `lvalue` - an expression that has an identity, and which we can take the address of
- `xvalue` - "eXpiring `lvalue`" - an object that we can move from, which we can reuse. Usually, its lifetime ends soon
- `prvalue` - pure `rvalue` - something without a name, which we cannot take the address of, we can move from such expression

To support Copy Elision, the authors of the proposal provided the updated definitions of `glvalue` and `prvalue`. From the Standard[5]:

> - `glvalue` - A `glvalue` is an expression whose evaluation computes the location of an object, bit-field, or function
> - `prvalue` - A `prvalue` is an expression whose evaluation initialises an object, bit-field, or operand of an operator, as specified by the context in which it appears

For example:

```
class X { int a; };
X{10}   // this expression is prvalue
X x;    // x is lvalue
x.a     // it's lvalue (location)
```

In short: `prvalues` perform initialisation, `glvalues` describe locations. The C++17 Standard specifies that when there's a `prvalue` initialising some `glvalue`, then there's no need to create a temporary and we can defer its materialisation.

In C++17 Copy Elision/Deferred Temporary Materialization happens when:

- in initialisation of an object from a `prvalue`: `Type t = T()`
- in a function call where the function returns a `prvalue` - like in our examples.

Extra Info

The change was proposed in: P0135R0[6](reasoning) - and P0135R1[7](wording).

[5]https://timsong-cpp.github.io/cppwp/n4659/basic.lval
[6]https://wg21.link/p0135r0
[7]https://wg21.link/p0135r1

Dynamic Memory Allocation for Over-Aligned Data

Embedded environments, kernel, drivers, game development and other areas might require a non-default alignment for memory allocations. Complying those requirements might improve the performance or satisfy some hardware interface.

For example, to perform geometric data processing using SIMD[8] instructions, you might need 16-byte or 32-byte alignment for a structure that holds 3D coordinates:

```
struct alignas(32) Vec3d { // alignas is available since C++11
    double x, y, z;
};
auto pVectors = new Vec3d[1000];
```

`Vec3d` holds `double` fields, and usually, its natural alignment should be 8 bytes. Now, with `alignas` keyword, we change this alignment to 32. This approach allows the compiler to fit the objects into SIMD registers like AVX (256-bit-wide registers).

Unfortunately, in C++11/14, you have no guarantee how the memory will be aligned after `new[]`. Often, you have to use routines like `std::aligned_alloc()` or MSVC's `_aligned_malloc()` to be sure the alignment is preserved. That's not ideal as it's not working easily with smart pointers and also makes memory management visible in the code.

C++17 fixes that hole by introducing new memory allocation function overloads for `new()` and `delete()` with the `align_val_t` parameter. Example function signatures below[9]:

```
void* operator new(size_t, align_val_t);
void operator delete(void*, size_t, align_val_t);
```

The Standard also defines `__STDCPP_DEFAULT_NEW_ALIGNMENT__` macro that specifies the default alignment for dynamic memory allocations. On common platforms, Clang, GCC and MSVC specify it as 16 bytes.

Now, in C++17, when you allocate:

```
auto pVectors = new Vec3d[1000];
```

[8]Single Instruction, Multiple Data, for example, SSE2, AVX, see https://en.wikipedia.org/wiki/Streaming_SIMD_Extensions
[9]See all 22 `new()` overloads at https://en.cppreference.com/w/cpp/memory/new/operator_new

The alignment for `Vec3d` is larger than `__STDCPP_DEFAULT_NEW_ALIGNMENT__`, and thus the compiler will select the overloads with the `align_val_t` parameter.

In Clang and GCC you can control the default alignment by using the `fnew-alignment` switch (see Clang's documentation[10]). The MSVC compiler exposes the `/Zc:alignedNew`[11] flag that turns the feature on or off.

We can also provide custom implementation, have a look:

Chapter Clarification/aligned_new.cpp

```
void* operator new(std::size_t size, std::align_val_t align) {
#if defined(_WIN32) || defined(__CYGWIN__)
    auto ptr = _aligned_malloc(size, static_cast<std::size_t>(align));
#else
    auto ptr = std::aligned_alloc(static_cast<std::size_t>(align), size);
#endif

    if (!ptr) throw std::bad_alloc{};

    std::cout << "new: " << size << ", align: "
              << static_cast<std::size_t>(align) << ", ptr: " << ptr << '\n';

    return ptr;
}

void operator delete(void* ptr, std::size_t size, std::align_val_t algn) noexcept {
    std::cout << "delete: " << size << ", align: "
              << static_cast<std::size_t>(algn) << ", ptr : " << ptr << '\n';
#if defined(_WIN32) || defined(__CYGWIN__)
    _aligned_free(ptr);
#else
    std::free(ptr);
#endif
}

void operator delete(void* ptr, std::align_val_t algn) noexcept { ... } // hidden
```

The code uses `_aligned_malloc()` and `_aligned_free()` for the Windows version[12]. It's because the Windows platform uses different allocation mechanisms for over-aligned

[10]https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fnew-alignment
[11]https://docs.microsoft.com/en-us/cpp/build/reference/zc-alignednew?view=vs-2019
[12]this applies to MSVC, MinGW, Clang on Windows or Cygwin.

data, and that's why `std::free()` wouldn't release the memory correctly. On other platforms that conform with C11 you can try using `std::aligned_alloc()`, as since C++17 the Standard is based on the C11 specification. In that context `free()` can delete the aligned memory.

The new functionality can significantly improve the code, and you can now hold the aligned objects in standard containers without writing custom allocators, or custom deleters for smart pointers.

For example:

Chapter Clarification/aligned_new.cpp

```
std::vector<Vec3d> vec;
vec.push_back({});
vec.push_back({});
vec.push_back({});
assert(reinterpret_cast<uintptr_t>(vec.data()) % alignof(Vec3d) == 0);
```

When executed, our replaced allocation functions might log the following output:

```
new: 32, align: 32, ptr: 000001F866625960
new: 64, align: 32, ptr: 000001F866625680
delete: 32, align: 32, ptr : 000001F866625960
new: 96, align: 32, ptr: 000001F866623EA0
delete: 64, align: 32, ptr : 000001F866625680
delete: 96, align: 32, ptr : 000001F866623EA0
```

The example allocates memory for a single entry, then deletes it and increases the size for the vector twice to make space for all three elements. At the end we check the pointer alignment to be sure it's aligned to 32 bytes.

You can read more about the experiments with the new functionality at New new() - The C++17's Alignment Parameter for Operator new()[13]. This blog post also shows the dangerous side when you try to ask for a non-standard alignment with placement new.

Extra Info

The change was proposed in: P0035[14].

[13]https://www.bfilipek.com/2019/08/newnew-align.html
[14]http://wg21.link/p0035

Exception Specifications in the Type System

Exception Specification for a function didn't use to belong to the type of the function, but now it will be part of it. You can now have two function overloads: one with `noexcept` and the second without it. See below:

Chapter Clarification/func_except_type.cpp

```
using TNoexceptVoidFunc = void (*)() noexcept;
void SimpleNoexceptCall(TNoexceptVoidFunc f) { f(); }

using TVoidFunc = void (*)();
void SimpleCall(TVoidFunc f) { f(); }

void fNoexcept() noexcept { }
void fRegular() { }

int main() {
    SimpleNoexceptCall(fNoexcept);
    //SimpleNoexceptCall(fRegular); // cannot convert

    SimpleCall(fNoexcept); // converts to regular function
    SimpleCall(fRegular);
}
```

A pointer to `noexcept` function can be converted to a pointer to a regular function (this also works for a pointer to a member function). But it's not possible the other way around (from a regular function pointer into a function pointer that is marked with `noexcept`).

One of the reasons for adding the feature is a chance to optimise the code better. If the compiler has a guarantee that a function won't throw, then it can generate faster code.

Also, as described in the previous chapter about Language Fixes, in C++17, the Exception Specification is cleaned up. Effectively, you can only use the `noexcept` specifier[15] for declaring that a function won't throw.

Extra Info

The change was proposed in: P0012R1[16].

[15]http://en.cppreference.com/w/cpp/language/noexcept_spec
[16]http://wg21.link/p0012r1

Compiler Support

Feature	GCC	Clang	MSVC
Stricter expression evaluation order	7.0	4.0	VS 2017
Guaranteed copy elision	7.0	4.0	VS 2017 15.6
Dynamic memory allocation for over-aligned data	7.0	4.0	VS 2017 15.5
Exception specifications part of the type system	7.0	4.0	VS 2017 15.5

4. General Language Features

In this section of the book, we'll look at widespread improvements to the language that have the potential to make your code more compact and expressive. A perfect example of such a general feature is structured binding. Using that feature, you can leverage a comfortable syntax for tuples (and tuple-like expressions). Something easy in other languages like Python is now possible with C++17.

In this chapter, you'll learn:

- Structured bindings/Decomposition declarations
- How to provide Structured Binding interface for your custom classes
- Init-statement for if/switch
- Inline variables and their impact on header-only libraries
- Lambda expressions that might be used in a `constexpr` context
- How to properly wrap the `this` pointer in lambda expressions
- Simplified use of nested namespaces
- How to test for header existence with `__has_include` directive

Structured Binding Declarations

Do you often work with tuples or pairs?

If not, then you should probably start looking into those handy types. Tuples enable you to bundle data ad-hoc with excellent library support instead of creating small custom types. The language features like structured binding make the code even more expressive and concise.

Consider a function that returns two results in a pair:

```
std::pair<int, bool> InsertElement(int el) { ... }
```

You can write:

```
auto ret = InsertElement(...)
```

And then refer to `ret.first` or `ret.second`. However, referring to values as `.first` or `.second` is also not expressive - you can easily confuse the names, and it's hard to read. Alternatively you can leverage `std::tie` which will unpack the tuple/pair into local variables:

```
int index { 0 };
bool flag { false };
std::tie(index, flag) = InsertElement(10);
```

Such code might be useful when you work with `std::set::insert` which returns `std::pair`:

```
std::set<int> mySet;
std::set<int>::iterator iter;
bool inserted { false };

std::tie(iter, inserted) = mySet.insert(10);

if (inserted)
    std::cout << "Value was inserted\n";
```

As you see, such a simple pattern - returning several values from a function - requires several lines of code. Fortunately, C++17 makes it much simpler!

With C++17 you can write thw following:

```
std::set<int> mySet;

auto [iter, inserted] = mySet.insert(10);
```

Now, instead of `pair.first` and `pair.second`, you can use variables with concrete names. In addition, you have one line instead of three, and the code is easier to read. The code is also safer as `iter` and `inserted` are initialised in the expression.

Such syntax is called a *structured binding expression*.

The Syntax

The basic syntax for structured bindings is as follows:

```
auto [a, b, c, ...] = expression;
auto [a, b, c, ...] { expression };
auto [a, b, c, ...] ( expression );
```

The compiler introduces all identifiers from the `a, b, c, ...` list as names in the surrounding scope and binds them to sub-objects or elements of the object denoted by expression.

Behind the scenes, the compiler might generate the following **pseudo code**:

```
auto tempTuple = expression;
using a = tempTuple.first;
using b = tempTuple.second;
using c = tempTuple.third;
```

Conceptually, the expression is copied into a tuple-like object (`tempTuple`) with member variables that are exposed through `a`, `b` and `c`. However, the variables `a`, `b` and `c` are not references; they are aliases (or bindings) to the generated object member variables. The temporary object has a unique name assigned by the compiler.

For example:

```
std::pair a(0, 1.0f);
auto [x, y] = a;
```

`x` binds to `int` stored in the generated object that is a copy of `a`. And similarly, `y` binds to `float`.

Modifiers

Several modifiers can be used with structured bindings:

`const` modifiers:

```
const auto [a, b, c, ...] = expression;
```

References:

```
auto& [a, b, c, ...] = expression;
auto&& [a, b, c, ...] = expression;
```

For example:

```
std::pair a(0, 1.0f);
auto& [x, y] = a;
x = 10;  // write access
// a.first is now 10
```

In the example, `x` binds to the element in the generated object, that is a reference to `a`.

Now it's also quite easy to get a reference to a tuple member:

```
auto& [ refA, refB, refC, refD ] = myTuple;
```

You can also add `[[attribute]]` to structured bindings:

```
[[maybe_unused]] auto& [a, b, c, ...] = expression;
```

Structured Bindings or Decomposition Declaration?

You might have seen another name used for this feature: "decomposition declaration". During the standardisation process, both names were considered, but "structured bindings" was selected.

Structured Binding Limitations

There are several limitations related to structured bindings. They cannot be declared as `static` or `constexpr` and also they cannot be used in lambda captures. For example:

```
constexpr auto [x, y] = std::pair(0, 0);
// generates:
error: structured binding declaration cannot be 'constexpr'
```

It was also unclear about the linkage of the bindings. Compilers had a free choice to implement it (and thus some of them might allow capturing a structured binding in lambdas). Fortunately, those limitations might be removed due to C++20 proposal (already accepted): P1091: Extending structured bindings to be more like variable declarations[1].

Binding

Structured Binding is not only limited to tuples, we have three cases from which we can bind from:

1. If the initializer is an array:

```
// works with arrays:
double myArray[3] = { 1.0, 2.0, 3.0 };
auto [a, b, c] = myArray;
```

In this case, an array is copied into a temporary object, and `a`, `b` and `c` refers to copied elements from the array.

The number of identifiers must match the number of elements in the array.

2. If the initializer supports `std::tuple_size<>`, provides `get<N>()` and also exposes `std::tuple_element` functions:

```
std::pair myPair(0, 1.0f);
auto [a, b] = myPair; // binds myPair.first/second
```

In the above snippet, we bind to `myPair`. But this also means that you can provide support for your classes, assuming you add `get<N>` interface implementation. See an example in the later section.

[1] https://wg21.link/P1091

3. If the initialiser's type contains only non-static data members:

```
struct Point  {
    double x;
    double y;
};

Point GetStartPoint() {
    return { 0.0, 0.0 };
}

const auto [x, y] = GetStartPoint();
```

`x` and `y` refer to `Point::x` and `Point::y` from the `Point` structure.

The class doesn't have to be `POD`, but the number of identifiers must equal to the number of non-static data members. The members must also be accessible from the given context.

Note: In C++17, initially, you could use structured bindings to bind to class members as long as they were public. That could be a problem when you wanted to access such members in a context of friend functions, or even inside a struct implementation. This issue was recognised quickly as a defect, and it's now fixed in C++17. See P0969R0[2].

Examples

This section will show you a few examples where structured bindings are helpful. In the first one, we'll use them to write more expressive code, and in the next one, you'll see how to provide API for your class to support structured bindings.

Expressive Code With Structured Bindings

If you have a map of elements, you might know that internally they are stored as pairs of `<const Key, ValueType>`.

[2] https://wg21.link/P0969R0

Now, when you iterate through elements of that map:

```
for (const auto& elem : myMap) { ... }
```

You need to write `elem.first` and `elem.second` to refer to the key and value. One of the **coolest use cases** of structured binding is that we can use it inside a range based for loop:

```
std::map<KeyType, ValueType> myMap;
// C++14:
for (const auto& elem : myMap) {
    // elem.first - is velu key
    // elem.second - is the value
}
// C++17:
for (const auto& [key,val] : myMap) {
    // use key/value directly
}
```

In the above example, we bind to a pair of `[key, val]` so we can use those names in the loop. Before C++17 you had to operate on an iterator from the map - which returns a pair `<first, second>`. Using the real names `key/value` is more expressive.

The above technique can be used in:

Chapter General Language Features/city_map_iterate.cpp

```
#include <map>
#include <iostream>
#include <string>

int main() {
    const std::map<std::string, int> mapCityPopulation {
        { "Beijing", 21'707'000 },
        { "London", 8'787'892 },
        { "New York", 8'622'698 }
    };

    for (auto&[city, population] : mapCityPopulation)
        std::cout << city << ": " << population << '\n';
}
```

In the loop body, you can safely use `city` and `population` variables.

Providing Structured Binding Interface for Custom Class

As mentioned earlier, you can provide Structured Binding support for a custom class.

To do that you have to define `get<N>`, `std::tuple_size` and `std::tuple_element` specialisations for your type.

For example, if you have a class with three members, but you'd like to expose only its public interface:

Chapter Chapter General Language Features/custom_structured_bindings.cpp

```cpp
class UserEntry {
public:
    void Load() { }

    std::string GetName() const { return name; }
    unsigned GetAge() const { return age; }
private:
    std::string name;
    unsigned age { 0 };
    size_t cacheEntry { 0 }; // not exposed
};
```

The interface for Structured Bindings:

Chapter Chapter General Language Features/custom_structured_bindings.cpp

```cpp
// with if constexpr:
template <size_t I> auto get(const UserEntry& u) {
    if constexpr (I == 0) return u.GetName();
    else if constexpr (I == 1) return u.GetAge();
}

namespace std {
    template <> struct tuple_size<UserEntry> : integral_constant<size_t, 2> { };

    template <> struct tuple_element<0,UserEntry> { using type = std::string; };
    template <> struct tuple_element<1,UserEntry> { using type = unsigned; };
}
```

`tuple_size` specifies how many fields are available, `tuple_element` defines the type for a specific element and `get<N>` returns the values.

Alternatively, you can also use explicit `get<>` specialisations rather than `if constexpr`:

```
template<> string get<0>(const UserEntry &u)  { return u.GetName(); }
template<> unsigned get<1>(const UserEntry &u) { return u.GetAge(); }
```

For a lot of types, writing two (or several) functions might be more straightforward than using `if constexpr`.

Now you can use `UserEntry` in a structured binding, for example:

```
UserEntry u;
u.Load();
auto [name, age] = u; // read access
std:: cout << name << ", " << age << '\n';
```

This example only allows read access of the class. If you want write access, then the class should also provide accessors that return references to members. Later you have to implement `get` with references support.

The code in this section used `if constexpr`, you can read more about this powerful feature in the next chapter: Templates: `if constexpr`.

Extra Info

The change was proposed in: P0217[3](wording), P0144[4](reasoning and examples), P0615[5](renaming “decomposition declaration” with “structured binding declaration”).

[3]https://wg21.link/p0217
[4]https://wg21.link/p0144
[5]https://wg21.link/p0615

Init Statement for `if` and `switch`

C++17 provides new versions of the if and switch statements:

```
if (init; condition)
```

And

```
switch (init; condition)
```

In the `init` section you can specify a new variable, similarly to the init section in for loop. Then check the variable in the `condition` section. The variable is visible only in `if`/`else` scope.

To achieve a similar result, before C++17, you had to write:

```
{
    auto val = GetValue();
    if (condition(val))
        // on success
    else
        // on false...
}
```

Please notice that `val` has a separate scope, without that it 'leaks' to enclosing scope.

Now, in C++17, you can write:

```
if (auto val = GetValue(); condition(val))
    // on success
else
    // on false...
```

Now, `val` is visible only inside the `if` and `else` statements, so it doesn't 'leak.' `condition` might be any boolean condition.

Why is this useful?

Let's say you want to search for a few things in a string:

```
const std::string myString = "My Hello World Wow";

const auto pos = myString.find("Hello");
if (pos != std::string::npos)
    std::cout << pos << " Hello\n"

const auto pos2 = myString.find("World");
if (pos2 != std::string::npos)
    std::cout << pos2 << " World\n"
```

You have to use different names for `pos` or enclose it with a separate scope:

```
{
    const auto pos = myString.find("Hello");
    if (pos != std::string::npos)
        std::cout << pos << " Hello\n"
}

{
    const auto pos = myString.find("World");
    if (pos != std::string::npos)
        std::cout << pos << " World\n"
}
```

The new if statement will make that additional scope in one line:

```
if (const auto pos = myString.find("Hello"); pos != std::string::npos)
    std::cout << pos << " Hello\n";

if (const auto pos = myString.find("World"); pos != std::string::npos)
    std::cout << pos << " World\n";
```

As mentioned before, the variable defined in the if statement is also visible in the `else` block. So you can write:

```
if (const auto pos = myString.find("World"); pos != std::string::npos)
    std::cout << pos << " World\n";
else
    std::cout << pos << " not found!!\n";
```

Plus, you can use it with structured bindings (following Herb Sutter code[6]):

```
// better together: structured bindings + if initializer
if (auto [iter, succeeded] = mymap.insert(value); succeeded) {
    use(iter);  // ok
    // ...
} // iter and succeeded are destroyed here
```

In the above example, you can refer to `iter` and `succeeded` rather than `pair.first` and `pair.second` that is returned from `mymap.insert`.

As you can see, structured bindings and tuples allow you to create even more variables in the init section of the if-statement. But is the code easier to read that way?

For example:

```
string str = "Hi World";
if (auto [pos, size] = pair(str.find("Hi"), str.size()); pos != string::npos)
    std::cout << pos << " Hello, size is " << size;
```

We can argue that putting more code into the init section makes the code less readable, so pay attention to such cases.

Extra Info

The change was proposed in: P0305R1[7].

[6] https://herbsutter.com/2016/06/30/trip-report-summer-iso-c-standards-meeting-oulu/
[7] http://wg21.link/p0305r1

Inline Variables

With Non-Static Data Member Initialisation introduced in C++11, you can now declare and initialise member variables in one place:

```
class User {
    int _age {0};
    std::string _name {"unknown"};
};
```

However, with static variables (or `const static`) you need a declaration and then a definition in the implementation file.

C++11 with `constexpr` keyword allows you to declare and define static variables in one place, but it's limited to constant expressions only.

Previously, only methods/functions could be specified as `inline`, but now you can do the same with variables, inside a header file.

From the proposal P0386R2[8]**:**

> A variable declared inline has the same semantics as a function declared inline: it can be defined, identically, in multiple translation units, must be defined in every translation unit in which it is used, and the behaviour of the program is as if there was exactly one variable.

For example:

```
// inside a header file:
struct MyClass {
    static const int sValue;
};

// later in the same header file:
inline int const MyClass::sValue = 777;
```

Or even declaration and definition in one place:

[8] http://wg21.link/p0386r2

```
struct MyClass {
    inline static const int sValue = 777;
};
```

Also, note that `constexpr` variables are `inline` implicitly, so there's no need to use `constexpr inline myVar = 10;`.

An `inline` variable is also more flexible than a `constexpr` variable as it doesn't have to be initialised with a constant expression. For example, you can initialise an `inline` variable with `rand()`, but it's not possible to do the same with `constexpr` variable.

How Can it Simplify the Code?

A lot of header-only libraries can limit the number of hacks (like using inline functions or templates) and switch to using inline variables.

For example:

```
class MyClass {
    static inline int Seed(); // static method
};

inline int MyClass::Seed() {
    static const int seed = rand();
    return seed;
}
```

Can be changed into:

```
class MyClass {
    static inline int seed = rand();
};
```

C++17 guarantees that `MyClass::seed` will have the same value (generated at runtime) across all the compilation units!

Extra Info

The change was proposed in: P0386R2[9].

[9]http://wg21.link/p0386r2

constexpr Lambda Expressions

Lambda expressions were introduced in C++11, and since that moment they've become an essential part of modern C++. Another significant feature of C++11 is the `constexpr` specifier, which is used to express that a function or value can be computed at compile-time. In C++17, the two elements are allowed to exist together, so your lambda can be invoked in a constant expression context.

In C++11/14 the following code doesn't compile, but works with C++17:

Chapter General Language Features/lambda_square.cpp

```
int main () {
    constexpr auto SquareLambda = [] (int n) { return n*n; };
    static_assert(SquareLambda(3) == 9, "");
}
```

Since C++17 lambda expressions (their call operator `operator()`) that follow the rules of standard `constexpr` functions are implicitly declared as `constexpr`.

What are the limitations of `constexpr` functions? Here's a summary (from 10.1.5 The constexpr specifier [dcl.constexpr][10]):

- they cannot be virtual
- their return type shall be a literal type
- their parameter types shall be a literal type
- their function bodies cannot contain: `asm` definition, a `goto` statement, try-block, or a variable that is a non-literal type or static or thread storage duration

In practice, in C++17, if you want your function or lambda to be executed at compile-time, then the body of this function shouldn't invoke any code that is not `constexpr`. For example, you cannot allocate memory dynamically or throw exceptions.

`constexpr` lambda expressions are also covered in the Other Changes Chapter and in a free ebook: C++ Lambda Story[11].

Extra Info

The change was proposed in: P0170[12].

[10]https://timsong-cpp.github.io/cppwp/n4659/dcl.constexpr#3
[11]https://leanpub.com/cpplambda
[12]http://wg21.link/p0170

Capturing `[*this]` in Lambda Expressions

When you write a lambda inside a class method, you can reference a member variable by capturing `this`. For example:

Chapter General Language Features/capture_this.cpp

```
struct Test  {
    void foo() {
        std::cout << m_str  << '\n';
        auto addWordLambda = [this]() { m_str += "World"; };
        addWordLambda ();
        std::cout << m_str  << '\n';
    }

    std::string m_str {"Hello "};
};
```

In the line with `auto addWordLambda = [this]() {... }` we capture `this` pointer and later we can access `m_str`.

Please notice that we captured `this` by value... to a pointer. You have access to the member variable, not its copy. The same effect happens when you capture by `[=]` or `[&]`. That's why when you call `foo()` on some `Test` object then you'll see the following output:

```
Hello
Hello World
```

`foo()` prints `m_str` two times. The first time we see `"Hello"`, but the next time it's `"Hello World"` because the lambda `addWordLambda` changed it.

How about more complicated cases? Do you know what will happen with the following code?

Returning a Lambda From a Method

```cpp
#include <iostream>

struct Baz {
    auto foo() {
        return [=] { std::cout << s << '\n'; };
    }
    std::string s;
};

int main() {
   auto f1 = Baz{"ala"}.foo();
   f1();
}
```

The code declares a `Baz` object and then invokes `foo()`. Please note that `foo()` returns a lambda that captures a member of the class.

A capturing block like `[=]` suggests that we capture variables by value, but if you access members of a class in a lambda expression, then it does this implicitly via the `this` pointer. So we captured a copy of `this` pointer, which is a dangling pointer as soon as we exceed the lifetime of the `Baz` object.

In C++17 you can write: `[*this]` and that will capture **a copy** of the whole object.

```cpp
auto lam = [*this]() { std::cout << s; };
```

In C++14, the only way to make the code safer is init capture `*this`:

```cpp
auto lam = [self=*this] { std::cout << self.s; };
```

Capturing `this` might get tricky when a lambda can outlive the object itself. This might happen when you use async calls or multithreading.

In C++20 (see P0806[13]) you'll also see an extra warning if you capture `[=]` in a method. Such expression captures the `this` pointer, and it might not be exactly what you want.

Extra Info

The change was proposed in: P0018[14].

[13]https://wg21.link/P0806
[14]http://wg21.link/p0018

Nested Namespaces

Namespaces allow grouping types and functions into separate logical units.

For example, it's common to see that each type or function from a library XY will be stored in a namespace `xy`. Like in the below case, where there's `SuperCompressionLib` and it exposes functions called `Compress()` and `Decompress()`:

```
namespace SuperCompressionLib {
    bool Compress();
    bool Decompress();
}
```

Things get interesting if you have two or more nested namespaces.

```
namespace MySuperCompany {
    namespace SecretProject {
        namespace SafetySystem {
            class SuperArmor {
                // ...
            };
            class SuperShield {
                // ...
            };
        } // SafetySystem
    } // SecretProject
} // MySuperCompany
```

With C++17 nested namespaces can be written more compactly:

```
namespace MySuperCompany::SecretProject::SafetySystem {
    class SuperArmor {
    // ...
    };
    class SuperShield {
    // ...
    };
}
```

Such syntax is comfortable, and it will be easier to use for developers that have experience in languages like C# or Java.

In C++17 also the Standard Library was "compacted" in several places by using the new nested namespace feature:

For example, for `regex`.

In C++17 it's defined as:

```
namespace std::regex_constants {
    typedef T1 syntax_option_type;
    // ...
}
```

Before C++17 the same was declared as:

```
namespace std {
    namespace regex_constants {
        typedef T1 syntax_option_type;
        // ...
    }
}
```

The above nested declarations appear in the C++ Specification, but it might look different in an STL implementation.

Extra Info

The change was proposed in: N4230[15].

[15]http://wg21.link/N4230

`__has_include` Preprocessor Expression

If your code has to work under two different compilers, then you might experience two different sets of available features and platform-specific changes.

In C++17 you can use `__has_include` preprocessor constant expression to check if a given header exists:

```
#if __has_include(<header_name>)
#if __has_include("header_name")
```

`__has_include` was available in Clang as an extension for many years, but now it was added to the Standard. It's a part of "feature testing" helpers that allows you to check if a particular C++ feature or a header is available. If a compiler supports this macro, then it's accessible even without the C++17 flag, that's why you can check for a feature also if you work in C++11, or C++14 "mode".

As an example, we can test if a platform has `<charconv>` header that declares C++17's low-level conversion routines:

Chapter General Language Features/has_include.cpp

```
#if defined __has_include
#    if __has_include(<charconv>)
#        define has_charconv 1
#        include <charconv>
#    endif
#endif

std::optional<int> ConvertToInt(const std::string& str) {
    int value { };
    #ifdef has_charconv
        const auto last = str.data() + str.size();
        const auto res = std::from_chars(str.data(), last, value);
        if (res.ec == std::errc{} && res.ptr == last)
            return value;
    #else
        // alternative implementation...
    #endif

    return std::nullopt;
}
```

In the above code, we declare `has_charconv` based on the `__has_include` condition. If the header is not there, we need to provide an alternative implementation for `ConvertToInt`. You can check this code against GCC 7.1 and GCC 9.1 and see the effect as GCC 7.1 doesn't expose the `charconv` header.

Note: In the above code we cannot write:

```
#if defined __has_include && __has_include(<charconv>)
```

As in older compilers - that don't support `__has_include` we'd get a compile error. The compiler will complain that since `__has_include` is not defined and the whole expression is wrong.

Another important thing to remember is that sometimes a compiler might provide a header stub. For example, in C++14 mode the `<execution>` header might be present (it defines C++17 parallel algorithm execution modes), but the whole file will be empty (due to `ifdefs`). If you check for that file with `__has_include` and use C++14 mode, then you'll get a wrong result.

In C++20 we'll have standardised feature test macros that simplify checking for various C++ parts. For example, to test for `std::any` you can use `__cpp_lib_any`, for lambda support there's `__cpp_lambdas`. There's even a macro that checks for attribute support: `__has_cpp_attribute( attrib-name)`. GCC, Clang and Visual Studio exposes many of the macros already, even before C++20 is ready. Read more in Feature testing (C++20) - cppreference[16]

`__has_include` along with feature testing macros might greatly simplify multiplatform code that usually needs to check for available platform elements.

Extra Info

`__has_include` was proposed in: P0061[17].

[16]https://en.cppreference.com/w/cpp/feature_test
[17]http://wg21.link/p0061

Compiler support

Feature	GCC	Clang	MSVC
Structured Binding Declarations	7.0	4.0	VS 2017 15.3
Init-statement for if/switch	7.0	3.9	VS 2017 15.3
Inline variables	7.0	3.9	VS 2017 15.5
`constexpr` Lambda Expressions	7.0	5.0	VS 2017 15.3
Lambda Capture of `*this`	7.0	3.9	VS 2017 15.3
Nested namespaces	6.0	3.6	VS 2015
`has_include`	5	Yes	VS 2017 15.3

5. Templates

Do you work with templates and/or meta-programming?

If your answer is “YES,” then you might be pleased with the updates from C++17.

The new standard introduces many enhancements that make template programming much easier and more expressive.

In this chapter, you’ll learn:

- Template argument deduction for class templates
- `template<auto>`
- Fold expressions
- `if constexpr` - the compile-time if for C++!
- Plus some smaller, detailed improvements and fixes

Template Argument Deduction for Class Templates

C++17 filled a gap in the deduction rules for templates. Now, the template argument deduction can occur for class templates and not just for functions. That also means that a lot of your code that uses `make_Type` functions can now be removed.

For instance, to create an `std::pair` object, it was usually more comfortable to write:

```cpp
auto myPair = std::make_pair(42, "hello world"s);
```

Rather than:

```cpp
std::pair<int, std::string> myPair{42, "hello world"};
```

Because `std::make_pair()` is a template function, the compiler can perform the deduction of function template arguments and there's no need to write:

```cpp
auto myPair = std::make_pair<int, std::string>(42, "hello world");
```

Now, since C++17, the conformant compiler will nicely deduce the template parameter types for class templates too!

The feature is called *"Class Template Argument Deduction"* or *CTAD* in short.

In our example, you can now write:

```cpp
using namespace std::string_literals;
std::pair myPair{42, "hello world"s};     // deduced automatically!
```

CTAD also works with copy initialisation and when allocating memory through `new()`:

```cpp
auto otherPair = std::pair{42, "Hello"s}; // also deduced
auto ptr = new std::pair{42, "World"s};   // for new
```

CTAD can substantially reduce complex constructions like:

```
// lock guard:
std::shared_timed_mutex mut;
std::lock_guard<std::shared_timed_mutex> lck(mut);

// array:
std::array<int, 3> arr {1, 2, 3};
```

Can now become:

```
std::shared_timed_mutex mut;
std::lock_guard lck(mut);

std::array arr { 1, 2, 3 };
```

Note, that partial deduction cannot happen, you have to specify all the template parameters or none:

```
std::tuple t(1, 2, 3);              // OK: deduction
std::tuple<int,int,int> t(1, 2, 3); // OK: all arguments are provided
std::tuple<int> t(1, 2, 3);         // Error: partial deduction
```

With this feature, a lot of `make_Type` functions might not be needed - especially those that "emulate" template deduction for classes.

Still, there are factory functions that do additional work. For example, `std::make_shared` - it not only creates `shared_ptr`, but also makes sure the control block, and the pointed object are allocated in one memory region:

```
// control block and int might be in different places in memory
std::shared_ptr<int> p(new int{10});

// the control block and int are in the same contiguous memory section
auto p2 = std::make_shared<int>(10);
```

How does template argument deduction for classes work?

Let's enter the "Deduction Guides" area.

Deduction Guides

The compiler uses special rules called "*Deduction Guides*" to work out parameter types.

There are two types of guides: compiler-generated (implicitly generated) and user-defined.

To understand how the compiler uses the guides, let's look at a simplified deduction guide[1] for `std::array`:

```
template<typename T, typename... U>
array(T, U...) ->
    array<enable_if_t<(is_same_v<T, U> && ...), T>, 1 + sizeof...(U)>;
```

The syntax looks like a template function with a trailing return type. The compiler treats such "imaginary" function as a candidate for the parameters. If the pattern matches, then the found types are returned from the deduction.

In our case when you write:

```
std::array arr {1, 2, 3, 4};
```

Then, assuming `T` and `U...` arguments are of the same type, we can build up an array object of the type `std::array<int, 4>`.

In most cases, you can rely on the compiler to generate automatic deduction guides. They will be created for each constructor (also copy/move) of the primary class template. Please note that classes that are specialised or partially specialised won't work here.

As mentioned, you might also write custom deduction guides. A classic example is a deduction of `std::string` rather than `const char*`:

```
template<typename T> struct MyType {
  T str;
};

// custom deduction guide
MyType(const char *) -> MyType<std::string>;
MyType t{"Hello World"}; // deduces std::string
```

Without the custom deduction `T` would be deduced as `const char*`.

[1]simplified version of libstdc++ code https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/include/std/array

Another example of custom deduction guide comes from the `overload` pattern[2]:

```
template<class... Ts>
struct overload : Ts... { using Ts::operator()...; };

template<class... Ts>
overload(Ts...) -> overload<Ts...>; // deduction guide
```

The `overload` class inherits from other classes `Ts...` and then exposes their `operator()`. The custom deduction guide is used here to "transform" a list of callable types into the list of classes that we can derive from.

CTAD Limitations

In C++17 template argument deduction for classes has the following limitations:

- it doesn't work with template aggregate types
- the deduction doesn't include inheriting constructors
- it doesn't work with template aliases

Those limitations will be removed in C++20 through the already accepted proposal: P1021[3].

Extra Info

The CTAD feature was proposed in: P0091R3[4] and P0433 - Deduction Guides in the Standard Library[5].

Please note that while a compiler might declare full support for Template Argument Deduction for Class Templates, its corresponding STL implementation might still lack of custom deduction guides for some STL types. See the Compiler Support section at the end of the chapter.

[2]Read more about this pattern in the chapter about `std::variant`, section related to `std::visit()`
[3]https://wg21.link/P1021
[4]http://wg21.link/p0091r3
[5]https://wg21.link/p0433

Fold Expressions

C++11 introduced variadic templates, which is a powerful feature, especially if you want to work with a variable number of input template parameters to a function. For example, previously (pre C++11) you had to write several different versions of a template function (one for one parameter, another for two parameters, another for three params...).

Still, variadic templates required some additional code when you wanted to implement "recursive" functions like `sum`, `all`. You had to specify rules for the recursion.

For example:

```
auto SumCpp11() {
    return 0;
}

template<typename T1, typename... T>
auto SumCpp11(T1 s, T... ts) {
    return s + SumCpp11(ts...);
}
```

And with C++17 we can write much simpler code:

```
template<typename ...Args> auto sum(Args ...args) {
    return (args + ... + 0);
}

// or even:
template<typename ...Args> auto sum2(Args ...args) {
    return (args + ...);
}
```

The following variations of fold expressions[6] with binary operators (`op`) exist:

Expression	Name	Expansion
`(... op e)`	unary left fold	`((e1 op e2) op ...) op eN`
`(init op ... op e)`	binary left fold	`(((init op e1) op e2) op ...) op eN`
`(e op ...)`	unary right fold	`e1 op (... op (eN-1 op eN))`
`(e op ... op init)`	binary right fold	`e1 op (... op (eN-1 op (eN op init)))`

[6]https://en.cppreference.com/w/cpp/language/fold

`op` is any of the following 32 binary operators: `+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && || , .* ->*`. In a binary fold, both ops must be the same.

For example, when you write:

```
template<typename ...Args> auto sum2(Args ...args) {
    return (args + ...); // unary right fold over '+'
}

auto value = sum2(1, 2, 3, 4);
```

The template function is expanded into:

```
auto value = 1 + (2 + (3 + 4));
```

Also by default we get the following values for empty parameter packs:

Operator	default value
`&&`	`true`
`||`	`false`
`,`	`void()`
any other	ill-formed code

That's why you cannot call `sum2()` without any parameters, as the unary fold over operator `+` doesn't have any default value for the empty parameter list.

More Examples

Here's a quite nice implementation of a `printf` using folds P0036R0[7]:

[7]http://wg21.link/p0036r0

```
template<typename ...Args>
void FoldPrint(Args&&... args) {
    (std::cout << ... << std::forward<Args>(args)) << '\n';
}

FoldPrint("hello", 10, 20, 30);
```

However, the above `FoldPrint` will print arguments one by one, without any separator. For the above function call, you'll see `"hello102030"` on the output.

If you want separators and more formatting options, you have to alter the printing technique and use fold over comma:

```
template<typename ...Args>
void FoldSeparateLine(Args&&... args) {
    auto separateLine = [](const auto& v) {
        std::cout << v << '\n';
    };
    (... , separateLine (std::forward<Args>(args))); // over comma operator
}
```

The technique with fold over the comma operator is handy. Another example of it might be a special version of `push_back`:

```
template<typename T, typename... Args>
void push_back_vec(std::vector<T>& v, Args&&... args) {
    (v.push_back(std::forward<Args>(args)), ...);
}

std::vector<float> vf;
push_back_vec(vf, 10.5f, 0.7f, 1.1f, 0.89f);
```

In general, fold expression allows you to write cleaner, shorter and probably more comfortable to read the code.

Extra Info

The change was proposed in: N4295[8] and P0036R0[9].

[8]https://wg21.link/n4295
[9]https://wg21.link/p0036r0

`if constexpr`

This is a big one!

The compile-time if for C++!

The feature allows you to discard branches of an if statement at compile-time based on a constant expression condition.

```
if constexpr (cond)
     statement1; // Discarded if cond is false
else
     statement2; // Discarded if cond is true
```

For example:

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t;
    else
        return t;
}
```

`if constexpr` has the potential to simplify a lot of template code - especially when tag dispatching, SFINAE or preprocessor techniques are used.

Why Compile Time If?

At first, you may ask, why do we need `if constexpr` and those complex templated expressions… wouldn't a regular `if` work?

Here's a code example:

```
template <typename Concrete, typename... Ts>
std::unique_ptr<Concrete> constructArgs(Ts&&... params) {
    if (std::is_constructible_v<Concrete, Ts...>) // regular `if`
        return std::make_unique<Concrete>(std::forward<Ts>(params)...);
    else
        return nullptr;
}
```

The above routine is an "updated" version of `std::make_unique`: it returns `std::unique_-ptr` when the parameters allow it to construct the wrapped objects, or it returns `nullptr`.

Below there's simple code that tests `constructArgs`:

```
class Test {
public:
    Test(int, int) { }
};

int main() {
    auto p = constructArgs<Test>(10, 10, 10); // 3 args!
}
```

The code tries to build `Test` out of three parameters, but please notice that `Test` has only constructor that takes two `int` arguments.

When compiling you might get a similar compiler error:

```
In instantiation of 'typename std::_MakeUniq<_Tp>::__single_object
std::make_unique(_Args&& ...) [with _Tp = Test; _Args = {int, int, int};
typename std::_MakeUniq<_Tp>::__single_object
 = std::unique_ptr<Test, std::default_delete<Test> >]':

main.cpp:8:40: required from 'std::unique_ptr<_Tp>
    constructArgs(Ts&& ...) [with Concrete = Test; Ts = {int, int, int}]'
```

Let's try to understand this error message. After the template deduction the compiler compiles the following code:

```
if (std::is_constructible_v<Concrete, int, int, int>)
   return std::make_unique<Concrete>(10, 10, 10);
else
   return nullptr;
```

During the runtime, the `if` branch won't be executed - as `is_constructible_v` returns `false`, yet the code in the branch must compile.

That's why we need `if constexpr`, to "discard" code and compile only the matching statement.

To fix the code you have to add `constexpr`:

```
template <typename Concrete, typename... Ts>
std::unique_ptr<Concrete> constructArgs(Ts&&... params) {
  if constexpr (std::is_constructible_v<Concrete, Ts...>) // fixed!
      return std::make_unique<Concrete>(std::forward<Ts>(params)...);
   else
       return nullptr;
}
```

Now, the compiler evaluates the `if constexpr` condition at compile-time and for the expression `auto p = constructArgs<Test>(10, 10, 10);` the whole `if` branch will be "removed" from the second step of the compilation process.

To be precise, the code in the discarded branch is not entirely removed from the compilation phase. Only expressions that are dependent on the template parameter used in the condition are not instantiated. The syntax must always be valid.

For example:

```
template <typename T>
void Calculate(T t) {
    if constexpr (std::is_integral_v<T>) {
        // ...
        static_assert(sizeof(int) == 100);
    }
    else {
        execute(t);
        strange syntax
    }
}
```

In the above artificial code, if the type `T` is `int`, then the `else` branch is discarded, which means `execute(t)` won't be instantiated. But the line `strange syntax` will still be compiled (as it's not dependent on `T`) and that's why you'll get a compile error about that.

Furthermore, another compilation error will come from `static_assert`, the expression is also not dependent on `T`, and that's why it will always be evaluated.

Template Code Simplification

Before C++17 if you had several versions of an algorithm - depending on the type requirements - you could use SFINAE or tag dispatching to generate a dedicated overload resolution set.

For example:

Chapter Templates/sfinae_example.cpp

```cpp
template <typename T>
std::enable_if_t<std::is_integral_v<T>, T> simpleTypeInfo(T t) {
    std::cout << "foo<integral T> " << t << '\n';
    return t;
}

template <typename T>
std::enable_if_t<!std::is_integral_v<T>, T> simpleTypeInfo(T t) {
    std::cout << "not integral \n";
    return t;
}
```

In the above example, we have two function implementations, but only one of them will end up in the overload resolution set. If `std::is_integral_v` is true for the `T` type, then the top function is taken, and the second one rejected due to SFINAE.

The same thing can happen when using tag dispatching:

Chapter Templates/tag_dispatching_example.cpp

```
template <typename T>
T simpleTypeInfoTagImpl(T t, std::true_type) {
    std::cout << "foo<integral T> " << t << '\n';
    return t;
}

template <typename T>
T simpleTypeInfoTagImpl(T t, std::false_type) {
    std::cout << "not integral \n";
    return t;
}

template <typename T>
T simpleTypeInfoTag(T t) {
    return simpleTypeInfoTagImpl(t, std::is_integral<T>{});
}
```

Now, instead of SFINAE, we generate a unique type tag for the condition: `true_type` or `false_type`. Depending on the result, only one implementation is selected.

We can now simplify this pattern with `if constexpr`:

```
template <typename T>
T simpleTypeInfo(T t) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "foo<integral T> " << t << '\n';
    }
    else {
        std::cout << "not integral \n";
    }
    return t;
}
```

Writing template code becomes more “natural” and doesn’t require that many “tricks”.

Examples

Let’s see a couple of examples:

Line Printer

You might have already seen the below example in the Jump Start section at the beginning of this Part of the book. Let's dive into the details and see how the code works.

```
template<typename T> void linePrinter(const T& x) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "num: " << x << '\n';
    }
    else if constexpr (std::is_floating_point_v<T>) {
        const auto frac = x - static_cast<long>(x);
        std::cout << "flt: " << x << ", frac " << frac << '\n';
    }
    else if constexpr(std::is_pointer_v<T>) {
        std::cout << "ptr, ";
        linePrinter(*x);
    }
    else {
        std::cout << x << '\n';
    }
}
```

`linePrinter` uses `if constexpr` to check the input type. Based on that, we can output additional messages. An interesting thing happens with the pointer type - when a pointer is detected the code dereferences it and then calls `linePrinter` recursively.

Declaring Custom `get<N>` Functions

The structured binding expression works for simple structures that have all public members, like

```
struct S {
    int n;
    std::string s;
    float d;
};

S s;
auto [a, b, c] = s;
```

However, if you have a custom type (with private members), then it's also possible to override `get<N>` functions so that structured binding can work. Here's some code to demonstrate this idea:

```
class MyClass {
public:
    int GetA() const { return a; }
    float GetB() const { return b; }

private:
    int a;
    float b;
};

template <std::size_t I> auto get(MyClass& c) {
    if constexpr (I == 0)      return c.GetA();
    else if constexpr (I == 1) return c.GetB();
}

// specialisations to support tuple-like interface
namespace std {
    template <> struct tuple_size<MyClass> : integral_constant<size_t, 2> { };

    template <> struct tuple_element<0,MyClass> { using type = int; };
    template <> struct tuple_element<1,MyClass> { using type = float; };
}
```

In the above code you have the advantage of having everything in one function. It's also possible to do it as template specialisations:

```
template <> auto& get<0>(MyClass &c) { return c.GetA(); }
template <> auto& get<1>(MyClass &c) { return c.GetB(); }
```

For more examples you can read the chapter about Replacing `std::enable_if` with `if constexpr` and also the chapter Structured Bindings - the section about custom `get<N>` specialisations.

You can also see the following article: Simplify code with if constexpr in C++17[10]

Extra Info

The change was proposed in: P0292R2[11].

[10]https://www.bfilipek.com/2018/03/ifconstexpr.html
[11]https://wg21.link/p0292r2

Declaring Non-Type Template Parameters With `auto`

This is another part of the strategy to use `auto` everywhere. With C++11 and C++14, you can use it to deduce variables or even return types automatically, plus there are also generic lambdas. Now you can also use it for deducing non-type template parameters.

For example:

```
template <auto value> void f() { }
f<10>();        // deduces int
```

This is useful, as you don't have to have a separate parameter for the type of non-type parameter. Like in C++11/14:

```
template <typename Type, Type value> constexpr Type TConstant = value;
constexpr auto const MySuperConst = TConstant<int, 100>;
```

With C++17 it's a bit simpler:

```
template <auto value> constexpr auto TConstant = value;
constexpr auto const MySuperConst = TConstant<100>;
```

There's no need to write `Type` explicitly.

As one of the advanced uses a lot of papers, and articles point to an example of heterogeneous compile time list:

```
template <auto ... vs> struct HeterogenousValueList {};
using MyList = HeterogenousValueList<'a', 100, 'b'>;
```

Before C++17 it was not possible to declare such list directly, some wrapper class would have had to be provided first.

Extra Info

The change was proposed in: P0127R2[12]. In P0127R1[13], you can find some more examples and reasoning.

[12] https://wg21.link/p0127r2
[13] https://wg21.link/p0127r1

Other Changes

In C++17 there are also other language and library features related to templates that are worth mentioning:

Allow `typename` in a Template Template Parameters.

Allows you to use `typename` instead of `class` when declaring a template template parameter. Normal type parameters can use them interchangeably, but template template parameters were restricted to `class`.

More information in N4051[14].

Allow Constant Evaluation for all Non-Type Template Arguments

Remove syntactic restrictions for pointers, references, and pointers to members that appear as non-type template parameters.

More information in N4268[15].

Variable Templates for Traits

All the type traits that yields `::value` got accompanying `_v` variable templates. For example:

`std::is_integral<T>::value` has `std::is_integral_v<T>`

`std::is_class<T>::value` has `std::is_class_v<T>`

This improvement already follows the `_t` suffix additions in C++14 (template aliases) to type traits that returns `::type`. Such change can considerably shorten template code.

More information in P0006R0[16].

[14]https://wg21.link/n4051
[15]https://wg21.link/n4268
[16]https://wg21.link/p0006r0

Pack Expansions in Using Declarations

The feature is an enhancement for variadic templates and parameter packs.

The compiler will now support the `using` keyword in pack expansions:

```
template<class... Ts> struct overloaded : Ts... {
    using Ts::operator()...;
};
```

The `overloaded` class exposes all overloads for `operator()` from the base classes. Before C++17, you would have to use recursion for parameter packs to achieve the same result. The `overloaded` pattern is a very useful enhancement for `std::visit`, read more in the "Overload" section in the Variant chapter.

More information in P0195[17].

Logical Operation Metafunctions

C++17 adds handy template metafunctions:

- `template<class... B> struct conjunction;` - logical `AND`
- `template<class... B> struct disjunction;` - logical `OR`
- `template<class B> struct negation;` - logical negation

Here's an example, based on the code from the proposal:

```
template<typename... Ts>
std::enable_if_t<std::conjunction_v<std::is_same<int, Ts>...> >
PrintIntegers(Ts ... args) {
    (std::cout << ... << args) << '\n';
}
```

The above function `PrintIntegers` works with a variable number of arguments, but they all have to be of type `int`.

The helper metafunctions can increase the readability of advanced template code. They are available in `<type_traits>` header.

More information in P0013[18].

[17]https://wg21.link/P0195
[18]https://wg21.link/P0013

`std::void_t` Transformation Trait

A surprisingly simple[19] metafunction that maps a list of types into `void`:

```
template< class... >
using void_t = void;
```

`void_t` is very handy to SFINAE ill-formed types. For example it might be used to detect a function overload:

```
void Compute(int &) { } // example function

template <typename T, typename = void>
struct is_compute_available : std::false_type {};

template <typename T>
struct is_compute_available<T,
           std::void_t<decltype(Compute(std::declval<T>())) >>
               : std::true_type {};

static_assert(is_compute_available<int&>::value);
static_assert(!is_compute_available<double&>::value);
```

`is_compute_available` checks if a `Compute()` overload is available for the given template parameter. If the expression `decltype(Compute(std::declval<T>()))` is valid, then the compiler will select the template specialisation. Otherwise, it's SFINEed, and the primary template is chosen.

More information in N3911[20].

[19]Compilers that don't implement a fix for CWG 1558 (for C++14) might need a more complicated version of it.
[20]https://wg21.link/n3911

Compiler Support

Feature	GCC	Clang	MSVC
Template argument deduction for class templates	7.0/8.0[21]	5.0	VS 2017 15.7
Deduction Guides in the Standard Library	8.0[22]	7.0/in progress[23]	VS 2017 15.7
Declaring non-type template parameters with auto	7.0	4.0	VS 2017 15.7
Fold expressions	6.0	3.9	VS 2017 15.5
`if constexpr`	7.0	3.9	VS 2017

[21]Additional improvements for Template Argument Deduction for Class Templates happened in GCC 8.0, P0512R0.

[22]Deduction Guides are not listed in the status pages of LibSTDC++, so we can assume they were implemented as part of Template argument deduction for class templates.

[23]The status page for LibC++ mentions that `<string>`, sequence containers, container adaptors and `<regex>` portions have been implemented so far.

6. Standard Attributes

Code annotations - attributes - are probably not the best-known feature of C++. However, they might be handy for expressing additional information for the compiler and also for other programmers. Since C++11, there has been a standard way of specifying attributes. And in C++17 we got even more useful additions.

In this chapter, you'll learn:

- What are the attributes in C++
- Vendor-specific code annotations vs the Standard form
- In what cases attributes are handy
- C++11 and C++14 attributes
- New additions in C++17

Why Do We Need Attributes?

Have you ever used `__declspec`, `__attribute__` or `#pragma` directives in your code?

For example:

```
// set an alignment
struct S { short f[3]; } __attribute__ ((aligned (8)));

// this function won't return
void fatal () __attribute__ ((noreturn));
```

Or for DLL import/export in MSVC:

```
#if COMPILING_DLL
    #define DLLEXPORT __declspec(dllexport)
#else
    #define DLLEXPORT __declspec(dllimport)
#endif
```

Those are existing forms of compiler-specific attributes/annotations.

So what is an attribute?

An attribute is additional information that can be used by the compiler to produce code. It might be utilised for optimisation or some specific code generation (like DLL stuff, OpenMP, etc.). Also, annotations allow you to write more expressive syntax and help other developers to reason about code.

Contrary to other languages such as C#, in C++, the compiler has fixed the meta-information system. You cannot add user-defined attributes. In C# you can derive from `System.Attribute`.

What's best about Modern C++ attributes?

Since C++11, we get more and more standardised attributes that will work with other compilers. We're moving away from compiler-specific annotation to standard forms. Rather than learning various annotation syntaxes you'll be able to write code that is common and has the same behaviour.

In the next section, you'll see how attributes used to work before C++11.

Before C++11

In the era of C++98/03, each compiler introduced its own set of annotations, usually with a different keyword.

Often, you could see code with `#pragma`, `__declspec`, `__attribute` spread throughout the code.

Here's the list of the common syntax from GCC/Clang and MSVC:

GCC Specific Attributes

GCC uses annotation in the form of `__attribute__((attr_name))`. For example:

```
int square (int) __attribute__ ((pure)); // pure function
```

Documentation:

- Attribute Syntax - Using the GNU Compiler Collection (GCC)[1]
- Using the GNU Compiler Collection (GCC): Common Function Attributes[2]

MSVC Specific Attributes

Microsoft mostly used `__declspec` keyword, as their syntax for various compiler extensions. See the documentation here: `__declspec` Microsoft Docs[3].

```
__declspec(deprecated) void LegacyCode() { }
```

Clang Specific Attributes

Clang, as it's straightforward to customise, can support different types of annotations, so look at the documentation to find more. Most of GCC attributes work with Clang.

See the documentation here: Attributes in Clang — Clang documentation[4].

[1]https://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/Attribute-Syntax.html
[2]https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes
[3]https://docs.microsoft.com/en-Us/cpp/cpp/declspec
[4]https://clang.llvm.org/docs/AttributeReference.html

Attributes in C++11 and C++14

C++11 took one big step to minimise the need to use vendor-specific syntax. By introducing the standard format, we can move a lot of compiler-specific attributes into the universal set.

C++11 provides a cleaner format of specifying annotations over our code.

The basic syntax is just `[[attr]]` or `[[namespace::attr]]`.

You can use `[[attr]]` over almost anything: types, functions, enums, etc., etc.

For example:

```
[[attrib_name]] void foo() { }      // on a function
struct [[deprecated]] OldStruct { } // on a struct
```

In C++11 we have the following attributes:

`[[noreturn]]`:

It tells the compiler that control flow will not return to the caller. Examples:

- `[[noreturn]] void terminate() noexcept;`
- functions like `std::abort` or `std::exit` are also marked with this attribute.

`[[carries_dependency]]`:

Indicates that the dependency chain in release-consume `std::memory_order` propagates in and out of the function, which allows the compiler to skip unnecessary memory fence instructions. Mostly to help to optimise multi-threaded code and when using different memory models.

C++14 added:

`[[deprecated]]` and `[[deprecated("reason")]]`:

Code marked with this attribute will be reported by the compiler. You can set its reason.

Example of `[[deprecated]]`:

```
[[deprecated("use AwesomeFunc instead")]] void GoodFunc() { }

// call somewhere:
GoodFunc();
```

GCC reports the following warning:

```
warning: 'void GoodFunc()' is deprecated: use AwesomeFunc instead
[-Wdeprecated-declarations]
```

You know a bit about the old approach, new way in C++11/14... so what's the deal with C++17?

C++17 Additions

With C++17 we get three more standard attributes:

- `[[fallthrough]]`
- `[[nodiscard]]`
- `[[maybe_unused]]`

Extra Info

The new attributes were specified in P0188[5] and P0068[6](reasoning).

Plus three supporting features:

- Attributes for Namespaces and Enumerators
- Ignore Unknown Attributes
- Using Attribute Namespaces Without Repetition

Let's go through the new attributes first.

[5]https://wg21.link/p0188
[6]https://wg21.link/p0068

`[[fallthrough]]` Attribute

Indicates that a fall-through in a switch statement is intentional and a warning should not be issued for it.

```
switch (c) {
case 'a':
    f(); // Warning! fallthrough is perhaps a programmer error
case 'b':
    g();
[[fallthrough]]; // Warning suppressed, fallthrough is ok
case 'c':
    h();
}
```

With this attribute, the compiler can understand the intentions of a programmer. It's also much more readable than using a comment.

`[[maybe_unused]]` Attribute

Suppresses compiler warnings about unused entities:

```
static void impl1() { ... } // Compilers may warn when function not called
[[maybe_unused]] static void impl2() { ... } // Warning suppressed

void foo() {
   int x = 42; // Compilers may warn when x is not used later
   [[maybe_unused]] int y = 42; // Warning suppressed for y
}
```

Such behaviour is helpful when some of the variables and functions are used in debug only path. For example in `assert()` macros;

```
void doSomething(std::string_view a, std::string_view b) {
    assert(a.size() < b.size());
}
```

If later `a` or `b` is no used in this function, then the compiler will generate a warning in release only builds. Marking the given argument with `[[maybe_unused]]` will solve this warning.

[[nodiscard]] Attribute

[[nodiscard]] can be applied to a function or a type declaration to mark the importance of the returned value:

```
[[nodiscard]] int Compute();
void Test() {
    Compute(); // Warning! return value of a
               // nodiscard function is discarded
}
```

If you forget to assign the result to a variable, then the compiler should emit a warning.

What it means is that you can force users to handle errors. For example, what happens if you forget about using the return value from new or std::async()?

Additionally, the attribute can be applied to types. One use case for it might be error codes:

```
enum class [[nodiscard]] ErrorCode {
    OK,
    Fatal,
    System,
    FileIssue
};

ErrorCode OpenFile(std::string_view fileName);
ErrorCode SendEmail(std::string_view sendto,
                    std::string_view text);
ErrorCode SystemCall(std::string_view text);
```

Now, every time you'd like to call such functions, you're "forced" to check the return value. For important functions checking return codes might be crucial and using [[nodiscard]] might save you from a few bugs.

You might also ask what it means "not to use" a return value?

In the Standard, it's defined as "Discarded-value expressions"[7]. It means that you call a function only for its side effects. In other words, there's no if statement around or an assignment expression. In that case, when a type is marked as [[nodiscard]] the compiler is encouraged to report a warning.

However, to suppress the warning you can explicitly cast the return value to void or use [[maybe_unused]]:

[7] http://en.cppreference.com/w/cpp/language/expressions#Discarded-value_expressions

```
[[nodiscard]] int Compute();
void Test() {
    static_cast<void>(Compute()); // fine...

    [[maybe_unused]] auto ret = Compute();
}
```

In addition, in C++20 the Standard Library will use `[[nodiscard]]` in a few places like: `operator new`, `std::async()`, `std::allocate()`, `std::launder()`, and `std::empty()`.
This feature was already merged into C++20 with P0600[8].

The second addition to C++20 is `[[nodiscard("reason")]]`, see in P1301[9]. This lets you specify why not using a returned value might generate issues — for example, some resource leak.

Attributes for Namespaces and Enumerators

The idea for attributes in C++11 was to be able to apply them to all sensible places like classes, functions, variables, typedefs, templates, enumerations… But there was an issue in the specification that blocked attributes when they were applied on namespaces or enumerators.

This is now fixed in C++17. We can now write:

```
namespace [[deprecated("use BetterUtils")]] GoodUtils {
    void DoStuff() { }
}

namespace BetterUtils {
    void DoStuff() { }
}

// use:
GoodUtils::DoStuff();
```

[8]https://wg21.link/p0600
[9]https://wg21.link/P1301

Clang reports:

```
warning: 'GoodUtils' is deprecated: use BetterUtils
[-Wdeprecated-declarations]
```

Another example is the use of deprecated attribute on enumerators:

```
enum class ColorModes {
    RGB [[deprecated("use RGB8")]],
    RGBA [[deprecated("use RGBA8")]],
    RGB8,
    RGBA8
};

// use:
auto colMode = ColorModes::RGBA;
```

Under GCC we'll get:

```
warning: 'RGBA' is deprecated: use RGBA8
[-Wdeprecated-declarations]
```

Extra Info

The change was described in N4266[10](wording) and N4196[11](reasoning).

Ignore Unknown Attributes

The feature is mostly for clarification.

Before C++17, if you tried to use some compiler-specific attribute, you might even get an error when compiling in another compiler that doesn't support it. Now, the compiler omits the attribute specification and won't report anything (or just a warning). This wasn't mentioned in the Standard, and it needed clarification.

[10]https://wg21.link/n4266
[11]https://wg21.link/n4196

```
// compilers which don't
// support MyCompilerSpecificNamespace will ignore this attribute
[[MyCompilerSpecificNamespace::do_special_thing]]
void foo();
```

For example in GCC 7.1 there's a warnings:

```
warning: 'MyCompilerSpecificNamespace::do_special_thing'
scoped attribute directive ignored [-Wattributes]
void foo();
```

Extra Info

The change was described in P0283R2[12](wording) and P0283R1[13](reasoning).

Using Attribute Namespaces Without Repetition

The feature simplifies the case where you want to use multiple attributes, like:

```
void f() {
    [[rpr::kernel, rpr::target(cpu,gpu)]] // repetition
    doTask();
}
```

Proposed change:

```
void f() {
    [[using rpr: kernel, target(cpu,gpu)]]
    doTask();
}
```

That simplification might help when building tools that automatically translate annotated code of that type into different programming models.

Extra Info

The change was described in: P0028R4[14].

[12]https://wg21.link/p0283r2
[13]https://wg21.link/p0283r1
[14]http://wg21.link/p0028r4

Section Summary

Attributes available in C++17:

Attribute	Description
`[[noreturn]]`	a function does not return to the caller
`[[carries_dependency]]`	extra information about dependency chains
`[[deprecated]]`	an entity is deprecated
`[[deprecated("reason")]]`	provides additional message about the deprecation
`[[fallthrough]]`	indicates a intentional fall-through in a switch statement
`[[nodiscard]]`	a warning is generated if the return value is discarded
`[[maybe_unused]]`	an entity might not be used in the code

Each compiler vendor can specify their syntax for attributes and annotations. In Modern C++, the ISO Committee tries to extract common parts and standardise it as `[[attributes]]`.

There's also a relevant quote from Bjarne Stroustrup's C++11 FAQ[15] about suggested use:

> There is a reasonable fear that attributes will be used to create language dialects. The recommendation is to use attributes to only control things that do not affect the meaning of a program but might help detect errors (e.g. `[[noreturn]]`) or help optimisers (e.g. `[[carries_dependency]]`).

Compiler support

Feature	GCC	Clang	MSVC
`[[fallthrough]]`	7.0	3.9	15.0
`[[nodiscard]]`	7.0	3.9	15.3
`[[maybe_unused]]`	7.0	3.9	15.3
Attributes for namespaces and enumerators	4.9/6[16]	3.4	14.0
Ignore unknown attributes	All versions	3.9	14.0
Using attribute namespaces without repetition	7.0	3.9	15.3

All of the above compilers also support C++11/14 attributes.

[15]http://stroustrup.com/C++11FAQ.html#attributes
[16]GCC 4.9 (namespaces) / GCC 6 (enumerations)

Part 2 - The Standard Library Changes

While new language features allow you to write more compact code, you also need the tools - in the form of the Standard Library types. The classes, algorithms, containers and components from the Library can significantly enhance your productivity. C++17 offers even more handy instruments: for example the filesystem, new vocabulary types and even parallel algorithms!

In this part you'll learn:

- How to represent nullable types with `std::optional`
- What's a tagged union? And why we need a type-safe union in the form of `std::variant`
- How to represent any type with `std::any`
- How to use `string_view` to gain performance and not break your application
- What are the new string operations available in the Standard Library
- How to work with the filesystem using the Standard Library
- What are the parallel algorithms
- What are other smaller changes to the Standard Library, like `std::byte`, polymorphic memory resources, new math functions or improvements to ordered containers
- What was deprecated or removed in C++17 in terms of the Standard Library

7. `std::optional`

C++17 adds a few wrapper types that make it possible to write more expressive code. In this chapter, you'll see `std::optional`, which models a `nullable` type. With this utility, your objects can easily express that they don't have any value. Such behaviour is more straightforward to achieve than using some unique values (like `-1`, `null`).

In this chapter, you'll learn:

- Why we need nullable types
- How does `std::optional` work and what does it do
- Operations on `std::optional`
- The performance cost of using the type
- Example use cases

Introduction

How can you mark that a type doesn't contain any value?

One approach is to achieve "null-ability" by using unique values (`-1`, infinity, `nullptr`). Before use, you need to compare the object against the predefined value to see if it's not empty. Such a pattern is widespread in programming. For instance `string::find` returns a value that represents the position or `npos` when it's "null" or the pattern is not found.

Alternatively, you could try with `std::unique_ptr<Type>` and treat the empty pointer as not initialised. That works but comes with the cost of allocating memory for the object and is not a recommended technique.

Another technique is to build a wrapper that adds a boolean flag to other types. Such wrapper can quickly determine the state of the object. And this is how in a nutshell works `std::optional`.

Optional types that come from the functional programming world bring type safety and expressiveness. Most other languages have something similar: for example `std::option` in Rust, `Optional<T>` in Java, `Data.Maybe` in Haskell.

`std::optional` was added in C++17 and brought a lot of experience from `boost::optional` that has been available for many years. With C++17 you can just `#include <optional>` and use the type.

What's more `std::optional` was available also in Library Fundamentals TS, so there's a chance that your C++14 compiler could also support it in the `<experimental/optional>` header.

`std::optional` is still a value type (so it can be copied, via deep copy). Additionally, `std::optional` doesn't need to allocate any memory on the free store.

`std::optional` is a part of C++ **vocabulary types** along with `std::any`, `std::variant` and `std::string_view`.

When to Use

You can usually use an optional wrapper in the following scenarios:

1) If you want to represent a nullable type.

- Rather than using unique values (like `-1`, `nullptr`, `NO_VALUE` or something)
- For example, a user's middle name is optional. You could assume that an empty string would work here, but knowing if a user entered something or not might be important. `std::optional<std::string>` gives you more information.

2) Return a result of some computation (processing) that fails to produce a value and is not an error.

For example, finding an element in a dictionary: if there's no element under a key, it's not an error, but we need to handle the situation.

3) To perform lazy-loading of resources.

For example, if the construction of a resource type is substantial, or if there's no default constructor, you can define it as `std::optional<Resource>`. In that form, you can pass it around the system, and then initialise it (load a resource), when the application wants to access it for the first time.

4) To pass optional parameters into functions.

The documentation for `boost.optional` has a useful summary on when we should use the type, see in When to use Optional[1]:

> It is recommended to use `optional<T>` in situations where there is exactly one, clear (to all parties) reason for having no value of type `T`, and where the lack of value is as natural as having any regular value of `T`.

While sometimes the decision to use optional might be blurry, it best suits the cases when the value is empty, and it's a normal state of the program.

[1]https://www.boost.org/doc/libs/1_67_0/libs/optional/doc/html/boost_optional/tutorial/when_to_use_optional.html

A Basic Example

Here's a simple example of what you can do with optional:

```
// UI class...
std::optional<std::string> UI::FindUserNick() {
    if (IsNickAvailable())
        return mStrNickName; // return a string

    return std::nullopt; // same as return { };
}

// use:
std::optional<std::string> UserNick = UI->FindUserNick();
if (UserNick)
    Show(*UserNick);
```

In the above code, we define a function that returns an `optional` containing a `string`. If the user's nickname is available, then it will return a string. If not, then it returns `nullopt`. Later we can assign it to an optional and check (it converts to `bool`) if it contains any value or not. Optional defines `operator*` so we can easily access the stored value.

In the following sections, you'll see how to create `std::optional`, operate on it, pass it around, and even what is the performance cost you might want to consider.

`std::optional` Creation

There are several ways to create `std::optional`:

- Initialise as empty
- Directly with a value
- With a value using deduction guides
- By using `make_optional`
- With `std::in_place`
- From other `optional`

See code below:

```
// empty:
std::optional<int> oEmpty;
std::optional<float> oFloat = std::nullopt;

// direct:
std::optional<int> oInt(10);
std::optional oIntDeduced(10); // deduction guides

// make_optional
auto oDouble = std::make_optional(3.0);
auto oComplex = std::make_optional<std::complex<double>>(3.0, 4.0);

// in_place
std::optional<std::complex<double>> o7{std::in_place, 3.0, 4.0};

// will call vector with direct init of {1, 2, 3}
std::optional<std::vector<int>> oVec(std::in_place, {1, 2, 3});

// copy from other optional:
auto oIntCopy = oInt;
```

As you can see in the above code sample, you have a lot of flexibility with the creation of optional. It's straightforward for primitive types, and this simplicity is extended even to complex types.

If you want the full control over the creation and efficiency, it's also good to know `in_place` helper types.

`in_place` Construction

`std::optional` is a wrapper type, so you should be able to create optional objects almost in the same way as the wrapped object. And in most cases you can:

```
std::optional<std::string> ostr{"Hello World"};
std::optional<int> oi{10};
```

You can write the above code without stating the constructor such as:

```
std::optional<std::string> ostr{std::string{"Hello World"}};
std::optional<int> oi{int{10}};
```

Because `std::optional` has a constructor that takes `U&&` (a universal reference to a type that converts to the type stored in the optional). In our case it's recognised as `const char*` and strings can be initialised from it.

So what's the advantage of using `std::in_place_t` in `std::optional`?

There are at least two crucial reasons:

- Default constructor
- Efficient construction for constructors with many arguments

Default Construction

If you have a class with a default constructor, like:

```
class UserName {
public:
    UserName() : mName("Default") {

    }
    // ...
};
```

How would you create an `optional` that contains `UserName{}`?

You can write:

```
std::optional<UserName> u0; // empty optional
std::optional<UserName> u1{}; // also empty

// optional with default constructed object:
std::optional<UserName> u2{UserName()};
```

That works but it creates an additional temporary object. If we traced each different constructor and destructor calls, we would get the following output:

```
UserName::UserName('Default')
UserName::UserName(move 'Default')  // move temp object
UserName::~UserName('')             // delete the temp object
UserName::~UserName('Default')
```

The code creates a temporary object and then moves it into the object stored in `optional`.

Here we can use a more efficient constructor - by leveraging `std::in_place_t`:

```
std::optional<UserName> opt{std::in_place};
```

With constructor and destructor traces you'd get the following output:

```
UserName::UserName('Default')
UserName::~UserName('Default')
```

The object stored in the optional is created in place, in the same way as you'd call `UserName{}`. No additional copy or move is needed.

See the example in `Chapter Optional/optional_in_place_default.cpp`. In the file, you'll also see the traces for constructors and destructor.

Non Copyable/Movable Types

As you saw in the example from the previous section, if you use a temporary object to initialise the contained value inside `std::optional` then the compiler will have to use a move or a copy constructor.

But what if your type doesn't allow that? For example, `std::mutex` is not movable or copyable.

In that case, `std::in_place` is the only way to work with such types.

Constructors With Many Arguments

Another use case is a situation where your type has more arguments in a constructor. By default `optional` can work with a single argument (r-value ref), and efficiently pass it to the wrapped type. But what if you'd like to initialise `Point(x, y)`?

You can always create a temporary copy and then pass it in the construction:

Chapter Optional/optional_point.cpp

```
struct Point {
    Point(int a, int b) : x(a), y(b) { }

    int x;
    int y;
};

std::optional<Point> opt{Point{0, 0}}; // temp created!
```

or use `in_place` and the version of the constructor that handles variable argument list:

```
template< class... Args >
constexpr explicit optional( std::in_place_t, Args&&... args );

// or initializer_list:

template< class U, class... Args >
constexpr explicit optional( std::in_place_t,
                             std::initializer_list<U> ilist,
                             Args&&... args );
```

Your code can look like this:

```
std::optional<Point> opt{std::in_place_t, 0, 0};
```

The second option is quite verbose and omits to create temporary objects. Temporaries, especially for containers or larger objects, are not as efficient as constructing in place.

`std::make_optional()`

If you don't like `std::in_place` then you can look at `make_optional` factory function.

The code:

```
auto opt = std::make_optional<UserName>();
auto opt = std::make_optional<Point>(0, 0);
```

Is as efficient as:

```
std::optional<UserName> opt{std::in_place};
std::optional<Point> opt{std::in_place_t, 0, 0};
```

`std::make_optional` implements in place construction equivalent to:

```
return std::optional<T>(std::in_place, std::forward<Args>(args)...);
```

Also due to mandatory copy elision there is no temporary object involved.

Returning `std::optional`

If you return an optional from a function, then it's very convenient to return just `std::nullopt` or the computed value.

Chapter Optional/optional_return_rvo.cpp

```
std::optional<std::string> TryParse(Input input) {
    if (input.valid())
        return input.asString();

    return std::nullopt;
}

// use:
auto oStr = TryParse(Input{...});
```

In the above example, you can see that the function returns `std::string` computed from `input.asString()` and it's wrapped in `optional`. If the value is unavailable, then it returns `std::nullopt`.

Due to mandatory copy elision from C++17 [2] the optional object - `oStr` will be created at the caller site.

Alternatively, you can also try with non-standardised Named Returned Value Optimisation. This happens when you create an object at the beginning of a function and then return it.

[2]Read more in section "Guaranteed Copy Elision" in the General Language Features Chapter

Chapter Optional/optional_return_rvo.cpp

```
std::optional<std::string> TryParseNrvo(Input input) {
    std::optional<std::string> oOut; // empty

    if (input.valid())
        oOut = input.asString();

    return oOut;
}

// use:
auto oStr = TryParseNrvo(Input{...});
```

In the second example, the `oStr` object should also be created at the caller side. Play with the sample, which includes extra logging to check the addresses of the optional variable.

Be Careful With Braces when Returning

You might be surprised by the following code[3]:

```
std::optional<std::string> CreateString() {
   std::string str {"Hello Super Awesome Long String"};
   return {str}; // this one will cause a copy
//   return str;   // this one moves
}
```

According to the Standard if you wrap a return value into braces `{}` then you prevent move operations from happening. The returned object will be copied only.

This is similar to the case with non-copyable types:

```
std::unique_ptr<int> foo() {
    std::unique_ptr<int> p;
    return {p};  // tries to copy a unique_ptr and it fails to compile
//    return p;  // this one moves, so it's fine with unique_ptr
}
```

[3]Thanks to JFT for pointing that problem out.

The Standard says [class.copy.elision]/3[4]

> In the following copy-initialisation contexts, a move operation might be used instead of a copy operation:
>
> - If the expression in a return statement ([stmt.return]) is a **(possibly parenthesised) id-expression** that names an object with automatic storage duration declared in the body or parameter-declaration-clause of the innermost enclosing function or lambda-expression, or
> - if the operand of a throw-expression is the name of a non-volatile automatic object (other than a function or catch-clause parameter) whose scope does not extend beyond the end of the innermost enclosing try-block (if there is one),

Try playing with the example that is located in:

`Chapter Optional/optional_return.cpp`.

The code shows a few examples with `std::unique_ptr`, `std::vector`, `std::string` and a custom type.

Accessing The Stored Value

Probably the most important operation for optional (apart from creation) is the way you can fetch the contained value.

There are several options:

- `operator*` and `operator->` - if there's no value the behaviour is **undefined**!
- `value()` - returns the value, or throws `std::bad_optional_access`
- `value_or(defaultVal)` - returns the value if available, or `defaultVal` otherwise

To check if the value is present, you can use the `has_value()` method or check `if (optional)` as optional is contextually convertible to `bool`.

[4] hhttps://timsong-cpp.github.io/cppwp/n4659/class.copy.elision#3

Here's an example that shows those possibilites:

```
// by operator*
std::optional<int> oint = 10;
std::cout<< "oint " << *oint << '\n';

// by value()
std::optional<std::string> ostr("hello");
try {
    std::cout << "ostr " << ostr.value() << '\n';
}
catch (const std::bad_optional_access& e) {
    std::cout << e.what() << '\n';
}

// by value_or()
std::optional<double> odouble; // empty
std::cout<< "odouble " << odouble.value_or(10.0) << '\n';
```

And here's a handy pattern that checks if the value is present and then accesses it:

```
// compute string function:
std::optional<std::string> maybe_create_hello();
// ...

if (auto ostr = maybe_create_hello(); ostr)
    std::cout << "ostr " << *ostr << '\n';
else
    std::cout << "ostr is null\n";
```

`std::optional` Operations

Let's see what other operations are available for the type.

Changing the Value & Object Lifetime

If you have an existing optional object, then you can quickly change the contained value by using several operations like `emplace`, `reset`, `swap`, assign. If you assign (or reset) with a `nullopt` then if the optional contains a value, its destructor will be called.

Here's an example that shows all of the cases:

Chapter Optional/optional_reset.cpp

```
#include <optional>
#include <iostream>
#include <string>

class UserName {
public:
    explicit UserName(std::string str) : mName(std::move(str)) {
        std::cout << "UserName::UserName('" << mName << "')\n";
    }
    ~UserName() {
        std::cout << "UserName::~UserName('" << mName << "')\n";
    }

private:
    std::string mName;
};

int main() {
    std::optional<UserName> oEmpty;

    // emplace:
    oEmpty.emplace("Steve");

    // calls ~Steve and creates new Mark:
    oEmpty.emplace("Mark");

    // reset so it's empty again
    oEmpty.reset(); // calls ~Mark
    // same as:
    //oEmpty = std::nullopt;

    // assign a new value:
    oEmpty.emplace("Fred");
    oEmpty = UserName("Joe");
}
```

Each time the object is changed, a destructor of the currently stored `UserName` is called.

Comparisons

`std::optional` allows you to compare contained objects almost "naturally", but with a few exceptions when the operands are `nullopt`. See below:

Chapter Optional/optional_comparision.cpp

```
#include <optional>
#include <iostream>

int main() {
    std::optional<int> oEmpty;
    std::optional<int> oTwo(2);
    std::optional<int> oTen(10);

    std::cout << std::boolalpha;
    std::cout << (oTen > oTwo) << '\n';
    std::cout << (oTen < oTwo) << '\n';
    std::cout << (oEmpty < oTwo) << '\n';
    std::cout << (oEmpty == std::nullopt) << '\n';
    std::cout << (oTen == 10) << '\n';
}
```

The above code generates:

```
true  // (oTen > oTwo)
false // (oTen < oTwo)
true  // (oEmpty < oTwo)
true  // (oEmpty == std::nullopt)
true  // (oTen == 10)
```

When operands contain values (of the same type), then you'll see the expected results. But when one operand is `nullopt` then it's always "less" than any optional with some value.

Performance & Memory Consideration

When you use `std::optional` you'll pay with an increased memory footprint.

The `optional` class wraps your type, prepares space for it and then adds one boolean parameter. This means it will extend the size of your type according to the alignment rules.

Conceptually your version of the standard library might implement optional as:

```
template <typename T>
class optional {
  bool _initialized;
  std::aligned_storage_t<sizeof(t), alignof(T)> _storage;

public: // operations
};
```

Alignment rules for the optional are defined as follows in optional.optional[5]:

> The contained value shall be allocated in a region of the optional storage suitably aligned for the type T.

For example, assuming `sizeof(double) = 8` and `sizeof(int) = 4`:

```
std::optional<double> od; // sizeof = 16 bytes
std::optional<int> oi;    // sizeof = 8 bytes
```

While the `bool` type usually takes only one byte, the optional type needs to obey the alignment rules, and it's larger than just `sizeof(YourType) + 1 byte`.

For example, if you have two types:

```
struct Range {
    std::optional<double> mMin;
    std::optional<double> mMax;
};

struct RangeCustom {
    bool mMinAvailable;
    bool mMaxAvailable;
    double mMin;
    double mMax;
};
```

[5] https://timsong-cpp.github.io/cppwp/n4659/optional.optional#1

The `Range` takes up more space than `RangeCustom`. In the first case, we're using 32 bytes! The second version is 24 bytes. This is because the second class can "pack" boolean variables at the front of the structure, while two optional objects in `Range` has to align to `double`.

You can see the full code in `Chapter Optional/optional_sizeof.cpp`.

Migration from `boost::optional`

`std::optional` was adapted directly from `boost::optional`, thus you should expect the same experience in both versions. Moving from one to the other should be easy, but of course, there are little differences.

The table below summarises the changes:

aspect	**std::optional**	**boost::optional (as of 1.67.0[6])**
Move semantics	yes	yes
noexcept	yes	yes
hash support	yes	no
a throwing value accessor	yes	yes
literal type (can be used in `constexpr` expressions)	yes	no
in place construction	`` `emplace`, tag `in_place` ``	`emplace()`, tags `in_place_init_if_t`, `in_place_init_t`, utility `in_place_factory`
disengaged state tag	`nullopt`	`none`
optional references	no	yes
conversion from `optional<U>` to `optional<T>`	yes	yes
explicit convert to ptr (`get_ptr`)	no	yes
deduction guides	yes	no

The main difference is that `std::optional` supports hashing, can be used in `constexpr` contexts and also has deduction guides. `boost::optional` however supports references which is blocked in C++17 version [7].

[7]read more in "Why Optional References Didn't Make It In C++17" at https://www.fluentcpp.com/2018/10/05/pros-cons-optional-references/

Special case: `optional<bool>` and `optional<T*>`

While you can use optional on any type, you need to pay attention to boolean and pointers.

`optional<bool>` - what does it model? With such a construction, you have a tri-state bool. If you need it, then maybe it's better to look for a real tri-state bool like `boost::tribool`[8].

What's more, it might be confusing to use such type because `optional<bool>` converts to `bool`. Also, if there's a value inside then accessing that value returns `bool`.

Likewise, you have a similar ambiguity with pointers:

```
// Don't try doing it this way, it's just an example!
std::optional<int*> opi { new int(10) };
if (opi && *opi) {
   std::cout << **opi << std::endl;
   delete *opi;
}

if (opi)
    std::cout << "opi is still not empty!";
```

In the above example, you have to check `opi` to see if the optional is empty or not, but then the value of `opi` can also be `nullptr`.

The pointer to `int` is naturally "nullable", wrapping it into optional makes it confusing to use.

Examples of `std::optional`

Here are a few more extended examples where `std::optional` fits nicely.

In the first example, you'll see how to use `optional` in a class. In the second sample, we'll cover parsing of integers and storing the result in an optional.

[8]https://www.boost.org/doc/libs/1_67_0/doc/html/tribool.html

User Name with an Optional Nickname and Age

Chapter Optional/optional_user_name.cpp

```cpp
#include <optional>
#include <iostream>
using namespace std;

class UserRecord {
public:
    UserRecord (string name, optional<string> nick, optional<int> age)
    : mName{move(name)}, mNick{move(nick)}, mAge{age}
    { }

    friend ostream& operator << (ostream& stream, const UserRecord& user);

private:
    string mName;
    optional<string> mNick;
    optional<int> mAge;
};

ostream& operator << (ostream& os, const UserRecord& user) {
    os << user.mName;

    if (user.mNick)
        os << ' ' << *user.mNick;
    if (user.mAge)
        os << ' ' << "age of " << *user.mAge;

    return os;
}

int main() {
    UserRecord tim { "Tim", "SuperTim", 16 };
    UserRecord nano { "Nathan", nullopt, nullopt };

    cout << tim << '\n';
    cout << nano << '\n';
}
```

The above example shows a simple class with optional fields. While the name is obligatory, the other attributes: “nickname” and “age” are optional.

Parsing `int`s From the Command Line

Chapter Optional/optional_parsing.cpp

```cpp
#include <optional>
#include <iostream>
#include <string>

std::optional<int> ParseInt(const char* arg) {
    try {
        return { std::stoi(std::string(arg)) };
    }
    catch (...) {
        std::cerr << "cannot convert '" << arg << "' to int!\n";
    }

    return { };
}

int main(int argc, const char* argv[]) {
    if (argc >= 3) {
        auto oFirst = ParseInt(argv[1]);
        auto oSecond = ParseInt(argv[2]);

        if (oFirst && oSecond) {
            std::cout << "sum of " << *oFirst << " and " << *oSecond;
            std::cout << " is " << *oFirst + *oSecond << '\n';
        }
    }
}
```

The above code uses `optional` to indicate whether we performed the conversion or not. Please note that we actually converted exception handling into `optional`, we suppress exceptions that might appear in the conversion process. Such a technique might look “controversial”.

The code uses `stoi` which might be replaced with new low-level functions `from_chars`. You can read more about new conversion utilities in the String Conversions Chapter.

Other Examples

Here are a few more ideas where you might use `std::optional`:

- Representing optional configuration values
- Geometry & Math: finding if there's an intersection between objects
- Return values for `Find*()` functions (assuming you don't care about errors, like connection drops, database errors or something)

You may find other exciting uses in the post: A Wall of Your std::optional Examples[9]. The blog post contains a lot of examples submitted by blog readers.

Summary

A few core elements to know about `std::optional`:

- `std::optional` is a wrapper type to express "null-able" types
- `std::optional` won't use any dynamic allocation
- `std::optional` contains a value or it's empty
  - use `operator *`, `operator->`, `value()` or `value_or()` to access the underlying value.
- `std::optional` is implicitly converted to `bool` so that you can easily check if it contains a value or not

Compiler Support

Feature	GCC	Clang	MSVC
`std::optional`	7.1	4.0	VS 2017 15.0

[9]https://www.bfilipek.com/2018/06/optional-examples-wall.html

8. `std::variant`

Another handy wrapper type that we get in C++17 is `std::variant`. This is a type-safe union - you can store different type variants with the proper object lifetime guarantee. The new type offers a huge advantage over the C-style union. You can store all of the types inside - no matter if it's something simple like `int`, or `float`, but also complex entities like `std::vector<std::string>`. In all of the cases, objects will be correctly initialised and cleaned up.

What's crucial is the fact that the new type enhances implementation of design patterns. For example, you can now use a visitor, pattern matching and runtime polymorphism for unrelated type hierarchies in a much easier way.

In this chapter, you'll learn:

- What problems can occur with unions
- What discriminated unions are, and why we need type-safety with unions
- How `std::variant` works and what it does
- Operations on `std::variant`
- Performance cost and memory requirements
- Example use cases

The Basics

Unions are rarely used in the client code, and most of the time, they should be avoided.

For example, there's a "common" trick with floating-point operations:

```
union SuperFloat {
    float f;
    int i;
}

int RawMantissa(SuperFloat f) {
    return f.i & ((1 << 23) - 1);
}
int RawExponent(SuperFloat f) {
    return (f.i >> 23) & 0xFF;
}
```

However, while the above code might work in C99, due to stricter aliasing rules it's undefined behaviour in C++!

There's an existing Core Guideline Rule on that C.183[1]:

> **C.183: Don't use a `union` for type punning[a]:**
>
> It is undefined behaviour to read a `union` member with a different type from the one with which it was written. Such punning is invisible, or at least harder to spot than using a named cast. Type punning using a `union` is a source of errors.
>
> [a]http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c183-dont-use-a-union-for-type-punning

Before C++17, if you wanted a type-safe union, you could use `boost::variant` or another third-party library. But now you have `std::variant`.

Here's a demo of what you can do with this new type:

[1]http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c183-dont-use-a-union-for-type-punning

Chapter Variant/variant_demo.cpp

```cpp
#include <string>
#include <iostream>
#include <variant>

using namespace std;

// used to print the currently active type
struct PrintVisitor {
    void operator()(int i) { cout << "int: " << i << '\n'; }
    void operator()(float f) { cout << "float: " << f << '\n'; }
    void operator()(const string& s) { cout << "str: " << s << '\n'; }
};

int main() {
    variant<int, float, string> intFloatString;
    static_assert(variant_size_v<decltype(intFloatString)> == 3);

    // default initialised to the first alternative, should be 0
    visit(PrintVisitor{}, intFloatString);

    // index will show the currently used 'type'
    cout << "index = " << intFloatString.index() << endl;
    intFloatString = 100.0f;
    cout << "index = " << intFloatString.index() << endl;
    intFloatString = "hello super world";
    cout << "index = " << intFloatString.index() << endl;

    // try with get_if:
    if (const auto intPtr = get_if<int>(&intFloatString))
        cout << "int: " << *intPtr << '\n';
    else if (const auto floatPtr = get_if<float>(&intFloatString))
        cout << "float: " << *floatPtr << '\n';

    if (holds_alternative<int>(intFloatString))
        cout << "the variant holds an int!\n";
    else if (holds_alternative<float>(intFloatString))
        cout << "the variant holds a float\n";
    else if (holds_alternative<string>(intFloatString))
        cout << "the variant holds a string\n";

    // try/catch and bad_variant_access
    try {
```

```
43            auto f = get<float>(intFloatString);
44            cout << "float! " << f << '\n';
45        }
46        catch (bad_variant_access&) {
47            cout << "our variant doesn't hold float at this moment...\n";
48        }
49    }
```

The output:

```
int: 0
index = 0
index = 1
index = 2
the variant holds a string
our variant doesn't hold float at this moment...
```

Several points are worth examining in the example above:

- Line 15, 19: If you don't initialise a variant with a value, then the variant is initialised with the first type. In that case, the first alternative type must have a default constructor. Line 22 will print the value 0.
- Line: 22, 24, 26, 34, 36, 38: You can check what the currently used type is via `index()` or `holds_alternative`.
- Line 29, 31: You can access the value by using `get_if` - it returns null pointer when the type is not active.
- Line 43, 46: You can access the value by using `get` (the compiler might throw the `bad_variant_access` exception).
- Type Safety - the variant doesn't allow you to get a value of the type that's not active.
- No extra heap allocation occurs.
- Line 8, 19: You can use a visitor to invoke an action on a currently active type. The example uses `PrintVisitor` to print the currently active value. It's a simple structure with overloads for `operator()`. The visitor is then passed to `std::visit` which performs the visitation.
- The variant class calls destructors and constructors of non-trivial types, so in the example, the `string` object is cleaned up before we switch to new variants.

When to Use

Unless you're doing some low-level stuff, possibly only with simple types, then unions might be a valid option[2]. But for all other uses cases, where you need alternative types, `std::variant` is the way to go.

Some possible uses:

- All the places where you might get a few types for a single field: so things like parsing command lines, ini files, language parsers, etc.
- Expressing several possible outcomes of a computation efficiently: like finding roots of equations.
- Error handling - for example you can return `variant<Object, ErrorCode>`. If the value is available, then you return `Object` otherwise you assign some error code.
- Finite State Machines.
- Polymorphism without `vtables` and inheritance (thanks to the visitor pattern).

A Functional Background

It's also worth mentioning that variant types (also called a tagged union, a discriminated union, or a sum type) come from the functional language world and Type Theory[3].

`std::variant` Creation

There are several ways you can create and initialize `std::variant`:

[2]See C++ Core Guidelines - Unions for examples of a valid use cases for unions.
[3]https://en.wikipedia.org/wiki/Algebraic_data_type

Chapter Variant/variant_creation.cpp

```
// default initialisation: (the first type has to have a default ctor)
std::variant<int, float> intFloat;
std::cout << intFloat.index() << ", val: " << std::get<int>(intFloat) << '\n';

// monostate for default initialisation:
class NotSimple {
public:
    NotSimple(int, float) { }
};

// std::variant<NotSimple, int> cannotInit; // error
std::variant<std::monostate, NotSimple, int> okInit;
std::cout << okInit.index() << '\n';

// pass a value:
std::variant<int, float, std::string> intFloatString { 10.5f };
std::cout << intFloatString.index()
          << ", value " << std::get<float>(intFloatString) << '\n';

// ambiguity
// double might convert to float or int, so the compiler cannot decide

//std::variant<int, float, std::string> intFloatString { 10.5 };

// ambiguity resolved by in_place
variant<long, float, std::string> longFloatString {
        std::in_place_index<1>, 7.6 // double!
};
std::cout << longFloatString.index() << ", value "
          << std::get<float>(longFloatString) << '\n';

// in_place for complex types
std::variant<std::vector<int>, std::string> vecStr {
    std::in_place_index<0>, { 0, 1, 2, 3 }
};
std::cout << vecStr.index() << ", vector size "
          << std::get<std::vector<int>>(vecStr).size() << '\n';

// copy-initialize from other variant:
std::variant<int, float> intFloatSecond { intFloat };
std::cout << intFloatSecond.index() << ", value "
          << std::get<int>(intFloatSecond) << '\n';
```

- By default, a variant object is initialised with the first type
  - if that's not possible when the type doesn't have a default constructor, then you'll get a compiler error
  - you can use `std::monostate` to pass it as the first type in that case
  - `std::monostate` allows you to build a variant with "no-value" so it can behave similarly to `std::optional`
- You can initialise it with a value, and then the best matching type is used
  - if there's an ambiguity, then you can use a version `std::in_place_index` to explicitly mention what type should be used
- `std::in_place` also allows you to create more complex types and pass more parameters to the constructor

About `std::monostate`

In the example, you might notice a special type called `std::monostate`. This is just an empty type that can be used with variants to represent an empty state. The type might be handy when the first alternative doesn't have a default constructor. In that situation, you can place `std::monostate` as the first alternative (or you can also shuffle the types, and find the type with a default constructor).

In Place Construction

`std::variant` has two `in_place` helpers that you can use:

- `std::in_place_type` - used to specify which type you want to change/set in the variant
- `std::in_place_index` - used to specify which index you want to change/set. Types are enumerated from `0`.
  - In a variant `std::variant<int, float, std::string>` - `int` has the index `0`, `float` has index `1` and the string has index of `2`. The index is the same value as returned from `variant::index` method.

Fortunately, you don't always have to use the helpers to create a variant. It's smart enough to recognise if it can be constructed from the passed single parameter:

```
// this constructs the second/float:
std::variant<int, float, std::string> intFloatString { 10.5f };
```

For a variant we need the helpers for at least two cases:

- ambiguity - to distinguish which type should be created where several could match
- efficient complex type creation (similar to optional)

Ambiguity

What if you have an initialisation like:

```
std::variant<int, float> intFloat { 10.5 }; // conversion from double?
```

The value `10.5` could be converted to `int` or `float`, and the compiler doesn't know which conversion should be applied. It might report a few pages of compiler errors. You can easily handle such errors by specifying which type you'd like to create:

```
std::variant<int, float> intFloat { std::in_place_index<0>, 10.5 };
// or
std::variant<int, float> intFloat { std::in_place_type<int>, 10.5 };
```

Complex Types

Similarly to `std::optional`, if you want to efficiently create objects that require several constructor arguments - then use `std::in_place_index` or `std::in_place_type`:

For example:

```
std::variant<std::vector<int>, std::string> vecStr {
    std::in_place_index<0>, { 0, 1, 2, 3 } // initializer list passed into vector
};
```

Unwanted Type Conversions And Narrowing

`std::variant` in the first implementations used regular C++ rules for converting constructors and assignment operator. In a case when a conversion was required, the compiler preferred narrowing conversions which might not be what you expected.

For example:

```
std::variant<std::string, int, bool> vStrIntBool = "Hello World";
```

The above line created a variant with `bool` as the active type, not `std::string`.

The string literal `"Hello World"` is not the exact type that appears in `vStrIntBool`, so the conversion has to happen. The compiler sees two possible conversions: one from `const char*` into `bool` and then from `const char*` into `std::string`. Since `bool` is the built-in type, the compiler will select it.

There's another case with narrowing conversions:

```
variant<float, long, double> v = 0;
```

Before the fix, this line won't compile (we have several narrowing conversions possible), but after the improvement, it will hold `long`.

The implementation was improved through the fix from P0608: A sane variant converting constructor[4] and is ready since GCC 10.0.

Below you can see a table that shows what type will be selected for a given expression, we have a colum before and after the fix (P0608):

Expression	Before Fix	After Fix
1. `variant<bool, string> v = "Hello"`	`bool`	`string`
2. `variant<float, optional<double>> x = 10.05`	`float`	`optional`
3. `variant<float, char> v = 0`	ill-formed	ill-formed
4. `variant<float, long> v = 0`	ill-formed	selects `long`

Notes:

1. the narrowing `bool` conversion is not taken into account now, and `string` is selected
2. prefers non-narrowing conversion into `std::optional`
3. both narrowing conversions required so the whole expression won't compile
4. before the fix the two conversions were possible after the fix `float` is not considered

Try to match the exact type that is available in a given `std::variant` to limit the case of unexpected conversions.

[4] https://wg21.link/p0608

Changing the Values

There are four ways to change the current value of the variant:

- the assignment operator
- `emplace`
- `get` and then assign a new value for the currently active type
- a visitor (you cannot change the type, but you can change the value of the current alternative)

The important part is to know that everything is type-safe and also that the object lifetime is honoured.

Chapter Variant/variant_changing_values.cpp

```
std::variant<int, float, std::string> intFloatString { "Hello" };

intFloatString = 10; // we're now an int

intFloatString.emplace<2>(std::string("Hello")); // we're now string again

// std::get returns a reference, so you can change the value:
std::get<std::string>(intFloatString) += std::string(" World");

intFloatString = 10.1f;
if (auto pFloat = std::get_if<float>(&intFloatString); pFloat)
    *pFloat *= 2.0f;
```

Object Lifetime

When you use `union`, you need to manage the internal state: call constructors or destructors. This is error-prone, and it's easy to shoot yourself in the foot. But `std::variant` handles object lifetime as you expect. That means that if it's about to change the currently stored type, then a destructor of the underlying type is called.

```
std::variant<std::string, int> v { "Hello A Quite Long String" };
// v allocates some memory for the string
v = 10; // we call destructor for the string!
// no memory leak
```

Or see this example with a custom type:

Chapter Variant/variant_lifetime.cpp

```
class MyType {
public:
    MyType() { std::cout << "MyType::MyType\n"; }
    ~MyType() { std::cout << "MyType::~MyType\n"; }
};

class OtherType {
public:
    OtherType() { std::cout << "OtherType::OtherType\n"; }
    ~OtherType() { std::cout << "OtherType::~OtherType\n"; }
};

int main() {
    std::variant<MyType, OtherType> v;
    v = OtherType();

    return 0;
}
```

This will produce the following output:

```
MyType::MyType
OtherType::OtherType
MyType::~MyType
OtherType::~OtherType
OtherType::~OtherType
```

At the start, we initialise with a default value of type `MyType`; then we change the value with an instance of `OtherType`, and before the assignment, the destructor of `MyType` is called. Later we destroy the temporary object and the object stored in the variant.

Accessing the Stored Value

From all of the examples you've seen so far, you might get an idea of how to access the value. But let's make a summary of this vital operation.

First of all, even if you know what the currently active type is you cannot do:

```
std::variant<int, float, std::string> intFloatString { "Hello" };
std::string s = intFloatString;

// error: conversion from
// 'std::variant<int, float, std::string>'
// to non-scalar type 'std::string' requested
// std::string s = intFloatString;
```

So you have to use helper functions to access the value.

You have `std::get<Type|Index>(variant)` which is a non member function. It returns a reference to the desired type if it's active (you can pass a Type or Index). If not then you'll get `std::bad_variant_access` exception.

```
std::variant<int, float, std::string> intFloatString;
try {
    auto f = std::get<float>(intFloatString);
    std::cout << "float! " << f << '\n';
}
catch (std::bad_variant_access&) {
    std::cout << "our variant doesn't hold float at this moment...\n";
}
```

The next option is `std::get_if`. This function is also a non-member and won't throw. It returns a pointer to the active type or `nullptr`. While `std::get` needs a reference to the variant, `std::get_if` takes a pointer.

```
if (const auto intPtr = std::get_if<0>(&intFloatString))
    std::cout << "int!" << *intPtr << '\n';
```

However, probably the most important way to access a value inside a variant is by using visitors.

Visitors for `std::variant`

With the introduction of `std::variant`, we also got a handy STL function called `std::visit`.

It can call a given "visitor" on all passed variants.

Here's the declaration:

```
template <class Visitor, class... Variants>
constexpr visit(Visitor&& vis, Variants&&... vars);
```

And it will call `vis` on the currently active type of variants.

If you pass only one variant, then you have to have overloads for the types from that variant. If you give two variants, then you have to have overloads for all possible *pairs* of the types from the variants.

A visitor is "a Callable that accepts every possible alternative from every variant".

Let's see some examples:

```
// a generic lambda:
auto PrintVisitor = [](const auto& t) { std::cout << t << '\n'; };

std::variant<int, float, std::string> intFloatString { "Hello" };
std::visit(PrintVisitor, intFloatString);
```

In the above example, a generic lambda is used to generate all possible overloads. Since all of the types in the variant support `<<` (stream output operator) then we can print them.

In another example we can use a visitor to change the value:

```
auto PrintVisitor = [](const auto& t) { std::cout << t << '\n'; };
auto TwiceMoreVisitor = [](auto& t) { t*= 2; };

std::variant<int, float> intFloat { 20.4f };
std::visit(PrintVisitor, intFloat);
std::visit(TwiceMoreVisitor, intFloat);
std::visit(PrintVisitor, intFloat);
```

Generic lambdas can work if our types share the same "interface", but in most cases, we'd like to perform different actions based on an active type.

That's why we can define a structure with several overloads for the `operator()`:

```
struct MultiplyVisitor {
    float mFactor;

    MultiplyVisitor(float factor) : mFactor(factor) { }

    void operator()(int& i) const {
        i *= static_cast<int>(mFactor);
    }

    void operator()(float& f) const {
        f *= mFactor;
    }

    void operator()(std::string& ) const {
        // nothing to do here...
    }
};

std::visit(MultiplyVisitor(0.5f), intFloat);
std::visit(PrintVisitor, intFloat);
```

In the example, you might notice that `MultiplyVisitor` uses a state to hold the desired scaling factor value. That gives a lot of options for visitation.

With lambdas, we got used to declaring things just next to its usage. And when you need to write a separate structure, you need to go out of that local scope. That's why it might be handy to use `overload` construction.

Overload

With this utility you can write all lambdas for all matching types in one place:

```
std::variant<int, float, std::string> myVariant;
std::visit(
  overload {
    [](const int& i) { std::cout << "int: " << i; },
    [](const std::string& s) { std::cout << "string: " << s; },
    [](const float& f) { std::cout << "float: " << f; }
  },
  myVariant
);
```

Currently this helper is not a part of the Standard Library (it might be added into with C++20). You can implement it with the following code:

```
template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;
```

The code creates a `struct` that inherits from lambdas and uses their `Ts::operator()`. The whole structure can now be passed to `std::visit` - it will then select the proper overload.

`overload` uses three C++17 features:

- Pack expansions in `using` declarations - short and compact syntax with variadic templates.
- Custom template argument deduction rules - this allows the compiler to deduce types of lambdas that are the base classes for the pattern. Without it, we'd have to define a "make" function.
- Extension to aggregate Initialisation - the overload pattern uses aggregate initialisation to init base classes. Before C++17, it was not possible.

Here's another example of how to use the overload pattern:

Chapter Variant/variant_overload.cpp

```
std::variant<int, float, std::string> intFloatString { "Hello" };
std::visit(overload{
    [](int& i) { i*= 2; },
    [](float& f) { f*= 2.0f; },
    [](std::string& s) { s = s + s; }
}, intFloatString);
std::visit(PrintVisitor, intFloatString);
// prints: "HelloHello"
```

Here's the paper for the proposal of `std::overload`: P0051 - C++ generic overload function[5].

And you can read more about the mechanics of the overload pattern in this blog post at bfilipek.com: 2 Lines Of Code and 3 C++17 Features - The overload Pattern[6]

Visiting Multiple Variants

`std::visit` allows you not only to visit one variant but many in the same call. However, it's essential to know that with multiple variants, you have to implement function overloads taking as many arguments as the number of input variants. And you have to provide all the possible combination of types.

For example for:

```
std::variant<int, float, char> v1 { 's' };
std::variant<int, float, char> v2 { 10 };
```

You have to provide 9 function overloads if you call `std::visit` on the two variants:

[5]https://wg21.link/p0051r3
[6]https://www.bfilipek.com/2019/02/2lines3featuresoverload.html

```
std::visit(overload{
    [](int a, int b) { },
    [](int a, float b) { },
    [](int a, char b) { },
    [](float a, int b) { },
    [](float a, float b) { },
    [](float a, char b) { },
    [](char a, int b) { },
    [](char a, float b) { },
    [](char a, char b) { }
}, v1, v2);
```

If you skip one overload, then the compiler will report an error.

Have a look at the following example, where each variant represents an ingredient and we want to compose two of them together:

Chapter Variant/visit_multiple.cpp

```
#include <iostream>
#include <variant>

template<class... Ts> struct overload : Ts... { using Ts::operator()...; };
template<class... Ts> overload(Ts...) -> overload<Ts...>;

struct Pizza { };
struct Chocolate { };
struct Salami { };
struct IceCream { };

int main() {
    std::variant<Pizza, Chocolate, Salami, IceCream> firstIngredient{IceCream()};
    std::variant<Pizza, Chocolate, Salami, IceCream> secondIngredient{Chocolate()};

    std::visit(overload{
        [](const Pizza& p, const Salami& s) {
                std::cout << "here you have, Pizza with Salami!\n";
            },
        [](const Salami& s, const Pizza& p) {
                std::cout << "here you have, Pizza with Salami!\n";
            },
        [](const Chocolate& c, const IceCream& i) {
                std::cout << "Chocolate with IceCream!\n";
            },
        [](const IceCream& i, const Chocolate& c) {
                std::cout << "IceCream with a bit of Chocolate!\n";
            },
        [](const auto& a, const auto& b) {
                std::cout << "invalid composition...\n";
            },
    }, firstIngredient, secondIngredient);

    return 0;
}
```

The code will output: `IceCream with a bit of Chocolate!`

The above code uses `overload` and uses multiple lambdas rather than a separate `struct` with overloads for `operator()`.

What's interesting is that the example provides implementation only for "valid" ingredient compositions, while the "rest" is handled by generic lambdas (from C++14).

A generic lambda `[](const auto& a, const auto& b) { }` is equivalent to the following callable type:

```
class UnnamedUniqueClass { // << compiler specific name...
public:
    template<typename T, typename U>
    auto operator () (const T& a, const T& b) const {  }
};
```

The generic lambda used in the example will provide all the remaining function overloads for the ingredient types. Since it's a template, it will always fall behind the concrete overload (lambda with concrete types) when the best viable function is determined.

Other `std::variant` Operations

Just for the sake of completeness:

- You can **compare** two variants of the same type:
  - if they contain the same active alternative, then the corresponding comparison operator is called.

  - If one variant has an "earlier" alternative then it's "less than" the variant with the next active alternative.

- Variant is a value type, so you can **move it**.
- `std::hash` on a variant is specialised if `std::hash` is available for all type alternatives. The hash value might be different than a hash for an active type as this enables to distinguish between variants that have duplicated types like `std::variant<int, int, float>`.

Exception Safety Guarantees

So far everything looks nice and smooth… but what happens when there's an exception during the creation of the alternative in a variant?

For example:

Chapter Variant/variant_valueless.cpp

```
class ThrowingClass {
public:
    explicit ThrowingClass(int i) { if (i == 0) throw int (10); }
    operator int () { throw int(10); }
};

int main(int argc, char** argv) {
    std::variant<int, ThrowingClass> v;

    // change the value:
    try {
        v = ThrowingClass(0);
    }
    catch (...) {
        std::cout << "catch(...)\n";
        // we keep the old state!
        std::cout << v.valueless_by_exception() << '\n';
        std::cout << std::get<int>(v) << '\n';
    }

    // inside emplace
    try {
        v.emplace<0>(ThrowingClass(10)); // calls the operator int
    }
    catch (...) {
        std::cout << "catch(...)\n";
        // the old state was destroyed, so we're not in invalid state!
        std::cout << v.valueless_by_exception() << '\n';
    }

    return 0;
}
```

In the first case - with the assignment operator - the exception is thrown in the constructor of the type. This happens before the old value is replaced in the variant, so the variant state is unchanged. As you can see we can still access `int` and print it.

However, in the second case - `emplace` - the exception is thrown after the old state of the variant is destroyed. `emplace` calls `operator int` to replace the value, but that throws. After that, the variant is in a wrong state, and we cannot recover the previous state.

Also note that a variant that is “valueless by exception” is in an invalid state. Accessing a value from such variant is not possible. That’s why `variant::index` returns `variant_npos`, and `std::get` and `std::visit` will throw `bad_variant_access`.

Performance & Memory Considerations

`std::variant` uses the memory in a similar way to union: so it will take the max size of the underlying types. But since we need something that will know what the currently active alternative is, then we need to use some more space. Plus everything needs to honour the alignment rules.

Here are some basic sizes:

Chapter Variant/variant_sizeof.cpp

```
std::cout << "sizeof string: "
          << sizeof(std::string) << '\n';

std::cout << "sizeof variant<int, string>: "
          << sizeof(std::variant<int, std::string>) << '\n';

std::cout << "sizeof variant<int, float>: "
          << sizeof(std::variant<int, float>) << '\n';
```

```
std::cout << "sizeof variant<int, double>: "
          << sizeof(std::variant<int, double>) << '\n';
```

On GCC 8.1, 32 bit:

```
sizeof string: 32
sizeof variant<int, string>: 40
sizeof variant<int, float>: 8
sizeof variant<int, double>: 16
```

What's more interesting is that `std::variant` won't allocate **any extra space**! No dynamic allocation happens to hold variants or the discriminator.

To have a safe sum type, you pay with an increased memory footprint. The additional bits might influence CPU caches. That's why you might want to do some benchmarking for the hot spots in your application that uses variants.

Migration From `boost::variant`

Boost Variant was introduced around the year 2004, so it was 13 years of experience before `std::variant` was added into the Standard. The STL type draws from the experience of the boost version and improves it.

Here are the main changes:

Feature	Boost.Variant (1.67.0)[7]	`std::variant`
Extra memory allocation	Possible on assignment, see Design Overview - Never Empty[8]	No
visiting	apply_visitor	std::visit
get by index	no	yes
recursive variant	yes, see make_recursive_variant[9]	no
duplicated entries	no	yes
empty alternative	`boost::blank`	`std::monostate`

[8]https://www.boost.org/doc/libs/1_67_0/doc/html/variant/design.html
[9]https://www.boost.org/doc/libs/1_67_0/doc/html/boost/make_recursive_variant.html

Examples of `std::variant`

Having learned most of the `std::variant` details, we can now explore a few examples.

Error Handling

The basic idea is to wrap the possible return type with some `ErrorCode`, and that way allow functions to output more information about the errors. Without using exceptions or output parameters. This is similar to what `std::expected` - a new type planned for the future C++ Standard.

Chapter Variant/variant_error_handling.cpp

```
enum class ErrorCode {
    Ok,
    SystemError,
    IoError,
    NetworkError
};

std::variant<std::string, ErrorCode> FetchNameFromNetwork(int i) {
    if (i == 0)
        return ErrorCode::SystemError;

    if (i == 1)
        return ErrorCode::NetworkError;

    return std::string("Hello World!");
}

int main() {
    auto response = FetchNameFromNetwork(0);
    if (std::holds_alternative<std::string>(response))
        std::cout << std::get<std::string>(response) << "n";
    else
        std::cout << "Error!\n";

    response = FetchNameFromNetwork(10);
    if (std::holds_alternative<std::string>(response))
        std::cout << std::get<std::string>(response) << "n";
    else
        std::cout << "Error!\n";
    return 0;
}
```

In the example, `ErrorCode` or a regular type is returned.

Parsing a Command Line

Command line might contain text arguments that could be interpreted in a few ways:

- as an integer
- as a floating-point
- as a boolean flag
- as a string (not parsed)
- or some other types…

We can build a variant that will hold all the possible options.

Here's a simple version with `int`, `float` and `string`:

Chapter Variant/variant_parsing_int_float.cpp

```
class CmdLine {
public:
    using Arg = std::variant<int, float, std::string>;

private:
    std::map<std::string, Arg> mParsedArgs;

public:
    explicit CmdLine(int argc, const char** argv) { ParseArgs(argc, argv); }

    std::optional<Arg> Find(const std::string& name) const;

    // ...
};
```

And the parsing code:

Chapter Variant/variant_parsing_int_float.cpp

```
CmdLine::Arg TryParseString(std::string_view sv) {
    // try with float first
    float fResult = 0.0f;
    const auto last = sv.data() + sv.size();
    const auto res = std::from_chars(sv.data(), last, fResult);
    if (res.ec != std::errc{} || res.ptr != last) {
        // if not possible, then just assume it's a string
        return std::string{sv};
    }

    // no fraction part? then just cast to integer
    if (static_cast<int>(fResult) == fResult)
        return static_cast<int>(fResult);

    return fResult;
}

void CmdLine::ParseArgs(int argc, const char** argv) {
    // the form: -argName value -argName value
    for (int i = 1; i < argc; i+=2) {
        if (argv[i][0] != '-') // super advanced pattern matching! :)
            throw std::runtime_error("wrong command name");

        mParsedArgs[argv[i]+1] = TryParseString(argv[i+1]);
    }
}
```

At the moment of writing, `std::from_chars` in GCC/Clang only supports integers. MSVC starting from the version 2017 15.8 has full support also for floating-point numbers. You can read more about `from_chars` in the separate String Conversions Chapter. If you want to play with the code in GCC/Clang, you can use the following file `variant_parsing_int_float_gcc.cpp` - it works only with integers and strings.

The idea of `TryParseString` is to try parsing the input string into the best matching type. So if it looks like an integer, then we try to fetch an integer. Otherwise, we'll return an unparsed string. Of course, we can extend this approach.

After the parsing is complete the client can use `Find()` method to test for existence of a parameter:

```
std::optional<CmdLine::Arg> CmdLine::Find(const std::string& name) const {
    if (const auto it = mParsedArgs.find(name); it != mParsedArgs.end())
        return it->second;

    return { };
}
```

`Find()` uses `std::optional` to return the value. If the argument cannot be found in the map, then the client will get empty optional.

Example of how we can use it:

Chapter Variant/variant_parsing_int_float.cpp

```
try {
    CmdLine cmdLine(argc, argv);

    if (auto arg = cmdLine.Find("paramInt"); arg)
        std::cout << "paramInt is " << std::get<int>(*arg) << '\n';

    if (auto arg = cmdLine.Find("paramFloat"); arg) {
        if (const auto intPtr = std::get_if<int>(&*arg); intPtr)
            std::cout << "paramFloat is " << *intPtr << " (integer)\n";
        else
            std::cout << "paramFloat is " << std::get<float>(*arg) << '\n';
    }

    if (auto arg = cmdLine.Find("paramText"); arg)
        std::cout << "paramText is " << std::get<std::string>(*arg) << '\n';
}
catch (const std::bad_variant_access& err) {
    std::cerr << " ...err: accessing a wrong variant type, "
              << err.what() << '\n';
}
catch (const std::runtime_error &err) {
    std::cerr << " ...err: " << err.what() << '\n';
}
```

The above example uses `cmdLine.Find()` to check if there's a given parameter. It returns `std::optional` so we have to check if it's not empty. When we're sure the parameter is available, we can check its type.

`CmdLine` tries to find the best matching type, so for example, with floats, we have ambiguity - as `90` is also `float` but the code will store it as `int` (as it doesn't have the fraction part).

To solve such ambiguities, we could pass some additional information about the desired type, or provide some helper methods.

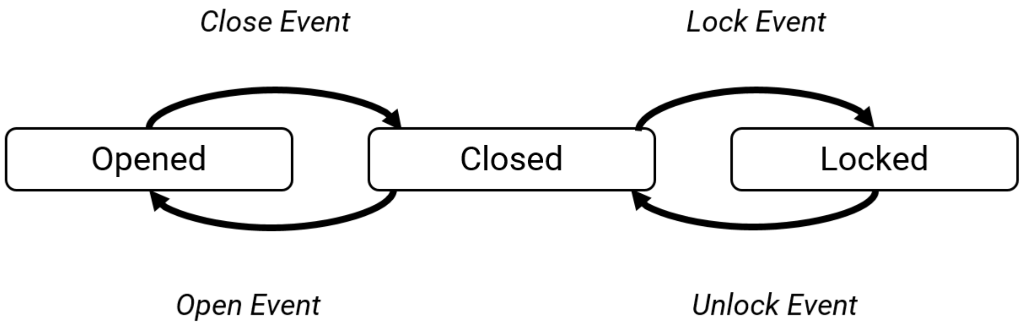
Parsing a Config File

The idea comes from the previous example of a command line. In the case of a configuration file, we usually work with pairs of `<Name, Value>`. Where `Value` might be a different type: `string`, `int`, array, `bool`, `float`, etc.

For such a use case, even `void*` could be used to hold such an unknown type. However, this pattern is extremely error-prone. We could improve the design by using `std::variant` if we know all the possible types, or leverage `std::any`.

State Machines

How about modelling a state machine? For example, door's state:



Door State Machine

We can use different types of states and the use visitors as events:

Chapter Variant/variant_fsm.cpp

```
struct DoorState {
    struct DoorOpened {};
    struct DoorClosed {};
    struct DoorLocked {};

    using State = std::variant<DoorOpened, DoorClosed, DoorLocked>;

    void open() {
        m_state = std::visit(OpenEvent{}, m_state);
    }

    void close() {
        m_state = std::visit(CloseEvent{}, m_state);
    }

    void lock() {
        m_state = std::visit(LockEvent{}, m_state);
    }

    void unlock() {
        m_state = std::visit(UnlockEvent{}, m_state);
    }

    State m_state;
};
```

And here are the events:

Chapter Variant/variant_fsm.cpp

```
struct OpenEvent {
    State operator()(const DoorOpened&){ return DoorOpened(); }
    State operator()(const DoorClosed&){ return DoorOpened(); }
    // cannot open locked doors
    State operator()(const DoorLocked&){ return DoorLocked(); }
};

struct CloseEvent {
    State operator()(const DoorOpened&){ return DoorClosed(); }
    State operator()(const DoorClosed&){ return DoorClosed(); }
    State operator()(const DoorLocked&){ return DoorLocked(); }
};
```

```
struct LockEvent {
    // cannot lock opened doors
    State operator()(const DoorOpened&){ return DoorOpened(); }
    State operator()(const DoorClosed&){ return DoorLocked(); }
    State operator()(const DoorLocked&){ return DoorLocked(); }
};

struct UnlockEvent {
    // cannot unlock opened doors
    State operator()(const DoorOpened&){ return DoorOpened(); }
    State operator()(const DoorClosed&){ return DoorClosed(); }
    // unlock
    State operator()(const DoorLocked&){ return DoorClosed(); }
};
```

We can now create `Door` object and switch between states:

```
DoorState state;
assert(std::holds_alternative<DoorState::DoorOpened>(state.m_state));
state.lock();
assert(std::holds_alternative<DoorState::DoorOpened>(state.m_state));
```

You can read more about state machines and implementation of a simple space game in the following blog post: A std::variant-Based State Machine by Example[10].

Polymorphism

Most of the time in C++, we can safely use runtime polymorphism based on a `vtable` approach. You have a collection of related types that share the same interface, and you have a well defined virtual method that can be invoked.

But what if you have "unrelated" types that don't share the same base class? What if you'd like to quickly add new functionality without changing the code of the supported types?

With `std::variant` and `std::visit` we can build the following example:

[10] https://www.bfilipek.com/2019/06/fsm-variant-game.html

Chapter Variant/variant_polymorphism.cpp

```cpp
class Triangle {
public:
    void Render()  { std::cout << "Drawing a triangle!\n"; }
};

class Polygon {
public:
    void Render() { std::cout << "Drawing a polygon!\n"; }
};

class Sphere {
public:
    void Render() { std::cout << "Drawing a sphere!\n"; }
};

int main() {
    std::vector<std::variant<Triangle, Polygon, Sphere>> objects {
        Polygon(),
        Triangle(),
        Sphere(),
        Triangle()
    };

    auto CallRender = [](auto& obj) { obj.Render(); };

    for (auto& obj : objects)
        std::visit(CallRender, obj);
}
```

The above example shows only the first case of invoking a method from unrelated types. It wraps all the possible shape types into a single variant and then uses a visitor to dispatch the call to the proper type.

If you'd like, for example, to sort objects, then you can write another visitor, one that holds some state. And that way you'll get more functionality without changing the types.

Wrap Up

Here are the things to remember about `std::variant`:

- It holds one of several alternatives in a type-safe way
- No extra memory allocation is needed. The variant needs the size of the max of the sizes of the alternatives, plus some little extra space for knowing the currently active value
- By default, it initialises with the default value of the first alternative
- You can access the value through `std::get`, `std::get_if` or through a form of a visitor
- To check the currently active type you can use `std::holds_alternative` or `std::variant::index`
- `std::visit` provides a way to perform an operation that is implemented for any possible type that might currently be the active one in the variant. Such a polymorphic operation is represented by a callable object that implements its call-operator for every possible type that this variant can hold
- Rarely `std::variant` might get into an invalid state, you can check this issue with the `valueless_by_exception()` method

Compiler Support

Feature	**GCC**	**Clang**	**MSVC**
`std::variant`	7.1	4.0	VS 2017 15.0

9. `std::any`

With `std::optional` you can represent a regular Type values or mark it as empty. With `std::variant` you can wrap several type alternatives into one entity. But C++17 gives us one more wrapper type: `std::any` which can hold anything in a type-safe way.

In this chapter, you'll learn:

- Why `void*` is a very unsafe pattern
- `std::any` and its primary usage
- `std::any` use cases with examples
- `any_cast` and how to use all its "modes"

The Basics

In C++14 there weren't many options for holding variable types in a variable. You could use `void*`, of course, but this wasn't safe. `void*` is just a raw pointer, and you have to manage the whole object lifetime and protect it from casting to a different type.

Potentially, `void*` could be wrapped in a class with some type discriminator.

```
class MyAny {
    void* _value;
    TypeInfo _typeInfo;
};
```

As you see, we have some basic form of the type, but there's a bit of coding required to make sure `MyAny` is type-safe. That's why it's best to use the Standard Library rather than rolling a custom implementation.

And this is what `std::any` from C++17 is in its basic form. It lets you store anything in an object, and it reports errors (or throw exceptions) when you'd like to access a type that is not active.

A little demo:

Chapter Any/any_demo.cpp

```
std::any a(12);

// set any value:
a = std::string("Hello!");
a = 16;
// reading a value:

// we can read it as int
std::cout << std::any_cast<int>(a) << '\n';

// but not as string:
try {
    std::cout << std::any_cast<std::string>(a) << '\n';
}
catch(const std::bad_any_cast& e) {
    std::cout << e.what() << '\n';
}
```

```
// reset and check if it contains any value:
a.reset();
if (!a.has_value()) {
    std::cout << "a is empty!" << '\n';
}

// you can use it in a container:
std::map<std::string, std::any> m;
m["integer"] = 10;
m["string"] = std::string("Hello World");
m["float"] = 1.0f;

for (auto &[key, val] : m) {
    if (val.type() == typeid(int))
        std::cout << "int: " << std::any_cast<int>(val) << '\n';
    else if (val.type() == typeid(std::string))
        std::cout << "string: " << std::any_cast<std::string>(val) << '\n';
    else if (val.type() == typeid(float))
        std::cout << "float: " << std::any_cast<float>(val) << '\n';
}
```

The code will output:

```
16
bad any_cast
a is empty!
float: 1
int: 10
string: Hello World
```

The example above shows us several things:

- `std::any` is not a template class like `std::optional` or `std::variant`.
- by default it contains no value, and you can check it via `.has_value()`.
- you can reset an any object via `.reset()`.
- it works on "decayed" types - so before assignment, initialisation, or emplacement the type is transformed by std::decay[1].
- when a different type is assigned, then the active type is destroyed.

[1]http://en.cppreference.com/w/cpp/types/decay

- you can access the value by using `std::any_cast<T>`. It will throw `bad_any_cast` if the active type is not `T`.
- you can discover the active type by using `.type()` that returns `std::type_info`[2] of the type.

When to Use

While `void*` might be an extremely unsafe pattern with some limited use cases, `std::any` adds type-safety, and that's why it has more applications.

Some possibilities:

- In Libraries - when a library type has to hold or pass anything without knowing the set of available types
- Parsing files - if you really cannot specify what the supported types are
- Message passing
- Bindings with a scripting language
- Implementing an interpreter for a scripting language
- User Interface - controls might hold anything
- Entities in an editor

In many cases, you can limit the number of supported types, and that's why `std::variant` might be a better choice. Of course, it gets tricky when you implement a library without knowing the final applications - so you don't know the possible types that will be stored in an object.

`std::any` Creation

There are several ways you can create `std::any` object:

- a default initialisation - then the object is empty
- a direct initialisation with a value/object
- in place `std::in_place_type`
- via `std::make_any`

[2] http://en.cppreference.com/w/cpp/types/type_info

You can see it in the following example:

Chapter Any/any_creation.cpp

```
// default initialisation:
std::any a;
assert(!a.has_value());

// initialisation with an object:
std::any a2{10}; // int
std::any a3{MyType{10, 11}};

// in_place:
std::any a4{std::in_place_type<MyType>, 10, 11};
std::any a5{std::in_place_type<std::string>, "Hello World"};

// make_any
std::any a6 = std::make_any<std::string>{"Hello World"};
```

In Place Construction

Following the style of `std::optional` and `std::variant`, `std::any` can use `std::in_-place_type` to efficiently create objects in place.

Complex Types

In the below example a temporary object will be needed:

```
std::any a{UserName{"hello"}};
```

but with:

```
std::any a{std::in_place_type<UserName>, "hello"};
```

The object is created in place with the given set of arguments.

For convenience `std::any` has a factory function called `std::make_any` that returns:

```
return std::any(std::in_place_type<T>, std::forward<Args>(args)...);
```

In the previous example we could also write:

```
auto a = std::make_any<UserName>{"hello"};
```

`make_any` is probably more straightforward to use.

Changing the Value

When you want to change the currently stored value in `std::any` then you have two options: use `emplace` or the assignment:

Chapter Any/any_changing.cpp

```
std::any a;

a = MyType(10, 11);
a = std::string("Hello");

a.emplace<float>(100.5f);
a.emplace<std::vector<int>>({10, 11, 12, 13});
a.emplace<MyType>(10, 11);
```

Object Lifetime

The crucial part of being safe for `std::any` is not to leak any resources. To achieve this behaviour `std::any` will destroy any active object before assigning a new value.

Chapter Any/any_lifetime.cpp

```
std::any var = std::make_any<MyType>();
var = 100.0f;
std::cout << std::any_cast<float>(var) << '\n';
```

If the constructors and destructors were instrumented with prints, we would get the following output:

```
MyType::MyType
MyType::~MyType
100
```

The any object is initialised with MyType, but before it gets a new value (of 100.0f) it calls the destructor of MyType.

Accessing The Stored Value

To access the currently active value in std::any you have one option:

std::any_cast<T>().

The function has three "modes" you can work with:

- read access - takes std::any as a reference, returns a copy of the value, throws std::bad_any_cast when it fails
- read/write access - takes std::any as a reference, returns a reference, throws std::bad_any_cast when it fails
- read/write access - takes std::any as a pointer, returns a pointer to the value (const or not) or nullptr

In short:

```
std::any var = 10;

// read access:
auto a = std::any_cast<int>(var);

// read/write access through a reference:
std::any_cast<int&>(var) = 11;

// read/write through a pointer:
int* ptr = std::any_cast<int>(&var);
*ptr = 12;
```

See the example:

Chapter Any/any_access.cpp

```
struct MyType {
    int a, b;

    MyType(int x, int y) : a(x), b(y) { }

    void Print() { std::cout << a << ", " << b << '\n'; }
};

int main() {
    std::any var = std::make_any<MyType>(10, 10);
    try {
        std::any_cast<MyType&>(var).Print();
        std::any_cast<MyType&>(var).a = 11; // read/write
        std::any_cast<MyType&>(var).Print();
        std::any_cast<int>(var); // throw!
    }
    catch(const std::bad_any_cast& e) {
        std::cout << e.what() << '\n';
    }

    int* p = std::any_cast<int>(&var);
    std::cout << (p ? "contains an int... \n" : "doesn't contain an int...\n");

    if (MyType* pt = std::any_cast<MyType>(&var); pt) {
        pt->a = 12;
        std::any_cast<MyType&>(var).Print();
    }
}
```

There are two options regarding error handling: via exceptions (`std::bad_any_cast`) or by returning a pointer (or `nullptr`). The function overloads for `std::any_cast` with pointer access are also marked with `noexcept`.

Performance & Memory Considerations

`std::any` looks quite powerful, and you might use it to hold variables of variable types... but you might ask what is the price for such flexibility.

The main issue here is the cost of extra dynamic memory allocations.

`std::variant` and `std::optional` don't require any extra memory allocations but this is because they know which type (or types) will be stored in the object. `std::any` doesn't know which types might be stored, and that's why it might use some heap memory.

Will it always happen, or sometimes? What are the rules? Will it happen even for a simple type like `int`?

Let's see what the standard says 23.8.3 [any.class][3]:

> Implementations should avoid the use of dynamically allocated memory for a small contained value. Example: where the object constructed is holding only an int. Such small-object optimisation shall only be applied to types `T` for which `is_nothrow_move_constructible_v<T>` is `true`.

To sum up, implementations are encouraged to use SBO (**Small Buffer Optimisation**). But that also comes at some cost: it will make the type larger to fit the buffer.

Let's check the size of `std::any`:

Here are the results from the three compilers:

Compiler	`sizeof(any)`
GCC 8.1 (Coliru)	16
Clang 7.0.0 (Wandbox)	32
MSVC 2017 15.7.0 32-bit	40
MSVC 2017 15.7.0 64-bit	64

In general, as you see, `std::any` is not a "simple" type and it brings a lot of overhead with it. It's usually not small - due to SBO - it takes 16 or 32 bytes (GCC, Clang, or even 64 bytes in MSVC).

You can see the code in `Chapter Any/any_sizeof.cpp`.

Migration from `boost::any`

Boost Any was introduced around the year 2001 (Version 1.23.0). Interestingly, the author of the boost library - Kevlin Henney - is also the author of the proposal for `std::any`. So the two types are strongly connected, and the STL version is heavily based on the predecessor.

[3] https://timsong-cpp.github.io/cppwp/n4659/any#class-3

Here are the main changes:

Feature	Boost.Any (1.67.0)[4]	**std::any**
Extra memory allocation	Yes	Yes
Small buffer optimisation	Yes	Yes
emplace	No	Yes
in_place_type_t in constructor	No	Yes

There are not many differences between the two types. Most of the time you can easily convert from `boost.any` into the STL version.

Examples of `std::any`

The core of `std::any` is flexibility. In the examples below, you can see some ideas where supporting variable types can make an application a bit simpler.

Parsing files

In the examples for `std::variant`, you can see how it's possible to parse configuration files and store the result as an alternative of several types. If you write an entirely generic solution - for example as a part of some library, then you might not know all the possible types.

Storing `std::any` as a value for a property might be good enough from the performance point of view and will give you flexibility.

Message Passing

In Windows API, which is C mostly, there's a message-passing system that uses message IDs with two optional parameters that store the value of the message. Based on that mechanism, you can implement `WndProc` to handle the messages passed to your window/control:

```
LRESULT CALLBACK WindowProc(
  _In_ HWND   hwnd,
  _In_ UINT   uMsg,
  _In_ WPARAM wParam,
  _In_ LPARAM lParam
);
```

The trick here is that the values are stored in `wParam` or `lParam` in various forms. Sometimes you have to use only a few bytes of `wParam`…

What if we changed this system into `std::any`, so that a message could pass anything to the handling method?

For example:

Chapter Any/any_winapi.cpp

```
class Message {
public:
    enum class Type {
        Init,
        Closing,
        ShowWindow,
        DrawWindow
    };

public:
    explicit Message(Type type, std::any param) :
        mType(type),
        mParam(param)
    {   }
    explicit Message(Type type) :
        mType(type)
    {   }

    Type mType;
    std::any mParam;
};

class Window {
public:
    virtual void HandleMessage(const Message& msg) = 0;
};
```

For example you can send a message to a window:

```
Message m(Message::Type::ShowWindow, std::make_pair(10, 11));
yourWindow.HandleMessage(m);
```

Then the window can respond to the message with the following message handler:

```
switch (msg.mType) {
// ...
case Message::Type::ShowWindow: {
    auto pos = std::any_cast<std::pair<int, int>>(msg.mParam);
    std::cout << "ShowWindow: "
              << pos.first << ", "
              << pos.second << '\n';
    break;
    }
}
```

Of course, you have to define how the values are specified (what the types of a value of a message are), but now you can use real types rather than doing various tricks with integers.

Properties

The original paper that introduces any to C++, N1939[5] shows an example of a property class.

```
struct property {
    property();
    property(const std::string &, const std::any &);

    std::string name;
    std::any value;
};

typedef std::vector<property> properties;
```

The `properties` object looks quite powerful as it can hold many different types. One of the examples where such a structure might be leveraged is a game editor.

[5]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1939.html

Wrap Up

Here are the things to remember about `std::any`:

- `std::any` is not a template class
- `std::any` uses Small Buffer Optimisation, so it will not dynamically allocate memory for simple types like ints, doubles… but for larger types, it will use extra `new`.
- `std::any` might be considered 'heavy', but offers a lot of flexibility and type-safety.
- you can access the currently stored value by using `any_cast` that offers a few "modes": for example it might throw an exception or return `nullptr`.
- use it when you don't know the possible types - in other cases consider `std::variant`.

Compiler Support

Feature	GCC	Clang	MSVC
`std::any`	7.1	4.0	VS 2017 15.0

10. `std::string_view`

Since C++11 and move semantics, passing strings has become much faster. Yet you can end up with many temporary string copies. In C++17 you get a new type called `string_view`. It allows you to create a constant, non-owning *view* of a contiguous character sequence. You can manipulate that view and pass it around without the need to copy the referenced data. Nevertheless, the feature comes at some cost: you need to be careful not to end up with "dangling" views, and usually such views might not be null-terminated.

In this chapter, you'll learn:

- What is `string_view`?
- Why might it speed up your code?
- What are the risks involved with using `string_view` objects?
- What is the reference lifetime extension, and what does it mean for `string_view`?
- How you can use `string_view` to make your API more generic?

The Basics

Let's try a little experiment:

How many string copies are created in the below example?

```
// string function:
std::string StartFromWordStr(const std::string& strArg, const std::string& word) {
    return strArg.substr(strArg.find(word)); // substr creates a new string
}

// call:
std::string str {"Hello Amazing Programming Environment" };

auto subStr = StartFromWordStr(str, "Programming Environment");
std::cout << subStr << '\n';
```

Can you count them all?

The answer is 3 or 5 depending on the compiler, but usually, it should be 3.

- The first one is for `str`.
- The second one is for the second argument in `StartFromWordStr` - the argument is `const string&` so since we pass `const char*` it will create a new string.
- The third one comes from `substr` which returns a new `string`.
- Then we might also have another copy or two - as the object is returned from the function. But usually, the compiler can optimise and elide the copies (especially since C++17 when Copy Elision became mandatory in that case).
- If the string is short, then there might be no heap allocation as Small String Optimisation[1].

The above example is simplistic. However, you might imagine a production code where string manipulations happen very often. In that scenario, it's even hard to count all the temporaries that the compiler creates.

A much better pattern to solve the problem with extra temporary copies is to use `std::string_view`. As the name suggests, instead of using the original string, you'll only get a non-owning

[1]Small String Optimisation is not defined in the C++ Standard, but it's a common optimisation across popular compilers. Currently, it's 15 characters in MSVC (VS 2017)/GCC (8.1) or 22 characters in Clang (6.0).

view of it. Most of the time, it will be a pointer to the contiguous character sequence and the length. You can pass it around and use most of the conventional string operations.

Views work well with string operations like substring - `substr`. In a typical case, each substring operation creates another, smaller copy of the string. With `string_view`, `substr` will only map a different portion of the original buffer, without additional memory usage, or dynamic allocation.

Here's the updated version of our code that accepts `string_view`:

Chapter string_view/avoiding_copies_string_view.cpp

```
std::string_view StartFromWord(std::string_view str, std::string_view word) {
    return str.substr(str.find(word)); // substr creates only a new view
}

// call:
std::string str {"Hello Amazing Programming Environment"};

auto subView = StartFromWord(str, "Programming Environment");
std::cout << subView << '\n';
```

In the above case, we have only one allocation - just for the main string - `str`. None of the `string_view` operations invokes copy or extra memory allocation for a new string. Of course, `string_view` is copied - but since it's only a pointer and a length, it's much more efficient than the copy of the whole string.

One warning: while this example shows the optimisation capability of string views, please read on to see the risks and assumptions with that code! Or maybe you can spot a few now?

Ok, so when you should use `string_view`:

When to Use

- Optimisation: you can carefully review your code and replace various string operations with `string_view`. In most cases, you should end up with faster code and fewer memory allocations.
- As a possible replacement for `const std::string&` parameter - especially in functions that don't need the ownership and don't store the string.
- Handling strings coming from other API: `QString`, `CString`, `const char*...` everything that is placed in a contiguous memory chunk and has a basic char-type.

> You can write a function that accepts `string_view` and no conversion from that other implementation will happen.

In any case, it's important to remember that it's only a **non-owning view**, so if the original object is gone, the view becomes rubbish and you might get into trouble.

Moreover, **`string_view` might not contain null terminator** so your code has to support that as well. For example, it's never a good idea to pass `string_view` to a function that accepts null-terminated strings. More on that in a separate section - about "Risks with `string_view`".

The `std::basic_string_view` Type

Although we talk about `string_view` it's important to know that this is only a specialisation of a template class called `basic_string_view`:

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>
> class basic_string_view;
```

`Traits` class is used to abstract the operations on the character type, for example, on how to compare characters, how to find one character in a sequence.

Such a hierarchy creates a similar relation as there is for `std::string` which is a specialisation of `std::basic_string`.

We have the following specialisations:

```
std::string_view        std::basic_string_view<char>
std::wstring_view       std::basic_string_view<wchar_t>
std::u16string_view     std::basic_string_view<char16_t>
std::u32string_view     std::basic_string_view<char32_t>
```

As you can see, the specialisations use a different underlying character type.

For the sake of convenience, in the rest of this chapter, only `string_view` will be considered.

std::string_view Creation

You can create a `string_view` in several ways:

- from `const char*` - providing a pointer to a null-terminated string
- from `const char*` with length
- by using a conversion from `std::string`
- by using `""sv` literal

Here's an example of various creation options:

Chapter string_view/string_view_creation.cpp

```
const char* cstr = "Hello World";

// the whole string:
std::string_view sv1 { cstr };
std::cout << sv1 << ", len: " << sv1.size() << '\n';

// slice
std::string_view sv2 { cstr, 5 }; // not null-terminated!
std::cout << sv2 << ", len: " << sv2.size() << '\n';

// from string:
std::string str = "Hello String";
std::string_view sv3 = str;
std::cout << sv3 << ", len: " << sv3.size() << '\n';

// ""sv literal
using namespace std::literals;
std::string_view sv4 = "Hello\0 Super World"sv;
std::cout << sv4 << ", len: " << sv4.size() << '\n';
std::cout << sv4.data() << " - till zero\n";
```

The code will print:

```
Hello World, len: 11
Hello, len: 5
Hello String, len: 12
Hello Super World, len: 18
Hello - till zero
```

Please notice the last two lines: `sv4` contains `'\0'` in the middle, but `std::cout` can still print the whole sequence. In the last line, we try to print with `.data()` and we end up with a string pointer so the printing breaks at the null terminator.

Other Operations

`string_view` is modelled to be very similar to `std::string`. The view, however, is non-owning, so any operation that modifies the data cannot go into the API. Here's a brief list of methods that you can use with this new type:

Iterators:

Method	Description
`cbegin()`, `begin()`	Return an iterator to the first character
`crbegin()`, `rbegin()`	Return a reverse iterator to the first character of the reversed view. It corresponds to the last character of the sequence.
`cend()`, `end()`	Returns an iterator to a place after the last character of a sequence
`crend()`, `rend()`	Returns an iterator to the end of reversed sequence. It corresponds to a place before the first character

Please note that all of the above methods are `constexpr` and `const`, so you always get a const iterator (even for `begin()` or `end()`).

Accessing Elements:

Method	Description
`operator[]`	Returns a const reference to the character at the specified position. Bounds are not checked.
`at()`	Returns a const reference to the character at specified position with bound checking (might throw `std::out_of_range`)
`front()`	Returns a const reference to the first character of the sequence
`back()`	Returns a const reference to the last character of the sequence
`data()`	Returns a pointer to the underlying data

If the view is empty then you'll get undefined behaviour for `operator[]`, `front()`, `back()` and `data()`.

Size & Capacity:

Method	Description
`size()/length()`	Returns the numbers of characters in a sequence
`max_size()`	The largest possible number of char-like objects that can be referred to by a `basic_string_view`.
`empty()`	Returns `size == 0`

Modifiers:

Method	Description
`remove_prefix(size_type n)`	Equivalent to: `data_ += n`; `size_ -= n;`
`remove_suffix(size_type n)`	Equivalent to: `size_ -= n;`
`swap(basic_string_view& s)`	Exchanges the values of `*this` and `s`

Other methods:

Method	Description
`copy(charT* s, size_type n, size_type pos)`	Copies `n` characters into `s` starting from `pos`. not `constexpr`
`substr(size_type pos, size_type n)`	Complexity `O(1)` and not `O(n)` as in `std::string`
`compare(...)` [2]	Compares strings, similarly to `std::basic_string::compare`
`find(...)`	Returns position of the first occurence of the input string or `basic_string_view::npos`
`rfind(...)`	Returns position of the last occurence of the input string or `basic_string_view::npos`
`find_first_of(...)`	Returns position of the first character that is equal to any character from the input pattern or `basic_string_view::npos`
`find_last_of(...)`	Returns position of the last character that is equal to any character from the input pattern or `basic_string_view::npos`
`find_first_not_of(...)`	Returns position of the first character that is different to any character from the input pattern or `basic_string_view::npos`
`find_last_not_of(...)`	Returns position of the last character that is different to any character from the input pattern or `basic_string_view::npos`

[2] ellipsis `(...)` means that a method has several overloads.

Non-member functions:

Function	Description
comparison operators: ==, !=, <=, >=, <, >	Lexicographically compares two string views
`operator <<`	For `ostream` output

Key things about the above methods, functions and types:

- All of the above methods (except for `copy`, `operator <<` and `std::hash` specialisation) are also `constexpr`! With this capability, you might now work with contiguous character sequences in constant expressions.
- The above list is almost the same as all non-mutable string operations. However, there are two new methods: `remove_prefix` and `remove_suffix` - they are not `const`, and they modify the `string_view` object. Note that they still cannot modify the referenced data.
- `operator[]`, `at`, `front`, `back`, `data` - are also `const` - thus you cannot change the underlying character sequence (it's only "read access"). In `std::string` there are overloads for those methods that return a reference, so you get "write access". That's not possible with `string_view`.
- `string_view` also has specialisation for `std::hash`
- `string_view` has a string literal `""sv`, and you can define a variable like `auto sv = "hello"sv;`

More in C++20!

In C++20 we'll get two new methods:

- `starts_with()`
- `ends_with()`

They are implemented both for `std::basic_string_view` and `std::basic_string`. As of August 2019 Clang 6.0, GCC 9.0 and VS 2019 16.2 support them.

Risks Using `string_view`

`std::string_view` was added into the Standard mostly to allow performance optimisations. Nevertheless, it's not a replacement for strings! That's why when you use views you have to remember about a few potentially risky things:

Taking Care of Not Null-Terminated Strings

`string_view` may not contain `\0` at the end of the string. So you have to be prepared for that.

- `string_view` is problematic with *all* functions that accept traditional C-strings because `string_view` breaks with C-string termination assumptions. If a function accepts only a `const char*` parameter, it's probably a bad idea to pass `string_view` into it. On the other hand, it might be safe when such a function accepts `const char*` and `length` parameters.
- Conversion into strings - you need to specify not only the pointer to the contiguous character sequence but also the length.

References and Temporary Objects

`string_view` doesn't own the memory, so you have to be very careful when working with temporary objects.

In general, the lifetime of a `string_view` must never exceed the lifetime of the string-owning object.

That might be crucial when:

- Returning `string_view` from a function - the view has to point to that data that is still alive after the function has completed.
- Storing `string_view` in objects or containers - this is similar to storing pointers in a container. The referenced data must be still present when you access elements of this container.

To explore all the issues, let's start with the initial example from this chapter.

Problems with the Initial Example

The intro section showed you an example of:

```
std::string_view StartFromWord(std::string_view str, std::string_view word) {
    return str.substr(str.find(word)); // substr creates only a new view
}
```

The code doesn't have any issues with non-null-terminated strings - as all the functions are from the `string_view` API.

However, how about temporary objects?

What will happen if you call:

```
auto str = "My Super"s;
auto sv = StartFromWord(str + " String", "Super");
// use `sv` later in the code...
```

Code like that might blow!

`"Super"` is a temporary `const char*` literal and it's passed as `string_view word` into the function. That's fine, as the temporary is guaranteed to live as long as the whole function invocation.

However, the result of string concatenation `str + " String"` is a temporary and the function returns a `string_view` of this temporary outside the call!

So the general advice in such cases is that while it's possible to return a `string_view` from a function, you have to be careful and be sure about the state of the underlying string.

To understand issues with temporary values, it's good to have a look at the reference lifetime extension.

Reference Lifetime Extension

What happens in the following case:

```
std::vector<int> GenerateVec() {
    return std::vector<int>(5, 1);
}
const std::vector<int>& refv = GenerateVec();
```

Is the above code safe?

Yes - the C++ rules say that the lifetime of a temporary object bound to a `const` reference is prolonged to the lifetime of the reference itself.

Here's a full example quoted from the standard (Draft C++17 - N4687[3]) 15.2 Temporary objects [class.temporary]:

> [Example:
>
> ```
> struct S {
>  S();
>  S(int);
>  friend S operator+(const S&, const S&);
>  ~S();
> };
> S obj1;
> const S& cr = S(16)+S(23);
> S obj2;
> ```
>
> The expression `S(16) + S(23)` creates three temporaries: a first temporary `T1` to hold the result of the expression `S(16)`, a second temporary `T2` to hold the result of the expression `S(23)`, and a third temporary `T3` to hold the result of the addition of these two expressions. The temporary `T3` is then bound to the reference `cr`. It is unspecified whether `T1` or `T2` is created first. On an implementation where `T1` is created before `T2`, `T2` shall be destroyed before `T1`. The temporaries `T1` and `T2` are bound to the reference parameters of `operator+`; these temporaries are destroyed at the end of the full-expression containing the call to `operator+`. The temporary `T3` bound to the reference `cr` is destroyed at the end of `cr`'s lifetime, that is, at the end of the program. In addition, the order in which `T3` is destroyed takes into account the destruction order of other objects with static storage duration. That is, because `obj1` is constructed before `T3`, and `T3` is constructed before `obj2`, `obj2` shall be destroyed before `T3`, and `T3` shall be destroyed before `obj1`. `-end example]`

[3] https://wg21.link/n4687

While it's better not to write such code for all of your variables, it might be a handy feature in cases like:

```
for (auto &elem : GenerateVec()) {
    // ...
}
```

In the above example, `GenerateVec` is bound to a reference (rvalue reference for the start of the vector) inside the range-based for loop. Without the extended lifetime support, the code would break.

How does it relate to `string_view`?

For `string_view` the below code is usually error-prone:

```
std::string func() {
    std::string s;
    // build s...
    return s;
}

std::string_view sv = func();
// no temp lifetime extension!
```

This might be not obvious - `string_view` is also a constant view, so should behave almost like a `const` reference. But according to existing C++ rules, it's not- the compiler immediately destroys the temporary object after the whole expression is done. The lifetime cannot be extended in this case.

`string_view` is just a proxy object, similar to another code:

```
std::vector<int> CreateVector() { ... }
std::string GetString() { return "Hello"; }

auto &x = CreateVector()[10]; // arbitrary element!
auto pStr = GetString().c_str();
```

In both cases `x` and `pStr` won't extend the lifetime of the temporary object created in `CreateVector()` or `GetString()`.

You might fix it by:

```
std::string func() {
    std::string s;
    // build s...
    return s;
}
auto temp = func();
std::string_view sv { temp };
// fine lifetime of temporary is extended through `temp`
```

Every time you assign a return value from some function, you have to be sure the lifetime of the object is correct.

There's a proposal to fix the issues with string_views and other types that should have extended reference lifetime semantics: see P0936[4].

Initializing `string` Members from `string_view`

Since `string_view` is a potential replacement for `const string&` when passing in functions, we might consider a case of `string` member initialisation. Is `string_view` the best candidate here? See the following example:

```
class UserName {
    std::string mName;
public:
    UserName(const std::string& str) : mName(str) { }
};
```

As you can see a constructor is taking `const std::string& str`.The other option is to use `string_view`:

```
UserName(std::string_view sv) : mName(sv) { }
```

Let's compare those alternatives implementations in three cases: creating from a string literal, creating from an `lvalue` and creating from an `rvalue` reference:

[4]https://wg21.link/p0936

```
// creation from a string literal
UserName u1{"John With Very Long Name"};

// creation from lvalue:
std::string s2 {"Marc With Very Long Name"};
UserName u2 { s2 };
// use s2 later...

// from rvalue reference
std::string s3 {"Marc With Very Long Name"};
UserName u3 { std::move(s3) };

// third case is also similar to taking a return value:
std::string GetString() { return "some string..."; }
UserName u4 { GetString() };
```

Now we can analyse two versions of `UserName` constructor - with a `string` reference or a `string_view`.

Please note that allocations/creation of `s2` and `s3` are not taken into account, we're only looking at what happens for the constructor call. For `s2` we can also assume it's used later in the code.

For `const std::string&`:

- `u1` - two allocations: the first one creates a temp string and binds it to the input parameter, and then there's a copy into `mName`.
- `u2` - one allocation: we have a no-cost binding to the reference, and then there's a copy into the member variable.
- `u3` - one allocation: we have a no-cost binding to the reference, and then there's a copy into the member variable.
- You'd have to write a `ctor` taking rvalue reference to skip one allocation for the `u1` case, and also that could skip one copy for the `u3` case (since we could move from rvalue reference).

For `std::string_view`:

- `u1` - one allocation - no copy/allocation for the input parameter, there's only one allocation when `mName` is created.

- u2 - one allocation - there's a cheap creation of a string_view for the argument, and then there's a copy into the member variable.
- u3 - one allocation - there's a cheap creation of a string_view for the argument, and then there's a copy into the member variable.
- You'd also have to write a constructor taking rvalue reference if you want to save one allocation in the u3 case, as you could move from rvalue reference.

While the string_view behaves better when you pass a string literal, it's no better when you use it with existing string, or you move from it.

However, since the introduction of move semantics in C++11, it's usually better, and safer to pass string as a value and then move from it.

For example:

```
class UserName {
    std::string mName;

public:
    UserName(std::string str) : mName(std::move(str)) { }
};
```

Now we have the following results:

For std::string:

- u1 - one allocation - for the input argument and then one move into the mName. It's better than with const std::string& where we got two memory allocations in that case. And similar to the string_view approach.
- u2 - one allocation - we have to copy the value into the argument, and then we can move from it.
- u3 - no allocations, only two move operations - that's better than with string_view and const string&!

When you pass std::string by value not only is the code simpler, there's also no need to write separate overloads for rvalue references.

See the full code sample in

Chapter string_view/initializing_from_string_view.cpp

The approach with passing by value is consistent with item 41 - "Consider pass by value for copyable parameters that are cheap to move and always copied" from Effective Modern C++ by Scott Meyers.

However, is `std::string` cheap to move?

Although the C++ Standard doesn't specify that, usually, strings are implemented with **Small String Optimisation** (SSO) - the string object contains extra space to fit characters without additional memory allocation[5]. That means that moving a string is the same as copying it. And since the string is short, the copy is also fast.

Let's reconsider our example of passing by value when the `string` is short:

```
UserName u1{"John"}; // fits in SSO buffer

std::string s2 {"Marc"}; // fits in SSO buffer
UserName u2 { s2 };

std::string s3 {"Marc"}; // fits in SSO buffer
UserName u3 { std::move(s3) };
```

Remember that each move is the same as copy.

For `const std::string&`:

- `u1` - two copies: one copy from the input string literal into a temporary string argument, then another copy into the member variable.
- `u2` - one copy: the existing string is bound to the reference argument, and then we have one copy into the member variable.
- `u3` - one copy: the `rvalue` reference is bound to the input parameter at no cost, later we have a copy into the member field.

For `std::string_view`:

- `u1` - one copy: no copy for the input parameter, there's only one copy when `mName` is initialised.
- `u2` - one copy: no copy for the input parameter, as `string_view` creation is fast, and then one copy into the member variable.

[5]SSO is not standardised and prone to change. Currently, it's 15 characters in MSVC (VS 2017)/GCC (8.1) or 22 characters in Clang (6.0). For multiplatform code, it's not a good idea to assume optimisations based on SSO.

- `u3` - one copy: `string_view` is cheaply created, there's one copy from the argument into `mName`.
- Extra risk that string_view might point to a deleted string.

For `std::string`:

- `u1` - two copies: the input argument is created from a string literal, and then there's copy into `mName`.
- `u2` - two copies: one copy into the argument and then there's the second copy into the member.
- `u3` - two copies: one copy into the argument (move means copy) and then there's the second copy into the member.

As you see for short strings passing by value might be "slower" when you pass some existing string, simply because you have two copies rather than one. On the other hand, the compiler might optimise the code better when it sees an object and not reference. What's more, short strings are cheap to copy, so the potential "slowdown" might not be even visible.

All in all, passing by value and then moving from a string argument is the preferred solution. You have simple code and better performance for larger strings.

As always, if your code needs maximum performance, then you have to measure all possible cases.

Other Types & Automation

The problem discussed in this section can also be extended to other copyable and movable types. If the move operation is cheap, then passing by value might be better than by reference. You can also use automation, like Clang-Tidy, which can detect potential improvements. Clang Tidy has a separate rule for that use case, see clang-tidy - modernize-pass-by-value[6].

[6]https://clang.llvm.org/extra/clang-tidy/checks/modernize-pass-by-value.html

Here's the summary of string passing and initialisation of a string member:

input parameter	`const string&`	`string_view`	`string` and `move`
`const char*`	2 allocations	**1 allocation**	1 allocation + move
`const char*` SSO	2 copies	**1 copy**	2 copies
lvalue	**1 allocation**	**1 allocation**	1 allocation + 1 move
lvalue SSO	**1 copy**	**1 copy**	2 copies
rvalue	1 allocation	1 allocation	**2 moves**
rvalue SSO	**1 copy**	**1 copy**	2 copies

Handling Non-Null Terminated Strings

If you get a `string_view` from a `string` then it will point to a null-terminated chunk of memory:

```
std::string s = "Hello World";
std::cout << s.size() << '\n';
std::string_view sv = s;
std::cout << sv.size() << '\n';
```

The two `cout` statements will both print `11`.

But what if you have just a part of the `string`:

```
std::string s = "Hello World";
std::cout << s.size() << '\n';
std::string_view sv = s;
auto sv2 = sv.substr(0, 5);
std::cout << sv2.data() << '\n'; /// ooops?
```

`sv2` should contain only `"Hello"`, but when you access the pointer to the underlying memory, you'll receive the pointer to the whole string. The expression: `cout << sv2.data()` will print the whole string, and not just a part of it! `sv2.data()` returns the pointer to the `"Hello World"` character array inside the string `s` object.

Of course when you print `sv2` you'll get the correct result:

```
std::cout << sv2 << '\n';
// prints "Hello"
```

This is because `std::cout` handles `string_view` type.

The example shows a potential problem with all third-party APIs that assume null-terminated strings. To name a few:

Printing with `printf()`

For example:

```
std::string s = "Hello World";
std::string_view sv = s;
std::string_view sv2 = sv.substr(0, 5);
printf("My String %s", sv2.data()); // oops?
```

Instead you should use:

```
printf("%.*s\n", static_cast<int>(sv2.size()), sv2.data());
```

`.*` - describes the precision, see in the `printf` specification[7]:

> The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

Conversion Functions Like `atoi()`/`atof()`:

```
std::string number = "123.456";
std::string_view svNum { number.data(), 3 };
auto f = atof(svNum.data()); // should be 123, but is 123.456!
std::cout << f << '\n';
```

`atof` takes only the pointer to a null-terminated string, so `string_view` is not compatible.

To fix this, you might have a look at `from_chars` functions (also added in C++17)

[7]http://www.cplusplus.com/reference/cstdio/printf/

```
// use from_chars (C++17)
std::string number = "123.456";
std::string_view svNum { number.data(), 3 };
int res = 0;
std::from_chars(svNum.data(), svNum.data()+svNum.size(), res);
std::cout << res << '\n';
```

A General Solution

If your API supports only null-terminated strings and you cannot switch to a function that takes additional `count` or `size` parameter, then you need to convert a view into the string.

For example:

Chapter string_view/string_view_null.cpp

```
void ConvertAndShow(const char *str) {
    auto f = atof(str);
    std::cout << f << '\n';
}

std::string number = "123.456";
std::string_view svNum { number.data(), 3 };
// ... some code
std::string tempStr { svNum.data(), svNum.size() };
ConvertAndShow(tempStr.c_str());
```

`ConvertAndShow` only works with null-terminated strings, so the only way we have is to create a temporary string `tempStr` and then pass it to the function.

If you want to create a string object from `string_view` then remember to use `.data()` and `.size()` so that you refer to the correct slice of the underlying character sequence.

Performance & Memory Considerations

The core idea behind adding `string_view` into the Standard Library was performance and memory consumption. By leveraging `string_view`, you can efficiently skip the creation of many temporary strings which might boost performance.

Regarding the memory: string_view is usually implemented as [ptr, len] - one pointer and usually size_t to represent the possible size.

That's why you should see the size of it as 8 bytes or 16 bytes (depending on whether the architecture is x86 or x64).

If we consider the std::string type, due to common Small String Optimisations std::string is usually 24 or 32 bytes, so double the size of string_view. If a string is longer than the SSO buffer then std::string allocates memory on the heap. If SSO is not supported (which is rare), then std::string would consist of a pointer to the allocated memory and the size.

Regarding the performance of string operations.

string_view has only a subset of string operations, those that don't modify the referenced character sequence. Functions like find() should offer the same performance as the string counterparts.

On the other hand, substr is just a copy of two elements in string_view, while string will perform a copy of a memory range. The complexity is O(1) vs O(n). That's why if you need to split a larger string and work with those splices, the string_view implementation should offer better speed.

Strings in Constant Expressions

The interesting property of string_view is that all of the methods are marked as constexpr (except for copy, operator << and std::hash functions specialised for string views). With this capability, you can work on strings at compile time.

For example:

Chapter string_view/string_view_constexpr.cpp

```
#include <string_view>

int main() {
    using namespace std::literals;

    constexpr auto strv = "Hello Programming World"sv;
    constexpr auto strvCut = strv.substr("Hello "sv.size());

    static_assert(strvCut == "Programming World"sv);
    return strvCut.size();
}
```

If you use a modern compiler, like GCC 8.1 with the following options `-std=c++1z -Wall -pedantic -O2`. Then the compiled assembler should be in the following form:

```
main:
        movl    $17, %eax
        ret
```

A similar version of such code, but with `std::string` would generate much more code. Since the example uses long strings, then Small String Optimisation is not possible, and then the compiler must generate code for `new/delete` to manage the memory of the strings.

Migration from `boost::string_ref` and `boost::string_view`

As with most of the new types in C++17 `string_view` is also inspired by `boost` libraries. Marshall Clow implemented `boost::string_ref` in the version 1.53 (February 2012) and then it evolved into `boost::string_view` (added into the version 1.61 - May 2016).

The main difference between `boost::string_ref` and `boost::string_view` is the support for `constexpr`.

`boost::string_view` implements the same functionality as `std::string_view` and also adds a few new functions:

- `starts_with`
- `ends_with`

See the full header file in boost/doc/libs/1_67_0/boost/utility/string_view.hpp[8] And in Boost utility library[9]

The link to the discussion about deprecation of `string_ref`: "string_view versus string_-ref"[10].

[8]https://www.boost.org/doc/libs/1_67_0/boost/utility/string_view.hpp
[9]https://www.boost.org/doc/libs/1_67_0/boost/utility/
[10]https://lists.boost.org/Archives/boost/2017/07/236903.php

Examples

Below you can find two examples of using `string_view`.

Working with Different String APIs

An interesting use case for `string_view` is when you use it in code that works with different string implementations.

For example, you might have `CString` from MFC, `const char*` from C-APIs, `QString` from QT, and of course `std::string`.

Rather than creating overloads for different string types, you might leverage `string_view`!

For example:

```
void Process(std::string_view sv) { }
```

If you want to use `Process` with different string implementations, then all you have to do is to create a string view from your type. Most of the string types should easily allow that.

For example:

```
// MFC Strings:
CString cstr;
Process(std::string_view{cstr.GetString(), cstr.GetLength()});

// QT Strings:
QString qstr;
Process(std::string_view{qstr.toLatin1().constData()};

// Your implementation:
MySuperString myStr;
// MySuperString::GetData() - returns char*
// MySuperString::Length() - returns length of a string
Process(std::string_view{myStr.GetData(), myStr.Length()});
```

Hypothetically, `Process()` could be implemented as `Process(const char*, int len)`, but with `string_view` the code is more explicit and simpler. Additionally, you have all the available methods of `string_view`, and such code is more convenient than C-style.

String Split

`string_view` might be a potential optimisation for string splitting. If you own a large persistent string, you might want to create a list of `string_view` objects that maps words of that larger string.

Please note that the code is inspired by the article by Marco Arena - string_view odi et amo[11].

Chapter string_view/string_view_split.cpp

```
std::vector<std::string_view>
splitSV(std::string_view strv, std::string_view delims = " ") {
    std::vector<std::string_view> output;
    auto first = strv.begin();

    while (first != strv.end()) {
        const auto second = std::find_first_of(
                             first, std::cend(strv),
                            std::cbegin(delims), std::cend(delims));

        if (first != second) {
           output.emplace_back(strv.substr(std::distance(strv.begin(), first),
                                   std::distance(first, second)));
        }

        if (second == strv.end())
            break;

        first = std::next(second);
    }
    return output;
}
```

Example use case:

```
const std::string str { "Hello    Extra,,, Super, Amazing World" };

for (const auto& word : splitSV(str, " ,"))
    std::cout << word << '\n';
```

[11]https://marcoarena.wordpress.com/2017/01/03/string_view-odi-et-amo/

This will print:

```
Hello
Extra
Super
Amazing
World
```

The algorithm iterates over the input `string_view` and finds breaks - characters that match the delimiter list. Then the code extracts part of that sequence - between the last and the new break. The sub-view is stored in the output vector.

Some notes regarding the implementation:

- The `string_view` version of the algorithm assumes the input string is persistent and not a temporary object. Be careful with the returned `vector` of `string_view` as it also points to the input string.
- The instruction `if (first != second)` - protects from adding empty "words", in a case where there are multiple delimiters next to each other (like double spaces).
- The algorithm uses `std::find_first_of` but it's also possible to use `string_view::find_first_of`. The member method doesn't return an iterator, but the position in the string.
- The member method of `string_view` appeared to be slower than the `std::find_first_of` version in some tests when the number of delimiters is small.

If you want to see some experiments regarding the code in this section have a look at: Performance of std::string_view vs std::string from C++17[12] and Speeding Up string_view String Split Implementation[13]. Those two blog posts describe the benchmark results and add some more possible improvements to the code.

[12]https://www.bfilipek.com/2018/07/string-view-perf.html
[13]https://www.bfilipek.com/2018/07/string-view-perf-followup.html

Wrap Up

Here are the things to remember about `std::string_view`:

- It's a specialisation of `std::basic_string_view<charType, traits<charType>>` - with `charType` equal to `char`.
- It's a non-owning view of a contiguous sequence of characters.
- It might not include null terminator at the end.
- It can be used to optimise code and limit the need for temporary copies of strings.
- It contains most of `std::string` operations that don't change the underlying characters.
- Its operations are also marked as `constexpr`.

But:

- Make sure the underlying sequence of characters is still present!
- While `std::string_view` looks like a constant reference to the string, the language doesn't extend the lifetime of returned temporary objects that are bound to `std::string_view`.
- Always remember to use `stringView.size()` when you build a `string` from `string_view`. The `size()` method properly marks the end of `string_view`.
- Be careful when you pass `string_view` into functions that accept null-terminated strings unless you're sure your `string_view` contains a null terminator.

Compiler support:

Feature	GCC	Clang	MSVC
`std::string_view`	7.1	4.0	VS 2017 15.0

11. String Conversions

`string_view` is not the only feature that we get in C++17 that relates to strings. While views can reduce the number of temporary copies, there's also another convenient feature: conversion utilities. In the new C++ Standard, you have two sets of functions `from_chars` and `to_chars` that are low level and promises impressive performance improvements.

In this chapter, you'll learn:

- Why do we need low-level string conversion routines?
- Why the current options in the Standard Library might not be enough?
- How to use C++17's conversion routines
- What performance gains you can expect from the new routines

Elementary String Conversions

The growing number of data formats like JSON or XML require efficient string processing and manipulation. The maximum performance is especially crucial when such data formats are used to communicate over the network, where high throughput is the critical factor.

For example, you get the characters in a network packet, you deserialise it (convert strings into numbers), then process the data, and finally, it's serialised back to the same file format (numbers into strings) and sent over the network as a response.

The Standard Library had bad luck in those areas. It's usually perceived to be too slow for such advanced string processing. Often developers prefer custom solutions or third-party libraries.

The situation might change as with C++17 we get two sets of functions: `from_chars` and `to_chars` that allow for low-level string conversions.

In the original paper (P0067[1]) there's a useful table that summarises all the current solutions:

Facility	**Shortcomings**
`sprintf`	format string, locale, buffer overrun
`snprintf`	format string, locale
`sscanf`	format string, locale
`atol`	locale, does not signal errors
`strtol`	locale, ignores whitespace and 0x prefix
`strstream`	locale, ignores whitespace
`stringstream`	locale, ignores whitespace, memory allocation
`num_put` / `num_get` facets	locale, virtual function
`to_string`	locale, memory allocation
`stoi` etc.	locale, memory allocation, ignores whitespace and 0x prefix, exceptions

As you can see from the table above, sometimes converting functions do too much work, which makes the whole processing slower. Often, there's no need for the extra features.

First of all, all of them use "locale". Even if you work with language-independent strings, you have to pay a small price for localisation support. For example, if you parse numbers from XML or JSON, there's no need to apply current system language, as those formats are interchangeable.

The next issue is error reporting. Some functions might throw an exception while others

[1]https://wg21.link/P0067

return just a converted value. Exceptions might not only be costly (as throwing might involve extra memory allocations) but often a parsing error is not an exceptional situation. Returning a simple value, for example, `0` for `atoi`, `0.0` for `atof` is also not satisfactory, as in that case you don't know if the parsing was successful or not.

The third topic, especially related to C-style API, is that you have to provide some form of the "format string". Parsing such string might involve some additional cost.

Another thing is "empty space" support. Functions like `strtol` or `stringstream` might skip empty spaces at the beginning of the string. That might be handy, but sometimes you don't want to pay for that extra feature.

There's also another critical factor: safety. Simple functions don't offer any buffer overrun solutions, and also they work only on null-terminated strings. In that case, you cannot use `string_view` to pass the data.

The new C++17 API addresses all of the above issues. Rather than providing many functionalities, they focus on giving very low-level support. That way, you can have the maximum speed and tailor them to your needs.

The new functions are guaranteed to be:

- non-throwing - in case of some error they won't throw exceptions (as opposed to `stoi`)
- non-allocating - the entire processing is done in place, without any extra memory allocation
- no locale support - the string is parsed as if used with default ("C") locale
- memory safety - input and output range are specified to allow for buffer overrun checks
- no need to pass string formats of the numbers
- error reporting - you'll get information about the conversion outcome

All in all, with C++17, you have two sets of functions:

- `from_chars` - for conversion from strings into numbers, integer and floating points.
- `to_chars` - for converting numbers into string.

Let's have a look at the functions in a bit more detail.

Converting From Characters to Numbers: `from_chars`

`from_chars` is a set of overloaded functions: for integral types and floating-point types.

For integral types we have the following functions:

```
std::from_chars_result from_chars(const char* first,
                                  const char* last,
                                  TYPE &value,
                                  int base = 10);
```

Where `TYPE` expands to all available signed and unsigned integer types and `char`.

`base` can be a number ranging from 2 to 36.

Then there's the floating point version:

```
std::from_chars_result from_chars(const char* first,
                   const char* last,
                   FLOAT_TYPE& value,
                   std::chars_format fmt = std::chars_format::general);
```

`FLOAT_TYPE` expands to `float`, `double` or `long double`.

`chars_format` is an enum with the following values:

```
enum class chars_format {
    scientific = /*unspecified*/,
    fixed = /*unspecified*/,
    hex = /*unspecified*/,
    general = fixed | scientific
};
```

It's a bit-mask type, that's why the values for enums are implementation-specific. By default, the format is set to be `general` so the input string can use "normal" floating-point format with scientific form as well.

The return value in all of those functions (for integers and floats) is `from_chars_result`:

```
struct from_chars_result {
    const char* ptr;
    std::errc ec;
};
```

`from_chars_result` holds valuable information about the conversion process.

Here's the summary:

- On **Success** `from_chars_result::ptr` points at the first character not matching the pattern, or has the value equal to `last` if all characters match and `from_chars_result::ec` is value-initialized.
- On **Invalid conversion** `from_chars_result::ptr` equals `first` and `from_chars_result::ec` equals `std::errc::invalid_argument`. `value` is unmodified.
- On **Out of range** - The number is too large to fit into the value type. `from_chars_result::ec` equals `std::errc::result_out_of_range` and `from_chars_result::ptr` points at the first character not matching the pattern. `value` is unmodified.

Examples

To sum up this section, here are two examples of how to convert a string into a number using `from_chars`. The first one will convert into `int` and the second one converts into a floating-point number.

1) Integral types

Chapter String Conversions/from_chars_basic.cpp

```
#include <charconv> // from_char, to_char
#include <iostream>
#include <string>

int main() {
    const std::string str { "12345678901234" };
    int value = 0;
    const auto res = std::from_chars(str.data(),
                                     str.data() + str.size(),
                                     value);

    if (res.ec == std::errc()) {
        std::cout << "value: " << value
                  << ", distance: " << res.ptr - str.data() << '\n';
    }
    else if (res.ec == std::errc::invalid_argument) {
        std::cout << "invalid argument!\n";
    }
    else if (res.ec == std::errc::result_out_of_range) {
        std::cout << "out of range! res.ptr distance: "
                  << res.ptr - str.data() << '\n';
    }
}
```

The example is straightforward. It passes a string `str` into `from_chars` and then displays the result with additional information if possible.

Below you can find an output for various `str` value.

`str` value	output
`12345`	`value: 12345, distance 5`
`-123456`	`value: -123456, distance: 7`
`12345678901234`	`out of range! res.ptr distance: 14`
`hfhfyt`	`invalid argument!`

In the case of `12345678901234`, the conversion routine could parse the number (all 14 characters were checked), but it's too large to fit in `int` thus we got `out_of_range`.

2) Floating Point

To get the floating point test, we can replace the top lines of the previous example with:

Chapter String Conversions/from_chars_basic_float.cpp

```
const std::string str { "16.78" };
double value = 0;
const auto format = std::chars_format::general;
const auto res = std::from_chars(str.data(),
                                 str.data() + str.size(),
                                 value,
                                 format);
```

The main difference is the last parameter: `format`.

Here's the example output that we get:

`str` value	`format` value	output
`1.01`	`fixed`	`value: 1.01, distance 4`
`-67.90000`	`fixed`	`value: -67.9, distance: 9`
`1e+10`	`fixed`	`value: 1, distance: 1` - scientific notation not supported
`1e+10`	`fixed`	`value: 1, distance: 1` - scientific notation not supported
`20.9`	`scientific`	`invalid argument!, res.p distance: 0`
`20.9e+0`	`scientific`	`value: 20.9, distance: 7`
`-20.9e+1`	`scientific`	`value: -209, distance: 8`
`F.F`	`hex`	`value: 15.9375, distance: 3`
`-10.1`	`hex`	`value: -16.0625, distance: 5`

The `general` format is a combination of `fixed` and `scientific` so it handles regular floating-point string with the additional support for `e+num` syntax.

You have a basic understanding of converting from strings to numbers, so let's have a look at how to do it the opposite way.

Parsing a Command Line

In the `std::variant` chapter, there's an example with parsing command line parameters. The example uses `from_chars` to match the best type: `int`, `float` or `std::string` and then stores it in a `std::variant`.

You can find the example here: Parsing a Command Line, the Variant Chapter

Converting Numbers into Characters: `to_chars`

`to_chars` is a set of overloaded functions for integral and floating-point types.

For integral types there's one declaration:

```
std::to_chars_result to_chars(char* first, char* last,
                              TYPE value, int base = 10);
```

Where `TYPE` expands to all available signed and unsigned integer types and `char`.

Since `base` might range from 2 to 36, the output digits that are greater than 9 are represented as lowercase letters: `a...z`.

For floating-point numbers, there are more options.

Firstly there's a basic function:

```
std::to_chars_result to_chars(char* first, char* last, FLOAT_TYPE value);
```

`FLOAT_TYPE` expands to `float`, `double` or `long double`.

The conversion works the same as with `printf` and in default ("C") locale. It uses `%f` or `%e` format specifier favouring the representation that is the shortest.

The next function adds `std::chars_format fmt` that let's you specify the output format:

```
std::to_chars_result to_chars(char* first, char* last,
                              FLOAT_TYPE value,
                              std::chars_format fmt);
```

Then there's the "full" version that allows also to specify `precision`:

```
std::to_chars_result to_chars(char* first, char* last,
                              FLOAT_TYPE value,
                              std::chars_format fmt,
                              int precision);
```

When the conversion is successful, the range `[first, last)` is filled with the converted string.

The returned value for all functions (for integer and floating-point support) is `to_chars_-result`, it's defined as follows:

```
struct to_chars_result {
    char* ptr;
    std::errc ec;
};
```

The type holds information about the conversion process:

- On **Success** - `ec` equals value-initialized `std::errc` and `ptr` is the one-past-the-end pointer of the characters written. Note that the string is not NULL-terminated.
- On **Error** - `ptr` equals `first` and `ec` equals `std::errc::invalid_argument`. `value` is unmodified.
- On **Out of range** - `ec` equals `std::errc::value_too_large` the range `[first, last)` in unspecified state.

An Example

To sum up, here's a basic demo of `to_chars`.

At the time of writing there was no support for floating-point overloads, so the example uses only integers.

Chapter String Conversions/to_chars_basic.cpp

```
#include <iostream>
#include <charconv> // from_chars, to_chars
#include <string>

int main() {
    std::string str { "xxxxxxxx" };
    const int value = 1986;

    const auto res = std::to_chars(str.data(),
                                   str.data() + str.size(),
                                   value);

    if (res.ec == std::errc()) {
        std::cout << str << ", filled: "
                  << res.ptr - str.data() << " characters\n";
    }
    else {
        std::cout << "value too large!\n";
    }
}
```

Below you can find a sample output for a set of numbers:

`value`	output
`1986`	`1986xxxx, filled: 4 characters`
`-1986`	`-1986xxx, filled: 5 characters`
`19861986`	`19861986, filled: 8 characters`
`-19861986`	`value too large!` (the buffer is only 8 characters)

The Benchmark

So far, the chapter has mentioned the huge performance potential of the new routines. It would be best to see some real numbers then!

This section introduces a benchmark that measures the performance of `from_chars` and `to_chars` against other conversion methods.

How does the benchmark work:

- Generates vector of random integers of the size `VECSIZE`.
- Each pair of conversion methods will transform the input vector of integers into a vector of strings and then back to another vector of integers. This round-trip will be verified so that the output vector is the same as the input vector.
- The conversion is performed `ITER` times.
- Errors from the conversion functions are not checked.
- The code tests:
  - `from_char/to_chars`
  - `to_string/stoi`
  - `sprintf/atoi`
  - `ostringstream/istringstream`

You can find the full benchmark code in:

“Chapter String Conversions/conversion_benchmark.cpp”

Here’s the code for `from_chars/to_chars`:

Chapter String Conversions/conversion_benchmark.cpp

```cpp
const auto numIntVec = GenRandVecOfNumbers(vecSize);
std::vector<std::string> numStrVec(numIntVec.size());
std::vector<int> numBackIntVec(numIntVec.size());

std::string strTmp(15, ' ');

RunAndMeasure("to_chars", [&]() {
    for (size_t iter = 0; iter < ITERS; ++iter) {
        for (size_t i = 0; i < numIntVec.size(); ++i) {
            const auto res = std::to_chars(strTmp.data(),
                                           strTmp.data() + strTmp.size(),
                                           numIntVec[i]);
            numStrVec[i] = std::string_view(strTmp.data(),
                                            res.ptr - strTmp.data());
        }
    }
    return numStrVec.size();
});

RunAndMeasure("from_chars", [&]() {
    for (size_t iter = 0; iter < ITERS; ++iter) {
        for (size_t i = 0; i < numStrVec.size(); ++i) {
            std::from_chars(numStrVec[i].data(),
                            numStrVec[i].data() + numStrVec[i].size(),
                            numBackIntVec[i]);
        }
    }
    return numBackIntVec.size();
});

CheckVectors(numIntVec, numBackIntVec);
```

`CheckVectors` - checks if the two input vectors of integers contain the same values and prints mismatches on error.

The benchmark converts `vector<int>` into `vector<string>` and we measure the whole conversion process which also includes the string object creation.

Here are the results (time in milliseconds) of running 1000 iterations on a vector with 1000 elements:

Method	GCC 8.2	Clang 7.0 Win	VS 2017 15.8 x64
`to_chars`	21.94	18.15	24.81
`from_chars`	15.96	12.74	13.43
`to_string`	61.84	16.62	20.91
`stoi`	70.81	45.75	42.40
`sprintf`	56.85	124.72	131.03
`atoi`	35.90	34.81	32.50
`ostringstream`	264.29	681.29	575.95
`stringstream`	306.17	789.04	664.90

The machine: Windows 10 x64, i7 8700 3.2 GHz base frequency, 6 cores/12 threads (although the benchmark uses only one thread for processing).

- GCC 8.2 - compiled with `-O2 -Wall -pedantic`, MinGW Distro[2]
- Clang 7.0 - compiled with `-O2 -Wall -pedantic`, Clang For Windows[3]
- Visual Studio 2017 15.8 - Release mode, x64

Some notes:

- On GCC `to_chars` is almost 3x faster than `to_string`, 2.6x faster than `sprintf` and 12x faster than `ostringstream`!
- On Clang `to_chars` is a bit slower than `to_string`, but ~7x faster than `sprintf` and surprisingly almost 40x faster than `ostringstream`!
- MSVC also has slower performance in comparison with `to_string`, but then `to_-chars` is ~5x faster than `sprintf` and ~23x faster than `ostringstream`.

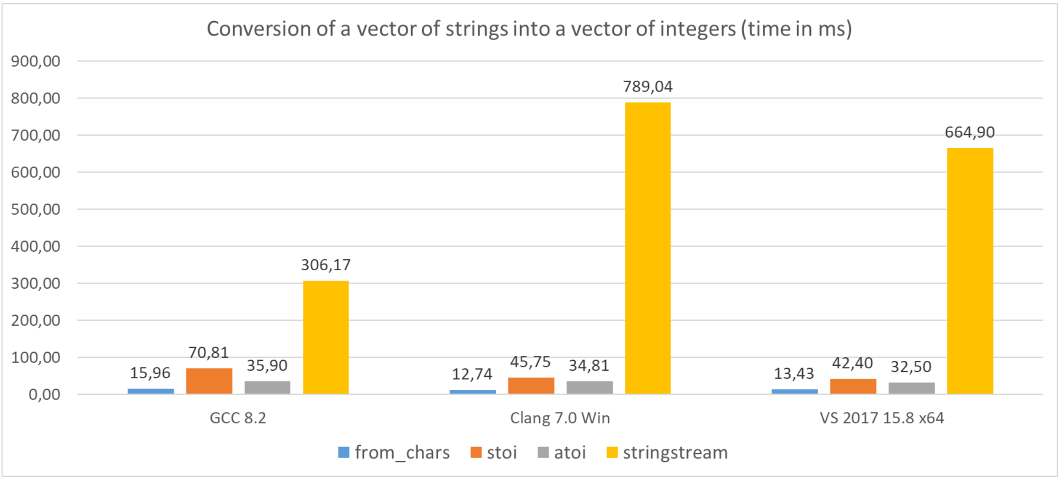
Looking now at `from_chars` :

- On GCC it's ~4,5x faster than `stoi`, 2,2x faster than `atoi` and almost 20x faster than `istringstream`.
- On Clang it's ~3,5x faster than `stoi`, 2.7x faster than `atoi` and 60x faster than `istringstream`!
- MSVC performs ~3x faster than `stoi`, ~2,5x faster than `atoi` and almost 50x faster than `istringstream`!

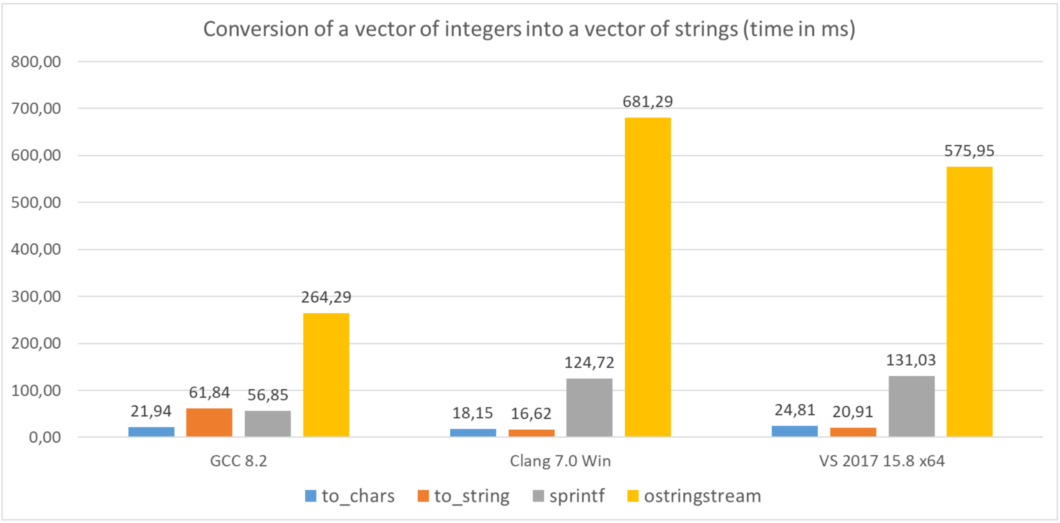
[2]https://nuwen.net/mingw.html
[3]http://releases.llvm.org/dow4

As mentioned earlier, the benchmark also includes the cost of string object creation. That's why `to_string` (optimised for strings) might perform a bit better than `to_chars`. If you already have a char buffer, and you don't need to create a string object, then `to_chars` should be faster.

Here are the two charts built from the table above.



Strings into Numbers, time in milliseconds



Numbers into Strings, time in milliseconds

As always, it's encouraged to run the benchmarks on your own before you make the final judgment. You might get different results in your environment, where maybe a different compiler or STL library implementation is available.

Summary

This chapter showed how to use two sets of functions `from_chars` - to convert strings into numbers, and `from_chars` that converts numbers into their textual representations.

The functions might look very raw and even C-style. This is a "price" you have to pay for having such low-level support, performance, safety and flexibility. The advantage is that you can provide a simple wrapper that exposes only the needed parts that you want.

Extra Info

The change was proposed in: P0067[4].

Compiler support

Feature	GCC	Clang	MSVC
Elementary String Conversions	8.0[5]	7.0[6]	VS 2017 15.7/15.8[7]

[4]https://wg21.link/P0067
[5]In progress, only integral types are supported
[6]In progress, only integral types are supported
[7]Integer support for `from_chars`/`to_chars` available in 15.7, floating-point support for `from_chars` ready in 15.8. Floating-point `to_chars` should be ready with 15.9. See STL Features and Fixes in VS 2017 15.8 | Visual C++ Team Blog.

12. Searchers & String Matching

`std::search` in C++14 offers a generic way to search for a pattern in a given range. The algorithm can be used not only for character containers but also for containers with custom types. This technique was, unfortunately, a bit limited as the performance was usually slow - it uses the naive matching algorithm, with the complexity of the size of the pattern times the size of the text. With C++17 we get new `std::search` overloads that expose new and powerful algorithms like Boyer Moore variations that have linear complexity in the average case.

In this chapter, you'll learn:

- How we can beat a naive search algorithm with pattern preprocessing.
- How you can use `std::search` to efficiently search for a pattern in a range.
- How to use `std::search` for custom types.

Overview of String Matching Algorithms

String-matching consists of finding one or all of the occurrences of a string ("pattern") in a text. The strings are built over a finite set of characters, called "alphabet".

There are lots of algorithms that solve this problem; here's a short list from Wikipedia[1]:

Algorithm	Preprocessing	Matching	Space
Naive string-search	none	`O(nm)`	none
Rabin–Karp	`O(m)`	average `O(n + m)`, worst `O((n-m)m)`	`O(1)`
Knuth–Morris–Pratt	`O(m)`	`O(n)`	`O(m)`
Boyer–Moore	`O(m + k)`	best `O(n/m)`, worst `O(mn)`	`O(k)`
Boyer–Moore-Horspool	`O(m + k)`	best `O(n/m)`, worst `O(mn)`	`O(k)`

`m` - the length of the pattern `n` - the length of the text `k` - the size of the alphabet

The naive algorithm tries to match the pattern at each position of the text:

```
Pattern = Hello
Text = SuperHelloWorld

   SuperHelloWorld
1. Hello <- XX
2.  Hello <- XX
3.   Hello <- XX
4.    Hello <- XX
5.     Hello <- XX
6.      Hello <- OK!
```

In the example above we're looking for "Hello" in "SuperHelloWorld". As you can see, the naive version tries each position until it finds the "Hello" at the 6th iteration.

The main difference between the naive way and the other algorithms is that the faster algorithms use additional knowledge about the input pattern. That way, they can skip a lot of fruitless comparisons.

To gain that knowledge, they usually build some lookup tables for the pattern in the preprocessing phase. The size of lookup tables is often tied to the size of the pattern and the alphabet.

[1]https://en.wikipedia.org/wiki/String-searching_algorithm#Single-pattern_algorithms

In the above case we can skip most of the iterations, as we can observe that when we try to match `Hello` in the first position, there's a difference at the last letter `o` vs `r`. In fact, since `r` doesn't occur in our pattern at all, we can actually move 5 steps further.

```
Pattern = Hello
Text = SuperHelloWorld

   SuperHelloWorld
1. Hello <- XX,
2.      Hello <- OK
```

We have a match with just 2 iterations! The rule that was used in that example comes from the Boyer Moore algorithm - it's called The Bad Character Rule.

In C++ string matching is implemented through `std::search`, which finds a range (the pattern) inside another range:

```
template< class ForwardIt1, class ForwardIt2 >
ForwardIt1 search( ForwardIt1 first, ForwardIt1 last,
                   ForwardIt2 s_first, ForwardIt2 s_last );
```

Before C++17, you had no control over the algorithm selection inside `std::search`. The complexity of `std::search` was specified as `O(mn)` - so it was usually the naive approach. Now, in C++17, you have a few more options.

To be precise, there's also `string::find` that works exclusively with character sequences. There might be different implementations of this method, and you don't have control over the algorithm that is used inside. In the examples section, you'll see some performance experiments that also compare the new algorithms for `std::search` with `string::find`.

New Algorithms Available in C++17

C++17 updated `std::search` algorithm in two ways:

- you can now use execution policy to run the default version of the algorithm in a parallel way.
- you can provide a Searcher object that handles the search.

In C++17, we have three searchers:

- `default_searcher` - same as the version before C++17, usually meaning the naive approach. Operates on Forward Iterators.
- `boyer_moore_searcher` - uses Boyer Moore Algorithm - the full version, with two rules: bad character rule and good suffix rule. Operates on Random Access Iterators.
- `boyer_moore_horspool_searcher` - Simplified version of Boyer-Moore that uses only Bad Character Rule, but still has good average complexity. Operates on Random Access Iterators.

`std::search` with a searcher cannot be used along with execution policy.

Using Searchers

The `std::search` function uses the following overload for searchers:

```
template<class ForwardIterator, class Searcher>
ForwardIterator search(ForwardIterator first, ForwardIterator last,
                       const Searcher& searcher );
```

For example:

```
string testString = "Hello Super World";
string needle = "Super";
const auto it = search(begin(testString), end(testString),
                       boyer_moore_searcher(begin(needle), end(needle));

if (it == cend(testString))
    cout << "The string " << needle << " not found\n";
```

Each searcher initialises its state through the constructor. The constructors need to store the pattern range and also perform the preprocessing phase. Then `std::search` calls their `operator()(iter TextFirst, iter TextLast)` method to perform the search in the text range.

Since a searcher is an object, you can pass it around in the application. That might be useful if you'd like to search for the same pattern inside various text objects. In that case, the preprocessing phase will be done only once.

Examples

Performance Experiments

This example builds a performance test to exercise several ways of finding a pattern in a larger text.

Here's how the test works:

- the application loads a text file (configurable via a command-line argument), for example, a book sample (like a 500KB text file)
- the entire file content is stored in one string - that will be "text" where we'll be doing the lookups.
- a pattern is selected - N letters from the input text. That way, we can be sure the pattern can be found. The position of the string can be located at the front, centre or at the end.
- the benchmark uses several algorithms and runs each search `ITER` times.

You can find the example in:

`Chapter Searchers/searchers_benchmark.cpp`

Example benchmarks:

the `std::string::find` version:

```
RunAndMeasure("string::find", [&]() {
    for (size_t i = 0; i < ITERS; ++i) {
        std::size_t found = testString.find(needle);
        if (found == std::string::npos)
            std::cout << "The string " << needle << " not found\n";
    }
});
```

The `boyer_moore_horspool` version:

```
RunAndMeasure("boyer_moore_horspool_searcher", [&]() {
    for (size_t i = 0; i < ITERS; ++i) {
        auto it = std::search(testString.begin(), testString.end(),
            std::boyer_moore_horspool_searcher(
                needle.begin(), needle.end()));
        if (it == testString.end())
            std::cout << "The string " << needle << " not found\n";
    }
});
```

`RunAndMeasure` is a function that takes a callable object to execute (for example a lambda). It measures the time of that execution and prints the results.

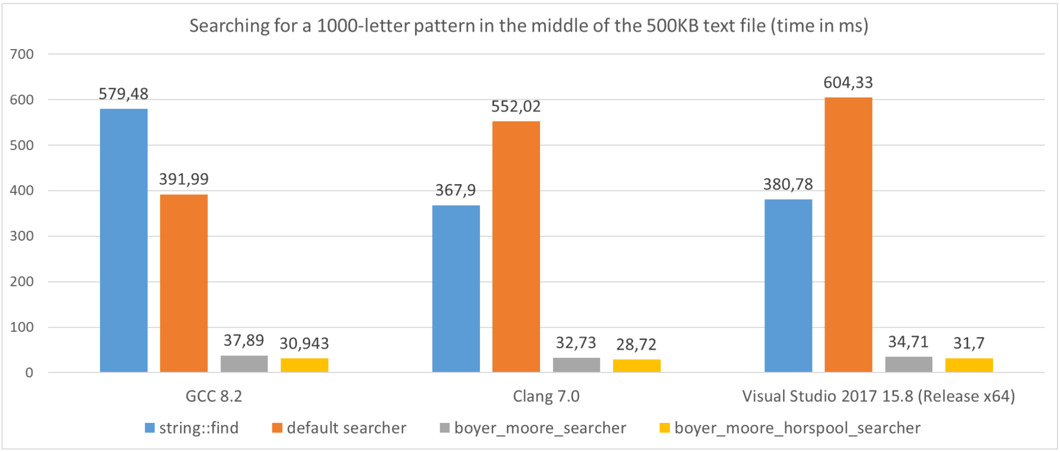
Since the input string is loaded from a file, the compiler cannot trick us and won't optimise code away.

Here are some of the results running the application on Win 10 64bit, i7 8700, 3.20 GHz base frequency, 6 cores/ 12 threads (the application runs on a single thread, however).

The string size is 547412 bytes (comes from a 500KB text file), and we run the benchmark 1000 times.

Algorithm	GCC 8.2	Clang 7.0	Visual Studio (Release x64)
`string::find`	579.48	367.90	380.78
`default searcher`	391.99	552.02	604.33
`boyer_moore_searcher`	37.89 (init 3.98)	32.73 (init 3.02)	34.71 (init 3.52)
`boyer_moore_horspool_searcher`	30.943 (init 0)	28.72 (init 0.5)	31.70 (init 0.69)

When searching for 1000 letters from the centre of the input string, both of the new algorithms were faster than the default searcher and `string::find`. `boyer_moore` uses more time to perform the initialisation than `boyer_moore_horspool` (it creates two lookup tables, rather than one, so it will use more space and preprocessing). The results also show that `boyer_moore` usually takes longer time to preprocess the input pattern than `boyer_moore_horspool`. And also, the second algorithm is faster in our case. But all in all, the new algorithms perform even 10…15x faster than the default versions.

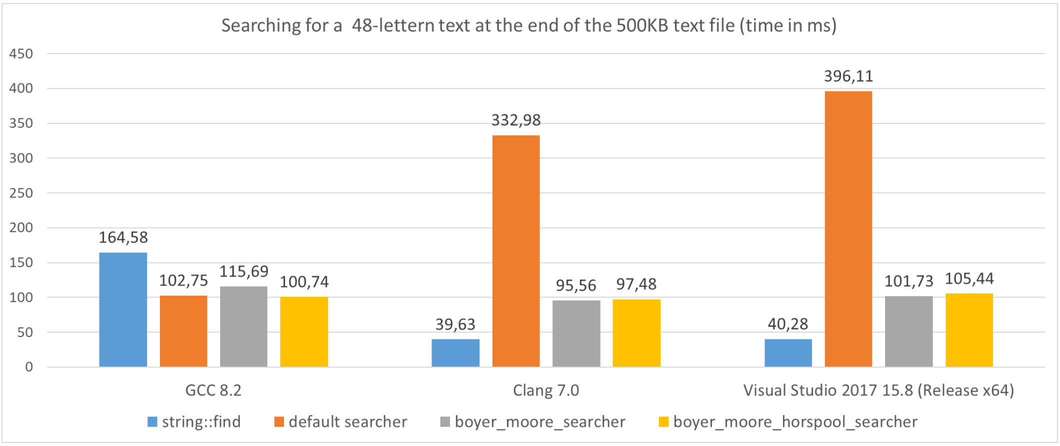


Searching for a 1000-letter pattern in the middle of the 500KB text file

Here are the results from another run, this time we use the same input string (from a 500KB text file), we perform 1000 iterations, but the pattern is only 48 letters. It's a sentence that's located at the end of the file (a single occurrence).

Algorithm	GCC 8.2	Clang 7.0	Visual Studio (Release x64)
`string::find`	164.58	39.63	40.28
`default searcher`	102.75	332.98	396.11
`boyer_moore_searcher`	115.69 (init 0.96)	95.56 (init 0.45)	101.73 (init 0.49)
`boyer_moore_-horspool_searcher`	100.74 (init 0)	97.48 (init 0.21)	105.44 (init 0.23)

In this test, Boyer-Moore algorithms in Visual Studio and Clang are 2.5x slower than `string::find`. However, on GCC `string::find` performed worse, and `boyer_-moore_horspool` is the fastest.



Searching for a 48-lettern text at the end of the 500KB text file

You can run the experiments and see how your STL implementation performs. There are many ways to configure the benchmark so you can test various positions (beginning, centre, end) of the text, or check for some string pattern.

DNA Matching

To demonstrate the range of uses for `std::search`, let's have a look at a simple DNA matching demo. The example will match custom types rather than regular characters.

For instance, we'd like to search a DNA sequence to see whether `GCTGC` occurs in the sequence `CTGATGTTAAGTCAACGCTGC`.

The application uses a simple data structure for Nucleotides:

Chapter Searchers/dna_demo.cpp

```
struct Nucleotide {
    enum class Type : uint8_t {
        A = 0,
        C = 1,
        G = 3,
        T = 2
    };

    Type mType;

    friend bool operator==(Nucleotide a, Nucleotide b) noexcept {
        return a.mType == b.mType;
    }

    static char ToChar(Nucleotide t);
    static Nucleotide FromChar(char ch);
};
```

With the two converting static methods:

Chapter Searchers/dna_demo.cpp

```
char Nucleotide::ToChar(Nucleotide t) {
    switch (t.mType) {
    case Nucleotide::Type::A: return 'A';
    case Nucleotide::Type::C: return 'C';
    case Nucleotide::Type::G: return 'G';
    case Nucleotide::Type::T: return 'T';
    }
    return 0;
}

Nucleotide Nucleotide::FromChar(char ch) {
    return Nucleotide { static_cast<Nucleotide::Type>((ch >> 1) & 0x03) };
}
```

And the two functions that work on a whole string:

Chapter Searchers/dna_demo.cpp

```
std::vector<Nucleotide> FromString(const std::string& s) {
    std::vector<Nucleotide> out;
    out.reserve(s.length());
    std::transform(std::cbegin(s), std::cend(s),
                   std::back_inserter(out), Nucleotide::FromChar);
    return out;
}

std::string ToString(const std::vector<Nucleotide>& vec) {
    std::stringstream ss;
    std::ostream_iterator<char> out_it(ss);
    std::transform(std::cbegin(vec), std::cend(vec), out_it, Nucleotide::ToChar);
    return ss.str();
}
```

The demo uses `boyer_moore_horspool_searcher` which requires hashing support. So we have to define it as follows:

Chapter Searchers/dna_demo.cpp

```
namespace std {
    template<> struct hash<Nucleotide> {
        size_t operator()(Nucleotide n) const noexcept {
            return std::hash<Nucleotide::Type>{}(n.mType);
        }
    };
}
```

`std::hash` has support for enums, so we just have to “redirect” it from the whole class.

And then the test code:

Chapter Searchers/dna_demo.cpp

```
const std::vector<Nucleotide> dna = FromString("CTGATGTTAAGTCAACGCTGC");
std::cout << ToString(dna) << '\n';
const std::vector<Nucleotide> s = FromString("GCTGC");
std::cout << ToString(s) << '\n';

std::boyer_moore_horspool_searcher searcher(std::cbegin(s), std::cend(s));
const auto it = std::search(std::cbegin(dna), std::cend(dna), searcher);

if (it == std::cend(dna))
    std::cout << "The pattern " << ToString(s) << " not found\n";
else {
    std::cout << "DNA matched at position: "
              << std::distance(std::cbegin(dna), it) << '\n';
}
```

As you can see, the example builds a vector of custom types - Nucleotides. To satisfy the searcher, a custom type needs to support `std::hash` interface and also define `operator==`.

The Nucleotide type wastes a bit of space - as we use the full byte just to store four options - C T G A. We could use only 2 bits, though the implementation would be more complicated. Another option is to represent the triplets of Nucleotides - Codons. Each codon can be expressed in 6 bits, so that way we’d use the full byte more efficiently.

Summary

In this chapter, you've learned about the searchers that can be passed into `std::search` algorithm. They allow you to use more advanced algorithms for string matching - Boyer-Moore and Boyer-Moore-Horspool that offers better complexity than a naive approach.

`std::search` with searchers is a general algorithm that works for most of the containers that expose random access iterators. If you work with strings and characters, then you might also compare it against `std::string::find`, which is usually specialised and optimised for character processing (implementation-dependent!).

Extra Info

The change was proposed in: N3905[2].

Compiler support

Feature	GCC	Clang	MSVC
Searchers	7.1	3.9	VS 2017 15.3

[2]https://wg21.link/n3905

13. Filesystem

Since early versions, the Standard Library has included an option to work with files. Through streams - like `fstream` - you can open files, read data, write bytes and perform many other operations. However, what was missing was an ability to work with the filesystem as a whole. For example, in C++14 you had to use some third party libraries to iterate over directories, compose paths, delete directories or read file permissions. Now with C++17, we've taken a big step forward in the form of the `std::filesystem` component!

In this chapter, you'll learn:

- How `std::filesystem` got into the Standard
- What the basic types and operations are
- How you can work with the paths
- How to handle errors in `std::filesystem`
- How to iterate over a directory
- How to create new directories and files

Filesystem Overview

While the Standard Library lacked some important features, you could always use Boost with its dozens of sub-libraries and do the work. The C++ Committee decided that the Boost libraries are very important and some parts of it were merged into the Standard. For example, smart pointers (although improved with the move semantics in C++11), regular expressions, `std::optional`, `std::any` and much more.

A similar story happened with `std::filesystem`.

The filesystem library is modelled directly from Boost filesystem, which has been available since 2003 (with the version 1.30). In C++ implementation, the committee also extended the component with non-POSIX systems. The library was first available as TS (Technical Specification) and later, after a long time of improvements and feedback, merged into the C++17 Standard.

The library is located in the `<filesystem>` header, and it uses namespace `std::filesystem`.

Core Parts of The Library

The filesystem library is a rather significant part of the Standard Library. It defines many types with dozens of methods, and also gives us many free functions.

Below we can define the core elements of this module:

- The `std::filesystem::path` object allows you to manipulate paths that represent existing or not existing files and directories in the system.
- `std::filesystem::directory_entry` represents an existing path with additional status information like last write time, file size, or other attributes.
- Directory iterators allow you to iterate through a given directory. The library provides a recursive and non-recursive version.
- Many supporting functions like getting information about the path, file manipulation, permissions, creating directories, and many more.

In the next section, you'll see a demo of all the parts that compose `std::filesystem`.

Demo

Instead of exploring the library piece by piece at the start, on the next page, you'll see a demo example: displaying basic information about all the files in a given directory (recursively). This should give you a high-level overview of what the library looks like.

Chapter Filesystem/filesystem_list_files.cpp

```cpp
#include <filesystem>
#include <iomanip>
#include <iostream>
namespace fs = std::filesystem;

void DisplayDirectoryTree(const fs::path& pathToScan, int level = 0) {
    for (const auto& entry : fs::directory_iterator(pathToScan)) {
        const auto filenameStr = entry.path().filename().string();
        if (entry.is_directory()) {
            std::cout << std::setw(level*3) << "" << filenameStr << '\n';
            DisplayDirectoryTree(entry, level + 1);
        }
        else if (entry.is_regular_file()) {
            std::cout << std::setw(level*3) << ""<< filenameStr
                << ", size " << fs::file_size(entry) << " bytes\n";
        }
        else
            std::cout << std::setw(level*3) << "" << " [?]" << filenameStr << '\n';
    }
}
int main(int argc, char* argv[]) {
    try  {
        const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };
        std::cout << "listing files in the directory: "
                  << fs::absolute(pathToShow).string() << '\n';
        DisplayDirectoryTree(pathToShow);
    }
    catch (const fs::filesystem_error& err) {
        std::cerr << "filesystem error! " << err.what() << '\n';

    }
    catch (const std::exception& ex) {
        std::cerr << "general exception: " << ex.what() << '\n';
    }
}
```

We can run this program on a temp path `D:\testlist` and see the following output:

Running on Windows:

```
.\ListFiles.exe D:\testlist\
listing files in the directory: D:\testlist\
abc.txt, size 357 bytes
def.txt, size 430 bytes
ghi.txt, size 190 bytes
dirTemp
   jkl.txt, size 162 bytes
   mno.txt, size 1728 bytes
tempDir
   abc.txt, size 174 bytes
   def.txt, size 163 bytes
   tempInner
      abc.txt, size 144 bytes
      mno.txt, size 1728 bytes
      xyz.txt, size 3168 bytes
```

The application lists files recursively, and with each indentation, you can see that we enter a new directory.

Running on a Linux (Ubuntu 18.04 on WSL):

```
fenbf@FEN-NODE:/mnt/f/wsl$ ./list_files.out testList/
listing files in the directory: /mnt/f/wsl/testList/
a.txt, size 965 bytes
b.txt, size 1667 bytes
c.txt, size 1394 bytes
d.txt, size 1408 bytes
directoryTemp
   a.txt, size 1165 bytes
   b.txt, size 1601 bytes
tempDir
   a.txt, size 1502 bytes
   b.txt, size 1549 bytes
   x.txt, size 1487 bytes
```

Let’s now examine the core elements of this demo.

To work with the library, we have to include relevant headers. For the filesystem library it's:

```
#include <filesystem>
```

All the types, functions and names live in the `std::filesystem` namespace. For convenience it's useful to make a namespace alias:

```
namespace fs = std::filesystem;
```

And now we can refer to the names as `fs::path` rather than `std::filesystem::path`.

Let's start with the `main()` function where the logic of the application starts.

The program takes a single optional argument from the command line. If it's empty then we use the current system path:

```
const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };
```

`pathToShow` can be created from strings - from `argv[1]` if available. If not, then we take `current_path()` which is a helper free function that returns the current system path.

In the next two lines - line 28 and 29 - we display the starting path for the iteration. The `absolute()` function "expands" the input path and it converts it from a relative form into the absolute form if required.

The core part of the application is the `DisplayDirectoryTree()` function.

Inside, we use `directory_iterator` to examine the directory and find other paths:

```
for (const auto& entry : fs::directory_iterator(pathToShow))
```

Each loop iteration yields another `directory_entry` that we need to check. We can decide if we should call the function recursively (when `entry.is_directory()` is `true`) or just show some basic information if it's a regular file.

As you can see, we have access to many methods of path and directory entry. For example, we use `filename` to return only the filename part of the path so we can display "tree" structure. We also invoke `fs::file_size` to query the size of the file.

After a little demo, let's now have a closer look at `filesystem::path`, `filesystem::directory_entry`, supporting, non-member functions and error handling.

The Path Object

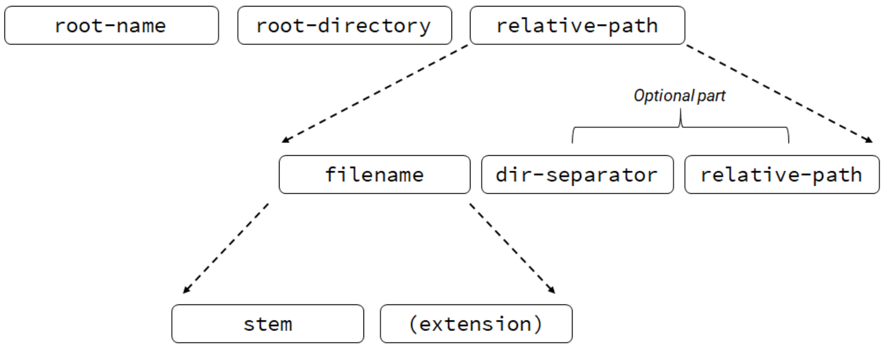
The core part of the library is the `path` object. It contains a pathname - a string that forms the name of the path. The object doesn't have to point to an existing file in the filesystem. The path might even be in an invalid form.

The path is composed of the following elements:

`root-name root-directory relative-path`:

- (optional) root-name: POSIX systems don't have a root name. On Windows, it's usually the name of a drive, like `"C:"`
- (optional) root-directory: distinguishes relative path from the absolute path
- relative-path:
  - filename
  - directory separator
  - relative-path

We can illustrate it with the following diagram:



The Path Structure

The class implements a lot of methods that extracts the parts of the path:

Method	Description
`path::root_name()`	returns the root-name of the path
`path::root_directory()`	returns the root directory of the path
`path::root_path()`	returns the root path of the path
`path::relative_path()`	returns path relative to the root path
`path::parent_path()`	returns the path of the parent path
`path::filename()`	returns the filename path component
`path::stem()`	returns the stem path component
`path::extension()`	returns the file extension path component

If a given element is not present, then the above functions return an empty path.

There are also methods that query elements of the path:

Query name	Description
`path::has_root_path()`	queries if a path has a root
`path::has_root_name()`	queries if a path has a root name
`path::has_root_directory()`	checks if a path has a root directory
`path::has_relative_path()`	checks if a path has a relative path component
`path::has_parent_path()`	checks if a path has a parent path
`path::has_filename()`	checks if a path has a filename
`path::has_stem()`	checks if a path has a stem component
`path::has_extension()`	checks if a path has an extension

We can use all of the above methods and compose an example that shows info about a given path:

Chapter Filesystem/filesystem_path_info.cpp

```
const filesystem::path testPath{...};

if (testPath.has_root_name())
    cout << "root_name() = " << testPath.root_name() << '\n';
else
    cout << "no root-name\n";

if (testPath.has_root_directory())
    cout << "root directory() = " << testPath.root_directory() << '\n';
else
    cout << "no root-directory\n";

if (testPath.has_root_path())
    cout << "root_path() = " << testPath.root_path() << '\n';
else
```

```
    cout << "no root-path\n";

if (testPath.has_relative_path())
    cout << "relative_path() = " << testPath.relative_path() << '\n';
else
    cout << "no relative-path\n";

if (testPath.has_parent_path())
    cout << "parent_path() = " << testPath.parent_path() << '\n';
else
    cout << "no parent-path\n";

if (testPath.has_filename())
    cout << "filename() = " << testPath.filename() << '\n';
else
    cout << "no filename\n";

if (testPath.has_stem())
    cout << "stem() = " << testPath.stem() << '\n';
else
    cout << "no stem\n";

if (testPath.has_extension())
    cout << "extension() = " << testPath.extension() << '\n';
else
    cout << "no extension\n";
```

Here's an output for a file path like `"C:\Windows\system.ini"`:

```
root_name() = "C:"
root directory() = "\\"
root_path() = "C:\\"
relative_path() = "Windows\\system.ini"
parent_path() = "C:\\Windows"
filename() = "system.ini"
stem() = "system"
extension() = ".ini"
```

Similarly, the output, fom a POSIX system, for a path `/usr/temp/abc.txt`:

```
no root-name
root directory() = "/"
root_path() = "/"
relative_path() = "usr/temp/abc.txt"
parent_path() = "/usr/temp"
filename() = "abc.txt"
stem() = "abc"
extension() = ".txt"
```

There's also a trick that lets you iterate over the parts of a path object. `std::filesystem::path` implements overloads for `begin()` and `end()` and that's why you can use it in a range based for loop:

```
int i = 0;
for (const auto& part : testPath)
    cout << "path part: " << i++ << " = " << part << '\n';
```

The output for `C:\Windows\system.ini`:

```
path part: 0 = C:
path part: 1 = \
path part: 2 = Windows
path part: 3 = system.ini
```

Path Operations

Below you can find a table with other important methods of the `path` class:

Operation	Description
`path::append()`	appends one path to the other, with a directory separator
`path::concat()`	concatenates the paths, without a directory separator
`path::clear()`	erases the elements and makes it empty
`path::remove_filename()`	removes the filename part from a path
`path::replace_filename()`	replaces a single filename component
`path::replace_extension()`	replaces the extension
`path::swap()`	swaps two paths
`path::compare()`	compares the lexical representations of the path and another path, returns an integer
`path::empty()`	checks if the path is empty

Comparison

The `path` class has several overloaded operators:

`==`, `!=`, `<`, `>`, `<=`, `=>`

And the `path::compare()` method, which returns an integer value.

All methods compare element by element, using the native format of the path.

```
fs::path p1 { "/usr/a/b/c" };
fs::path p2 { "/usr/a/b/c" };
assert(p1 == p2);
assert(p1.compare(p2) == 0);

p1 = "/usr/a/b/c";
p2 = "/usr/a/b/c/d";
assert(p1 < p2);
assert(p1.compare(p2) < 0);
```

And on Windows we can also test the cases where we have a root element in a path:

```
p1 = "C:/test";
p2 = "abc/xyz"; // no root path, so it's "less" than a path with a root
assert(p1 > p2);
assert(p1.compare(p2) > 0);
```

Or, also on Windows, a case where path formats are different:

```
fs::path p3 { "/usr/a/b/c" }; // on Windows it's converted to native format
fs::path p4 { "\\usr/a\\b/c" };
assert(p3 == p4);
assert(p3.compare(p4) == 0);
```

You can play with the code in `Chapter Filesystem/filesystem_compare.cpp`.

Path Composition

We have two methods that let us compose a path:

- `path::append()` - adds a path with a directory separator.

- `path::concat()` - only adds the 'string' without any separator.

The functionality is also available with operators `/`, `/=` (append), `+` and `+=` (concat).

For example:

```
// append:
fs::path p1{"C:\\temp"};
p1 /= "user";
p1 /= "data";
cout << p1 << '\n';

// concat:
fs::path p2("C:\\temp\\");
p2 += "user";
p2 += "data";
cout << p2 << '\n';
```

The output:

```
C:\temp\user\data
C:\temp\userdata
```

However, appending a path has several rules that you have to be aware of.

For example, if the other path is absolute or the other path has a root-name, and the root-name is different from the current path root name. Then the append operation will replace `this`.

```
auto resW = fs::path{"foo"} / "D:\";   // Windows
auto resP = fs::path{"foo"} / "/bar";  // POSIX
// resW is "D:\" now
// resP is now "/bar"
```

In the above case `resW` and `resP` will contain the value from the second operand. As `D:\` and `/bar` contains root elements.

Stream Operators

The `path` class also implements `>>` and `<<` operators.

The operators use `std::quoted` to preserve the correct format. That's why the paths in the examples showed quotes.

On Windows, this will also cause the runtime to output `"\\"` for paths in native format.

For example on POSIX:

```
fs::path p1 { "/usr/test/temp.xyz" };
std::cout << p1;
```

The code will print `"/usr/test/temp.xyz"`.

And on Windows:

```
fs::path p1{ "usr/test/temp.xyz" };
fs::path p2{ "usr\\test\\temp.xyz" };
std::cout << p1 << '\n' << p2;
```

The code will output:

```
"usr/test/temp.xyz"
"usr\\test\\temp.xyz"
```

Path Formats and Conversion

The filesystem library is modelled on top of POSIX (for example all Unix Based systems implements POSIX standard), but also works with other filesystems, for instance with Windows. Because of that, there are a few things to keep in mind when using paths in a portable way.

The first thing is the path format. We have two core modes:

- generic - generic format, the format as specified by the standard (based on the POSIX format)
- native - format used by the particular implementation

In POSIX systems native format is equal to generic. But On Windows it's different.

The main difference of the format is that Windows uses backslashes (`\`) rather than slashes (`/`). Another point is that Windows has a root directory - like `C:`, `D:` or other drive letters.

One more important aspect is the string type that is used to hold path elements. In POSIX it's `char` (and `std::string`), but on Windows it's `wchar_t` and `std::wstring`. The `path` type specifies `path::value_type` and `string_type` (defined as `std::basic_string<value_type>`) to expose those properties.

The `path` class has several methods that allow you to use the best matching format.

If you want to work with the native format you can use:

Operation	Description
`path::c_str()`	returns `value_type*`
`path::native()`	returns `string_type&`

And there are many methods that will convert the native format:

Operation	Description
`path::string()`	converts to `string`
`path::wstring()`	converts to `wstring`
`path::u8string()`	converts to `u8string`
`path::u16string()`	converts to `u16string`
`path::u32string()`	converts to `u32string`

Since Windows uses `wchar_t` as the underlying type for paths, then you need to be aware of "hidden" conversions to `char`.

The Directory Entry & Directory Iteration

While the `path` class represent files or paths that exist or not, we also have another object that is more concrete: it's `directory_entry` object. This object points to existing files and directories, and it's usually obtained with the aid of filesystem iterators.

What's more, implementations are encouraged to cache the additional file attributes. That way, there can be fewer system calls.

Traversing a Path with Directory Iterators

You can traverse a path using two available iterators:

- `directory_iterator` - iterates in a single directory, input iterator.
- `recursive_directory_iterator` - iterates recursively, input iterator

In both approaches the order of the visited filenames is unspecified, each directory entry is visited only once.

If a file or a directory is deleted or added to the directory tree after the directory iterator has been created, it is unspecified whether the change would be observed through the iterator.

In both iterators the directories `.` and `..` are skipped.

You can iterate through a directory using the following pattern:

```
for (auto const & entry : fs::directory_iterator(pathToShow)) {
   ...
}
```

Or another way, with an algorithm, where you can also filter out paths:

```
std::filesystem::path inPath = /* GetInputPath() */;
std::vector<std::filesystem::directory_entry> outEntries;
std::filesystem::recursive_directory_iterator dirpos{ inPath };

std::copy_if(begin(dirpos), end(dirpos), std::back_inserter(outEntries),
             some_predicate);
```

`some_predicate` is a predicate that takes `const directory_entry&` and returns `true` or `false` depending on if a given `directory_entry` object matches our filter or not. All matching paths are pushed back to the `outEntries` vector. See "Filtering Files Using Regex" in the Examples section of this chapter to see the use case of this technique. Also, instead of the output vector of directory entries, you can use a vector of paths since directory entries can convert into paths.

`directory_entry` Methods

Here's a list of `directory_entry` methods:

Operation	Description
`directory_entry::assign()`	replaces the path inside the entry and calls `refresh()` to update the cached attributes
`directory_entry::replace_-filename()`	replaces the filename inside the entry and calls `refresh()` to update the cached attributes
`directory_entry::refresh()`	updates the cached attributes of a file
`directory_entry::path()`	returns the path stored in the entry
`directory_entry::exists()`	checks if a directory entry points to existing file system object
`directory_entry::is_block_file()`	returns true if the file entry is a block file
`directory_entry::is_character_-file()`	returns true if the file entry is a character file
`directory_entry::is_directory()`	returns true if the file entry is a directory
`directory_entry::is_fifo()`	returns true if the file entry refers to a named pipe
`directory_entry::is_other()`	returns true if the file entry is refers to another file type
`directory_entry::is_regular_file()`	returns true if the file entry is a regular file
`directory_entry::is_socket()`	returns true if the file entry is a named IPC socket
`directory_entry::is_symlink()`	returns true if the file entry is a symbolic link
`directory_entry::file_size()`	returns the size of the file pointing by the directory entry
`directory_entry::hard_link_count()`	returns the number of hard links referring to the file
`directory_entry::last_write_time()`	gets or sets the last time write for a file
`directory_entry::status()`	returns status of the file designated by this directory entry
`directory_entry::symlink_status()`	returns the symlink_status of the file designated by this directory entry

Supporting Functions

So far we've covered three elements of the filesystem: the path class, directory_entry and directory iterators. The library also provides a set of non-member functions.

Query functions:

function	description
`filesystem::is_block_file()`	checks whether the given path refers to block device
`filesystem::is_character_file()`	checks whether the given path refers to a character device
`filesystem::is_directory()`	checks whether the given path refers to a directory
`filesystem::is_empty()`	checks whether the given path refers to an empty file or directory
`filesystem::is_fifo()`	checks whether the given path refers to a named pipe
`filesystem::is_other()`	checks whether the argument refers to another file
`filesystem::is_regular_file()`	checks whether the argument refers to a regular file
`filesystem::is_socket()`	checks whether the argument refers to a named IPC socket
`filesystem::is_symlink()`	checks whether the argument refers to a symbolic link
`filesystem::status_known()`	checks whether file status is known
`filesystem::exists()`	checks whether path refers to existing file system object
`filesystem::file_size()`	returns the size of a file
`filesystem::last_write_time()`	gets or sets the time of the last data modification

Path related:

function name	description
`filesystem::absolute()`	composes an absolute path
`filesystem::canonical()`, `weakly_canonical()`	composes a canonical path
`filesystem::relativeproximate()`	composes a relative path
`filesystem::current_path()`	returns or sets the current working directory
`filesystem::equivalent()`	checks whether two paths refer to the same file system object

Directory and files management

function name	description
`filesystem::copy()`	copies files or directories
`filesystem::copy_file()`	copies file contents
`filesystem::copy_symlink()`	copies a symbolic link
`filesystem::create_directory()`, `filesystem::create_directories()`	creates new directory
`filesystem::create_hard_link()`	creates a hard link
`filesystem::create_symlink()`, `filesystem::create_directory_-symlink()`	creates a symbolic link

function name	description
`filesystem::hard_link_count()`	returns the number of hard links referring to the specific file
`filesystem::permissions()`	modifies file access permissions
`filesystem::read_symlink()`	obtains the target of a symbolic link
`filesystem::remove()`, `filesystem::remove_all()`	removes a single file or whole directory recursively with all its content
`filesystem::rename()`	moves or renames a file or directory
`filesystem::resize_file()`	changes the size of a regular file by truncation or zero-fill
`filesystem::space()`	determines available free space on the file system
`filesystem::status()`, `filesystem::symlink_status()`	determines file attributes, determines file attributes, checking the symlink target
`filesystem::temp_directory_path()`	returns a directory suitable for temporary files

Getting & Displaying the File Time

In C++17 there's one thing about `last_write_time()` values that's inconvenient.

We have one free function and a method in `directory_entry`. They both return `file_time_type` which is currently defined as:

```
using file_time_type = std::chrono::time_point</*trivial-clock*/>;
```

From the standard, 30.10.25 Header `<filesystem>` synopsis:

> `trivial-clock` is an implementation-defined type that satisfies the TrivialClock requirements and that is capable of representing and measuring file time values. Implementations should ensure that the resolution and range of `file_time_type` reflect the operating system dependent resolution and range of file time values.

In other words, it's implementation-dependent.

For example, in GCC/Clang STL file time is implemented on top of `chrono::system_clock`, but in MSVC it's a platform-specific clock.

Here are some more details about the implementation decisions in Visual Studio: std::filesystem::file_-time_type does not allow easy conversion to time_t[1]

The situation might soon improve as in C++20 we'll get `std::chrono::file_clock` and also conversion routines between clocks. See P0355[2] (already added into C++20).

Let's have a look at some code.

```
auto filetime = fs::last_write_time(myPath);
const auto toNow = fs::file_time_type::clock::now() - filetime;
const auto elapsedSec = duration_cast<seconds>(toNow).count();
// skipped std::chrono prefix for duration_cast and seconds
```

The above code gives you a way to compute the number of seconds that have elapsed since the last update. This is however, not as useful as showing a real date.

On POSIX (GCC and Clang Implementation) you can easily convert file time to `system_-clock` and then obtain `std::time_t`:

```
auto filetime = fs::last_write_time(myPath);
std::time_t convfiletime = std::chrono::system_clock::to_time_t(filetime);
std::cout << "Updated: " << std::ctime(&convfiletime) << '\n';
```

In MSVC the code won't compile. However there's a guarantee that `file_time_type` is usable with native OS functions that takes `FILETIME`. So we can write the following code to solve the issue:

```
auto filetime = fs::last_write_time(myPath);
FILETIME ft;
memcpy(&ft, &filetime, sizeof(FILETIME));
SYSTEMTIME  stSystemTime;
if (FileTimeToSystemTime(&ft, &stSystemTime)) {
    // use stSystemTime.wYear, stSystemTime.wMonth, stSystemTime.wDay, ...
}
```

See `Chapter Filesystem/filesystem_list_files_info.cpp` for the full sample.

[1]https://developercommunity.visualstudio.com/content/problem/251213/stdfilesystemfile-time-type-does-not-allow-easy-co.html

[2]https://wg21.link/p0355

File Permissions

In the table above you might noticed functions related to file permissions. We have two major functions:

- `std::filesystem::status()` and
- `std::filesystem::permissions()`

The first one returns `file_status` which contains information about the file type and also its permissions.

And you can use the second function to modify the file permissions. For example, to change a file to be read-only.

File permissions - `std::filesystem::perms` - it's an enum class that represents the following values:

Name	Value (octal)	POSIX macro	Notes
none	0000		There are no permissions set for the file
owner_read	0400	`S_IRUSR`	Read permission, owner
owner_write	0200	`S_IWUSR`	Write permission, owner
owner_exec	0100	`S_IXUSR`	Execute/search permission, owner
owner_all	0700	`S_IRWXU`	Read, write, execute/search for owner
group_read	0040	`S_IRGRP`	Read permission, group
group_write	0020	`S_IWGRP`	Write permission, group
group_exec	0010	`S_IXGRP`	Execute/search permission, group
group_all	0070	`S_IRWXG`	Read, write, execute/search by group
others_read	0004	`S_IROTH`	Read permission, others
others_write	0002	`S_IWOTH`	Write permission, others
others_exec	0001	`S_IXOTH`	Execute/search permission, others
others_all	0007	`S_IRWXO`	Read, write, execute/search for others
all	0777		`owner_all | group_all | others_all`
set_uid	04000	`S_ISUID`	Set-user-ID on execution
set_gid	02000	`S_ISGID`	Set-group-ID on execution
sticky_bit	01000	`S_ISVTX`	Operating system dependent
mask	07777		`all | set_uid | set_gid | sticky_bit`
unknown	0xFFFF		The permissions are not known

Here's a short code that demonstrates how to print file permissions:

Chapter Filesystem/filesystem_permissions.cpp

```
std::ostream& operator<< (std::ostream& stream, fs::perms p)
{
    stream << "owner: "
       << ((p & fs::perms::owner_read)  != fs::perms::none ? "r" : "-")
       << ((p & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
       << ((p & fs::perms::owner_exec)  != fs::perms::none ? "x" : "-");
    stream << " group: "
       << ((p & fs::perms::group_read)  != fs::perms::none ? "r" : "-")
       << ((p & fs::perms::group_write) != fs::perms::none ? "w" : "-")
       << ((p & fs::perms::group_exec)  != fs::perms::none ? "x" : "-");
    stream << " others: "
       << ((p & fs::perms::others_read)  != fs::perms::none ? "r" : "-")
       << ((p & fs::perms::others_write) != fs::perms::none ? "w" : "-")
       << ((p & fs::perms::others_exec)  != fs::perms::none ? "x" : "-");
    return stream;
}
```

You can use the above `operator<<` implementation as follows:

```
std::cout << "perms: " << fs::status("myFile.txt").permissions() << '\n';
```

Setting Permissions

To change the permissions you can use the following code:

```
std::cout << "after creation: " << fs::status(sTempName).permissions() << '\n';
fs::permissions(sTempName, fs::perms::owner_read, fs::perm_options::remove);
std::cout << "after change: " << fs::status(sTempName).permissions() << '\n';
```

`std::filesystem::permissions` is a function that takes a path and then a flag and the "action" parameter.

`fs::perm_options` has three modes:

- `replace` - The permissions flag you pass will replace the existing state. It's the default value for this parameter.
- `add` - The permission flag will be bitwise OR-ed with the existing state.

- `remove` - The permissions will be replaced by the bitwise AND of the negated argument and current permissions.
- `nofollow` - The permissions will be changed on the symlink itself, rather than on the file it resolves to

For example:

```
// remove "owner_read"
fs::permissions(myPath, fs::perms::owner_read, fs::perm_options::remove);

// add "owner_read"
fs::permissions(myPath, fs::perms::owner_read, fs::perm_options::add);

// replace and set "owner_all":
fs::permissions(myPath, fs::perms::owner_all); // replace is default param
```

Note for Windows

Windows is not a POSIX system, and it doesn't map POSIX file permissions to its scheme. For `std::filesystem` it only supports two modes: read-only and all.

From Microsoft Docs filesystem documentation[3]:

> The supported values are essentially "readonly" and all. For a readonly file, none of the *_write bits are set. Otherwise, the `all` bit (0777) is set.

Thus, unfortunately, you have limited options if you want to change file permissions on Windows.

Error Handling & File Races

So far, the examples in this chapter have used exception handling to report errors. The filesystem API is also equipped with function and method overloads that outputs an error code. You can decide if you want exceptions or error codes.

[3]https://docs.microsoft.com/en-us/cpp/standard-library/filesystem-enumerations?view=vs-2017

For example we have two overloads for `file_size`:

```
uintmax_t file_size(const path& p);
uintmax_t file_size(const path& p, error_code& ec) noexcept;
```

the second one can be used in the following way:

```
const std::filesystem::path testPath("C:\test.txt");
std::error_code ec{};
auto size = std::filesystem::file_size(testPath, ec);
if (ec == std::error_code{})
    std::cout << "size: " << size << '\n';
else
    std::cout << "error when accessing test file, size is: "
              << size << " message: " << ec.message() << '\n';
```

`file_size` takes an additional output parameter - `error_code` and will set a value if something happens. If the operation is successful, then `ec` will be value initialised.

File Races

It's important to point out the undefined behaviour that might happen when a file race occurs.

From *30.10.2.3 File system race behavior*:

> The behavior is undefined if the calls to functions in this library introduce a file system race.

And from *30.10.9 file system race*:

> The condition that occurs when multiple threads, processes, or computers interleave access and modification of the same object within a file system.

Examples

In this section, we'll analyse a few examples where `std::filesystem` is used. We'll go from a simple case - loading a file into a string, then explore creating directories and then filtering filenames using `std::regex`.

The demo sample is available in `Chapter Filesystem/filesystem_list_files.cpp` and its extended version (that shows file time and size) is located at `Chapter Filesystem/filesystem_list_files_info.cpp`

For more use cases you can also read the chapter - How to Parallelise CSV Reader - where `std::filesystem` is a crucial element for finding CSV files.

Loading a File into a String

The first example shows a compelling case where we leverage `std::filesystem`'s size-related functions to build a buffer for the file contents.

Here's the code:

Chapter Filesystem/filesystem_load_string.cpp

```
[[nodiscard]] std::string GetFileContents(const fs::path& filePath) {
    std::ifstream inFile{ filePath, std::ios::in | std::ios::binary };
    if (!inFile)
        throw std::runtime_error("Cannot open " + filePath.string());

    const auto fsize = fs::file_size(filePath);
    if (fsize > std::numeric_limits<size_t>::max())
        throw std::runtime_error("file is too large to fit into size_t! "
                    + filePath.string());

    std::string str(static_cast<size_t>(fsize), 0);

    inFile.read(str.data(), str.size());
    if (!inFile)
        throw std::runtime_error("Could not read the full contents from "
                    + filePath.string());

    return str;
}
```

Before C++17 to get the file size you'd usually reposition the file pointer to the end and then read the position again. For example:

```
ifstream testFile("test.file", ios::binary);
const  auto begin = myfile.tellg();
testFile.seekg (0, ios::end);
const auto end = testFile.tellg();
const auto fsize =  (end-begin);
```

You could also open a file with `ios::ate` flag and then the file pointer will be positioned automatically at the end.

However, all of the above methods require to open a file but with `std::filesystem` the code is much shorter, and we only have to read file properties.

What's more, the `std::filesystem` technique requires "lower" access rights as you only need parent directory read permission. There's no need to have "file read" permission.

If you use `std::filesystem::directory_entry` method, then it's possible that the file size comes from a cache.

Creating Directories

In the second example, we'll build `N` directories each with `M` files.

The core part of the `main()`:

Chapter Filesystem/filesystem_build_temp.cpp

```
const fs::path startingPath{ argc >= 2 ? argv[1] : fs::current_path() };
const std::string strTempName{ argc >= 3 ? argv[2] : "temp" };
const int numDirectories{ argc >= 4 ? std::stoi(argv[3]) : 4 };
const int numFiles{ argc >= 5 ? std::stoi(argv[4]) : 4 };

if (numDirectories < 0 || numFiles < 0)
    throw std::runtime_error("negative input numbers...");

const fs::path tempPath = startingPath / strTempName;
CreateTempData(tempPath, numDirectories, numFiles);
```

And the `CreateTempData()` function:

Chapter Filesystem/filesystem_build_temp.cpp

```
std::vector<fs::path> GeneratePathNames(const fs::path& tempPath,
                                        unsigned num) {
    std::vector<fs::path> outPaths{ num, tempPath };
    for (auto& dirName : outPaths) {
        // use pointer value to generate unique name...
        const auto addr = reinterpret_cast<uintptr_t>(&dirName);
        dirName /= std::string("tt") + std::to_string(addr);
    }
    return outPaths;
}

void CreateTempFiles(const fs::path& dir, unsigned numFiles) {
    auto files = GeneratePathNames(dir, numFiles);
    for (auto &oneFile : files)
    {
        std::ofstream entry(oneFile.replace_extension(".txt"));
        entry << "Hello World";
    }
}

void CreateTempData(const fs::path& tempPath, unsigned numDirectories,
                    unsigned numFiles) {
    fs::create_directory(tempPath);
    auto dirPaths = GeneratePathNames(tempPath, numDirectories);

    for (auto& dir : dirPaths) {
        if (fs::create_directory(dir))
            CreateTempFiles(dir, numFiles);
    }
}
```

In `CreateTempData()` we first create the root of our folder structure. Then we generate path names in `GeneratePathNames()`. Each pathname is built from a pointer address. Such an approach should give us a good selection of unique names. When we have a vector of unique paths, then we also generate another vector of paths for files. In this case, we use the `.txt` extension. While a directory is created using `fs::create_directory`, to create files we can use standard stream objects.

If we run the application with the following parameters: "`. temp 2 4`" it will create the following directory structure:

```
temp
   tt22325368
      tt22283456.txt, size 11 bytes
      tt22283484.txt, size 11 bytes
      tt22283512.txt, size 11 bytes
      tt22283540.txt, size 11 bytes
   tt22325396
      tt22283456.txt, size 11 bytes
      tt22283484.txt, size 11 bytes
      tt22283512.txt, size 11 bytes
      tt22283540.txt, size 11 bytes
```

Filtering Files Using Regex

The last example in this chapter will filter file names with the addition of `std::regex`, which has been available since C++11.

The core of the `main()`:

Chapter Filesystem/filesystem_filter_files.cpp

```cpp
const fs::path pathToShow{ argc >= 2 ? argv[1] : fs::current_path() };
const std::regex reg(argc >= 3 ? argv[2] : "");

auto files = CollectFiles(pathToShow);

std::sort(files.begin(), files.end());

for (auto& entry : files) {
    const auto strFileName = entry.mPath.filename().string();
    if (std::regex_match(strFileName, reg))
        std::cout << strFileName << "\tsize: " << entry.mSize << '\n';
}
```

The application collects all the files from a given directory (recursively). Later a file entry is shown if it matches regex.

The `CollectFiles()` function uses recursive directory iterator to find all the files, and it also builds necessary information about each file. In `main()` we sort each file by size.

Chapter Filesystem/filesystem_filter_files.cpp

```
struct FileEntry {
    fs::path mPath;
    uintmax_t mSize{ 0 };

    static FileEntry Create(const fs::path& filePath) {
        return FileEntry{ filePath, fs::file_size(filePath) };
    }

    friend bool operator < (const FileEntry& a, const FileEntry& b) noexcept {
        return a.mSize < b.mSize;
    }
};

std::vector<FileEntry> CollectFiles(const fs::path& inPath) {
    std::vector<fs::path> paths;
    if (fs::exists(inPath) && fs::is_directory(inPath)) {
        std::filesystem::recursive_directory_iterator dirpos{ inPath };

        std::copy_if(begin(dirpos), end(dirpos), std::back_inserter(paths),
            [](const fs::directory_entry& entry) {
                return entry.is_regular_file();
            }
        );
    }
    std::vector<FileEntry> files(paths.size());
    std::transform(paths.cbegin(), paths.cend(), files.begin(), FileEntry::Create);
    return files;
}
```

In `CollectFiles` we use a recursive iterator and then `std::copy_if` to filter only regular files. Later, once the files are collected, we create the output vector of File Entries. `FileEntry::Create()` initialises objects and also fetches the size of a file.

For example, if we run the application with the following parameters "`temp .*.txt`" we'll be looking for all `txt` files in a directory.

```
.\FilterFiles.exe temp .*.txt
tt22283456.txt  size: 11
tt22283484.txt  size: 11
tt22283512.txt  size: 11
tt22283540.txt  size: 11
tt22283456.txt  size: 11
tt22283484.txt  size: 11
tt22283512.txt  size: 11
tt22283540.txt  size: 11
```

Optimisation & Code Cleanup

The `CollectFiles()` function iterates over a directory and then outputs regular file's paths into a vector. Later the function creates `FileEntry` objects and returns them into a separate vector.

It appears that we don't leverage all the benefits of `filesystem::directory_entry` objects that we have during the scan. For example, the `directory_entry::file_size` member method will be much faster than a free function `filesystem::file_size` because `directory_entry` usually keeps file attributes in cache.

Another element to optimise is a temporary vector of paths. We can skip it by using range-based for loop.

Here's the final code:

Chapter Filesystem/filesystem_filter_files.cpp

```
std::vector<FileEntry> CollectFilesOpt(const fs::path& inPath) {
    std::vector<FileEntry> files;
    if (fs::exists(inPath) && fs::is_directory(inPath)) {
        for (const auto& entry : fs::recursive_directory_iterator{ inPath }) {
            if (entry.is_regular_file())
                files.push_back({ entry, entry.file_size() });
        }
    }
    return files;
}
```

Chapter Summary

In this chapter, we dove into one of the most significant additions of C++17: `std::filesystem`. You saw the core elements of the library: the `path` class, `directory_entry` and iterators and lots of supporting free functions.

Throughout the chapter, we also explored lots of examples: from simple cases like composing a path, getting file size, iterating through directories to even more complex: filtering with regex or creating temp directory structures.

You should be equipped with solid knowledge about `std::filesystem` and be prepared to explore the library on your own.

The full implementation of `std::filesystem` is described in the paper P0218: Adopt the File System TS for C++17[4]. There are also others updates like P0317: Directory Entry Caching[5], P0430 – File system library on non-POSIX-like operating systems[6], P0492R2 - Resolution of C++17 National Body Comments[7], P0392 -Adapting string_view by filesystem paths[8]

You can find the final specification in C++17 draft - N4687[9]: the "filesystem" section, 30.10. Or under this online location timsong-cpp/filesystems[10].

[4]https://wg21.link/p0218
[5]https://wg21.link/p0317
[6]https://wg21.link/p0430
[7]https://wg21.link/p0492
[8]https://wg21.link/p0392r0
[9]https://wg21.link/n4687
[10]https://timsong-cpp.github.io/cppwp/n4659/filesystems

Compiler Support

GCC/libstdc++

The library was added in the version 8.0, see commit - Implement C++17 Filesystem[11]. Since GCC 5.3, you can play with the experimental version - the TS implementation.

Starting with GCC 9.1 the filesystem library is located in the same binary as the rest of The Standard Library, but before that release, you have to link with `-lstdc++fs`.

To compile `demo.cpp` you should write the following command:

```
// GCC 9.1 and up:
g++ -std=c++17 -O2 -Wall -Werror demo.cpp
// before GCC 9.1:
g++ -std=c++17 -O2 -Wall -Werror demo.cpp -lstdc++fs
```

Clang/libc++

The support for `<filesystem>` was implemented in version 7.0, you can see this commit[12]. Since Clang 3.9, you can start playing with the experiential version, TS implementation.

Similarly to GCC (before GCC 9.1), you have to link to `libc++fs.a`.

Visual Studio

The full implementation of `<filesystem>` was added in Visual Studio 2017 15.7.

Before 15.7, you could play with `<experimental/filesystem>` in a much earlier version. The experimental implementation was available even in Visual Studio 2012, and later it was gradually improved with each release.

Compiler Support Summary

Feature	GCC	Clang	MSVC
Filesystem	8.0	7.0	VS 2017 15.7

[11]https://github.com/gcc-mirror/gcc/commit/3b90ed62fb848046ed7ddef07df7c806e7f3fadb
[12]https://github.com/llvm-mirror/libcxx/commit/a0866c5fb5020d15c69deda94d32a7f982b88dc9

14. Parallel STL Algorithms

Concurrency and Parallelism are core aspects of any modern programming language. Before C++11, there was no standard support in the language for threading - you could use third-party libraries or System APIs. Modern C++ started to bring more and more necessary features: threads, atomics, locks, `std::async` and futures.

C++17 gives us a way to parallelise most of the standard library algorithms. With a robust and yet straightforward abstraction layer, you can leverage more computing power out of a machine.

In this chapter, you'll learn:

- What's on the way for C++ regarding parallelism and concurrency
- Why `std::thread` is not enough
- What the execution policies are
- How to run parallel algorithms
- Which algorithms were parallelised
- What the new algorithms are
- Expected performance of parallel execution
- Examples of parallel execution and benchmarks

Introduction

If we look at the computers that surround us, we can observe that most of them are multi-processors units. Even mobile phones have four or even eight cores. Not to mention graphics cards that are equipped with hundreds (or even thousands) of small computing cores.

The trend towards multicore machines was summarised perfectly in a famous article by Herb Sutter The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software[1]. The article appeared in 2006.

A glance around is all it takes to see that this trend is not slowing down.

While for a simple application there's probably no need to use the full computing capacity of your machine, there are applications which do require just that. Gaming, fast, responsive apps, graphics processing, video/music editing, data processing, financial, servers and many more types of systems. Spawning threads on a CPU and processing tasks concurrently is one way to achieve that.

With C++11/14 we've finally got threading into the Standard Library. You can now create instances of `std::thread` and not just depend on third-party libraries or a system API. What's more, there's also async processing with futures (`std::async`).

Multithreading is a significant aspect of modern C++. In the C++ Committee, there's a separate group - "SG1, Concurrency" that works on bringing more features like this to the standard.

What's on the way?

- Coroutines
- Atomic Smart pointers
- Transactional Memory
- Barriers
- Tasks blocks
- Parallelism
- Compute
- Executors
- Heterogeneous programming models support

[1]http://www.gotw.ca/publications/concurrency-ddj.htm

As you can see, the plan is to expose as much of your machine's computing power as possible, directly from The Standard Library.

Not Only Threads

As mentioned earlier, using threads is not the only way of leveraging the power of your machine.

If your system has 8 cores in the CPU then you can use 8 threads and assuming you can split your work into separate chunks then you can hypothetically process your tasks several times faster than on a single thread.

But there's a chance to speed up things even more!

So where's the rest of the power coming from?

Vector Instructions from CPU and GPU computing.

The first element - vector instructions - allows you to compute several components of an array in a single instruction. It's also called SIMD - Single Instruction Multiple Data. Most of CPUs have 128-bit wide registers, and recent chips contain registers even 256 or 512 bits wide (AVX 256, AVX 512).

For example, using AVX-512 instructions, you can operate on 16 integer values (32-bit) at the same time!

The second element is the GPU. It might contain hundreds of smaller cores.

There are third-party APIs that allow you to access GPU/vectorisation: for example, we have CUDA, OpenCL, OpenGL, Intel TBB, OpenMP and many more. There's even a chance that your compiler will try to auto-vectorise some of the code. Still, we'd like to have such support directly from the Standard Library. That way the same code can be used on many platforms.

C++17 moves us a bit into that direction and allows us to use more computing power: it unlocks the auto vectorisation/auto parallelisation feature for algorithms in The Standard Library.

Overview

The new feature of C++17 looks surprisingly simple from a user point of view. You have a new template parameter that can be passed to most of the standard algorithms: this new argument is called **execution policy**.

```
template< class ExecutionPolicy, class RandomIt, ... >
std::algorithm_name(ExecutionPolicy&& policy, RandomIt first, RandomIt last, ...);
```

We'll go into the details later, but the general idea is that you call an algorithm and then you specify **how** it can be executed. Can it be parallel or just serial.

For example:

```
std::vector<int> v = genLargeVector();
// sort a vector using a parallel policy
std::sort(std::execution::par, v.begin(), v.end());
```

The above example will sort a vector in parallel - as specified by the first argument `std::execution::par`. The whole machinery is hidden from a user perspective. It's up to the STL implementation to choose the best approach to run tasks in parallel. Usually, they might leverage thread pools.

The hint - the execution policy parameter - is necessary because the compiler cannot deduce everything from the code. You, as the author of the code, only know if there are any side effects, possible race conditions, deadlocks, or if there's no sense in the running in parallel (such as if you have a small collection of items).

C++17's Parallelism comes from the Technical Specification that was published officially in 2015. The whole project of bringing parallel algorithms into C++ took more than five years - it started in 2012 and was merged into the standard in 2017. See the paper: P0024 - The Parallelism TS Should be Standardised[2].

Execution Policies

The execution policy parameter suggests how the algorithm should be executed.

[2]https://wg21.link/P0024

We have the following options:

Policy Name	Description
`sequenced_policy`	It is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and require that a parallel algorithm's execution not be parallelised.
`parallel_policy`	It is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelised.
`parallel_unsequenced_policy`	It is an execution policy type used as a unique type to disambiguate parallel algorithm overloading and indicate that a parallel algorithm's execution may be parallelised and vectorised.

We have also three global objects corresponding to each execution policy type:

- `std::execution::seq`
- `std::execution::par`
- `std::execution::par_unseq`

Please note that execution policies are unique types, with their corresponding global objects. They are not enumerations, nor do they share the same base type.

Execution policy declarations and global objects are located in the `<execution>` header.

Understanding Execution Policies

To understand the difference between execution policies, let's try to build a model of how an algorithm might work.

Consider a simple vector of `float` values. In the below example, each element of the vector is multiplied by 2 and then the result is stored into an output container:

std::transform on a vector

```
std::vector<float> vecX = {...};  // generate
std::vector<float> vecY(vecX.size());

std::transform(
    std::execution::seq,
    begin(vecX), end(vecX),            // input range
    begin(vecY),                       // output
    [](float x) { return x * 2.0f; }); // operation
```

Here's a pseudo-code for sequential execution of the above algorithm:

```
operation
{
    load vecX[i] into RegisterX
    multiply RegisterX by 2.0f
    store RegisterX into vecY[i]
}
```

In the sequential execution, we'll access one element (from `vecX`), perform an operation and then store the result into the output (`vecY`). Execution for all elements happens on a single thread (on the calling thread).

With the `par` policy, the whole `operation` for the i-th element will be executed on one thread. But there may be many threads that process different elements of the container. For example, if you have 8 free threads in the system, 8 elements of the container might be computed at the same time. The element access order is unspecified.

The Standard Library implementations might usually leverage some thread-pool to execute a parallel algorithm. The pool holds some worker threads (generally as many as system cores count), and then a scheduler will divide the input into chunks and assign them into the worker threads[3]. In theory, on a CPU, you could also create as many threads as elements in your container, but due to context switching that wouldn't give you good performance. On the other hand, implementations that use GPUs might provide hundreds of smaller "cores", in that scenario, the scheduler might work entirely differently.

The third execution policy `par_unseq` is an extension of the parallel execution policy. The operation for the i-th element will be performed on a separate thread, but also instructions of that operation might be interleaved and vectorised.

[3]You might watch this great interview with Pedro Teixeira about thread pools in Windows - Inside Windows 8: Pedro Teixeira - Thread pools | Channel 9. This is what MSVC implementation is using.

For example:

```
operation
{
    load vecX[i...i+3] into RegisterXYZW
    multiply RegisterXYZW by 2 // 4 elements at once!
    store RegisterXYZW into vecY[i...i+3]
}
```

The above pseudo-code uses `RegisterXYZW` to represent a wide register that could store 4 elements of the container. For example, in SSE (Streaming SIMD Extensions) you have 128-bit registers that can handle 4 32-bit values, like 4 integers or 4 floating-point numbers (or can store 8 16-bit values). However, such vectorisation might be extended to even larger registers like with AVX where you have 256 or even 512-bit registers. It's up to the implementation of the Standard Library to choose the best vectorisation scheme.

In this case, each instruction of the operation is "duplicated" and interleaved with others. That way the compiler can generate instructions that will handle several elements of the container at the same time.

In theory, if you have 8 free system threads, with 128-bit SIMD registers, and we process float values (32-bit values) - then, we can compute 8*4 = 32 values at once!

Why do you need the sequential policy?

Most of the time you'll be probably interested in using parallel policy or parallel unsequenced one. But for debugging it might be easier to use `std::execution::seq`. The parameter is also quite convenient as you might easily switch between the execution model using a template parameter. For some algorithms, the sequential policy might also give better performance than the C++14 counterpart. Read more in the Benchmark section.

Limitations and Unsafe Instructions

The whole point of execution policies is to parallelise standard algorithms effortlessly. Nevertheless, there are some limitations you need to be aware of.

For example, with `std::par` if you want to modify a shared resource, you need to use some synchronisation mechanism to prevent data races and deadlocks[4]:

Locking inside a parallel operation

```
std::vector<int> vec(1000);
std::iota(vec.begin(), vec.end(), 0);
std::vector<int> output;
std::mutex m;
std::for_each(std::execution::par, vec.begin(), vec.end(),
[&output, &m, &x](int& elem) {
    if (elem % 2 == 0) {
        std::lock_guard guard(m);
        output.push_back(elem);
    }
});
```

The above code filters out the input vector and then puts the elements in the output container.

If you forget about using a mutex (or another form of synchronisation), then `push_back` might cause **data races** - as multiple threads might try to add a new element to the vector at the same time.

The above example will also demonstrate weak performance, as using too many synchronisation points kills the parallel execution.

When using `par` execution policy try to access the shared resources as little as possible.

With `par_unseq` function invocations might be interleaved, so it's forbidden to use unsafe vectorised code. For example, using mutexes or memory allocation might lead to data races and deadlocks.

[4]When you only want to read a shared resource, then there's no need to synchronise.

Unsafe Instructions in Par Unseq Execution

```
std::vector<int> vec = GenerateData();
std::mutex m;
int x = 0;
std::for_each(std::execution::par_unseq, vec.begin(), vec.end(),
[&m, &x](int& elem) {
    std::lock_guard guard(m);
    elem = x;
    x++; // increment a shared value
});
```

Since the instructions might be interleaved on one thread, you may end up with the following sequence of actions:

```
std::lock_guard guard(m) // for i-th element
std::lock_guard guard(m) // for i+1-th element
...
```

As you can see, two locks (in the same mutex) will happen on a single thread causing a deadlock!

Don't use synchronisation and memory allocation when executing with `par_unseq` policy.

Exceptions

When using execution policies, you need to be prepared for two kinds of situations.

- the scheduler or the implementation fails to allocate resources for the invocation - then `std::bad_alloc` is thrown.
- an exception is thrown from the user code (a functor) - in that case, the exception is not re-thrown, `std::terminate()` is called.

See the example below.

Chapter Parallel Algorithms/par_exceptions.cpp

```
try {
    std::vector<int> v{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    std::for_each(std::execution::par, v.begin(), v.end(),
        [](int& i) {
            std::cout << i << '\n';

            if (i == 5)
                throw std::runtime_error("something wrong... !");
        });
}
catch (const std::bad_alloc& e) {
    std::cout << "Error in execution: " << e.what() << '\n';
}
catch (const std::exception& e) { // will not happen
    std::cout << e.what() << '\n';
}
catch (...) {
    std::cout << "error!\n";
}
```

If you run the above code, the `catch` section will only handle `std::bad_alloc`. And if you exit a lambda because of some exception, then the `std::terminate` will be called. The exceptions are not re-thrown.

When you use parallel algorithms, for better error handling, try to make your functors `noexcept`.

Algorithm Update

The execution policy parameter was added to most of the existing algorithms.

Here's the list of new overloads for the algorithms:

`adjacent_difference`	`inplace_merge`	`replace_copy`
`adjacent_find`	`is_heap`	`replace_copy_if`
`all_of`	`is_heap_until`	`replace_if`
`any_of`	`is_partitioned`	`reverse`
`copy`	`is_sorted`	`reverse_copy`
`copy_if`	`is_sorted`	`rotate`
`copy_n`	`is_sorted_until`	`rotate_copy`
`count`	`lexicographical_compare`	`search`
`count_if`	`max_element`	`search_n`
`equal`	`merge`	`set_difference`
`exclusive_scan`	`min_element`	`set_intersection`
`fill`	`minmax_element`	`set_symmetric_difference`
`fill_n`	`mismatch`	`set_union`
`find`	`move`	`sort`
`find_end`	`none_of`	`stable_partition`
`find_first_of`	`nth_element`	`stable_sort`
`find_if`	`partial_sort`	`swap_ranges`
`find_if_not`	`partial_sort_copy`	`transform`
`for_each`	`partition`	`transform_exclusive_scan`
`for_each_n`	`partition_copy`	`transform_inclusive_scan`
`generate`	`remove`	`transform_reduce`
`generate_n`	`remove_copy`	`uninitialized_copy`
`includes`	`remove_copy_if`	`uninitialized_copy_n`
`inclusive_scan`	`remove_if`	`uninitialized_fill`
`inner_product`	`replace`	`uninitialized_fill_n`
	`unique`	`unique_copy`

New Algorithms

To fully support new parallel execution patterns The Standard Library was also equipped with a set of new algorithms:

Algorithm	Description
`for_each`	similar to `for_each` except returns `void`
`for_each_n`	applies a function object to the first `n` elements of a sequence
`reduce`	similar to `accumulate`, except out of order execution to allow parallelism
`transform_reduce`	transforms the input elements using a unary operation, then reduces the output out of order
`exclusive_scan`	parallel version of `partial_sum`, excludes the i-th input element from the i-th sum, out of order execution to allow parallelism
`inclusive_scan`	parallel version of `partial_sum`, includes the i-th input element in the i-th sum, out of order execution to allow parallelism
`transform_exclusive_scan`	applies a functor, then calculates exclusive scan
`transform_inclusive_scan`	applies a functor, then calculates inclusive scan

The new algorithms form three groups: `for_each`, `reduce` and then `scan`, plus their alternatives.

With `reduce` and `scan` you also get "fused" versions like `transform_reduce`. These compositions should give you much better performance than using two separate steps - because the cost of parallel execution setup is smaller and also you have one loop traversal less.

The new algorithms also provide overloads without the execution policy parameter so that you can use them in a standard serial version.

Below you'll find a description of each group.

For Each Algorithm

In the serial version of `for_each`, the version that was available before C++17 you get a unary function as a return value from the algorithm.

Returning such an object is not possible in a parallel version, as the order of invocations is indeterminate.

Here's a basic example:

Chapter Parallel Algortihms/par_basic.cpp

```
std::vector<int> v(100);
std::iota(v.begin(), v.end(), 0);

std::for_each(std::execution::par, v.begin(), v.end(),
    [](int& i) { i += 10; });

std::for_each_n(std::execution::par, v.begin(), v.size()/2,
    [](int& i) { i += 10; });
```

The first `for_each` algorithm will update all of the elements of a vector, while the second execution will work only on the first half of the container.

Understanding Reduce Algorithms

Another core algorithm that is available with C++17 is `std::reduce`. This new algorithm provides a parallel version of `std::accumulate`. But it's important to understand the difference.

`std::accumulate` returns the sum of all the elements in a range (or a result of a binary operation that can be different than just a sum).

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto sum = std::accumulate(v.begin(), v.end(), /*init*/0);
// sum is 55
```

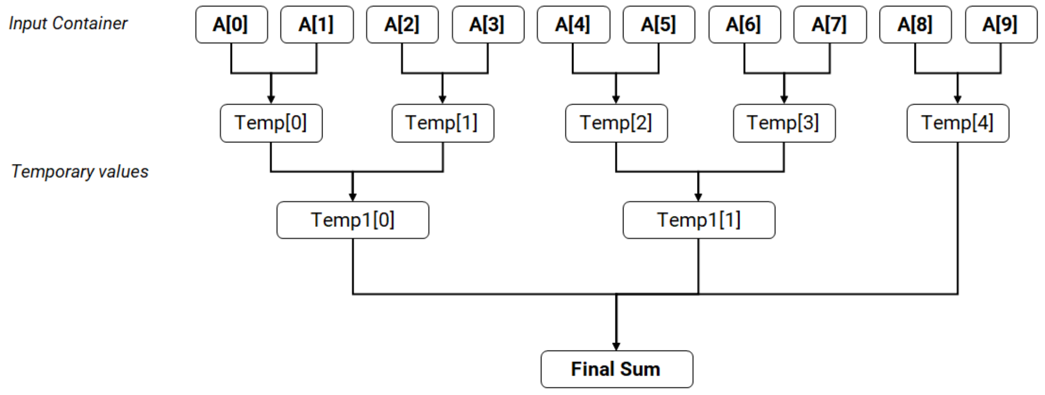
The algorithm is sequential and performs "left fold", which means it will accumulate elements from the start to the end of a container.

The above example can be expanded into the following code:

```
sum = init +
      v[0] + v[1] + v[2] +
      v[3] + v[4] + v[5] +
      v[6] + v[7] + v[8] + v[9];
```

The parallel version - `std::reduce` - computes the final sum using a tree approach (sum sub-ranges, then merge the results, divide and conquer). This method can invoke the binary operation/sum in a non-deterministic order. Thus if `binary_op` is not associative or not commutative, the behaviour is also non-deterministic.

Here's a simplified picture that illustrates how a sum of 10 elements might work in a parallel way:



Parallel Sum Example

The above example with accumulate can be rewritten into `reduce`:

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto sum = std::reduce(std::execution::par, v.begin(), v.end(), 0);
```

By default `std::plus<>{}` is used to compute the reduction steps.

A little explanation about associative and commutative operations:

A binary operation `@` on a set `S` is **associative** if the following equation holds for all `x`, `y`, and `z` in `S`:

`(x @ y) @ z = x @ (y @ z)`

An operation is **commutative** if:

`x @ y = y @ x`

For example, we'll get the same results for accumulate and reduce for a vector of integers (when doing a sum), but we might get a slight difference for a vector of floats or doubles. That's because floating-point sum operation is not associative.

An example:

```
// #include <limits> - for numeric_limits
std::cout.precision(std::numeric_limits<double>::max_digits10);
std::cout << (0.1+0.2)+0.3 << " != " << 0.1+(0.2+0.3) << '\n';
```

The output:

```
0.60000000000000009 != 0.59999999999999998
```

Another example might be the operation type: `plus`, for integer numbers, is associative and commutative, but `minus` is not associative nor commutative:

```
1+(2+3) == (1+2)+3 // sum is associative
1+8     == 8+1     // sum is commutative

1-(5-4) != (1-5)-4 // subtraction is not associative
1-7     != 7-1     // subtraction is not commutative
```

`transform_reduce` - Fused Algorithm

To get even more flexibility and performance, the `reduce` algorithm also has a version where you can apply a transform operation before performing the reduction.

Fused Algorithm - Transform and then Reduce

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
auto sumTransformed = std::transform_reduce(std::execution::par,
    v.begin(),
    v.end(),
    0,
    std::plus<int>{},
    [](const int& i) { return i * 2; }
);

// sum is 110
```

The above code will first execute the unary functor - the lambda that doubles the input value; then the results will be reduced into a single sum.

The fused version will be faster than using two algorithms: `std::reduce` firstly and then `std::reduce` - because the implementation will need to perform the parallel execution setup only once.

Scan Algorithms

The third group of new algorithms is `scan`. They implement a version of partial sum, but out of order.

The exclusive scan does not include the i-th element in the output i-th sum, while inclusive scan does.

For example for `array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`

We'll get the following values for partials sums:

Name										
Index	0	1	2	3	4	5	6	7	8	9
Values	1	2	3	4	5	6	7	8	9	10
Exclusive partial sums	0	1	3	6	10	15	21	28	36	45
Inclusive partial sums	1	3	6	10	15	21	28	36	45	55

Similarly to `std::reduce` the order of the operations is unsequenced, thus to get deterministic results, the `binary_op` must be associative.

`scan` has also two fused algorithms: `transform_exclusive_scan` and `transform_-inclusive_scan`. Both of the algorithms will perform a unary operation on the input container, and then they will compute the prefix sums on the output.

Prefix sums have an essential role in many applications, for example for stream compaction, computing summed-area tables or radix sort. Here's a link to an article that describes the algorithms in detail: GPU Gems 3 - Chapter 39. Parallel Prefix Sum (Scan) with CUDA[5].

Performance of Parallel Algorithms

Parallel algorithms are a robust abstraction layer. Although they're relatively easy to use, assessing their performance is less straightforward.

The first point to note is that a parallel algorithm will generally do more work than the sequential version. That's because the algorithm needs to set up and arrange the threading subsystem to run the tasks.

For example, if you invoke `std::transform` on a vector of 100k elements, then the STL implementation needs to divide your vector into chunks and then schedule each chunk to be executed appropriately. If necessary, the implementation might even copy the elements

[5]https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch39.html

before the execution. If you have a system with 10 free threads, the 100k element vector might be divided into chunks of 10k, and then each chunk is transformed by one thread. Due to the setup cost and other limitations of parallel code, the whole execution time won't be 10x faster than the serial version.

The **second** factor that plays an important role is synchronisation. If your operations need to synchronise on some shared objects, then the parallel execution performance decreases. A parallel algorithm performs best when you execute separate tasks.

The **third** element that has a big effect on the execution is memory throughput. If we look at a common desktop CPU, we can see that all cores share the same memory bus. That's why if your instructions wait for the data to be fetched from memory, then the performance improvement over the sequential won't be visible, as all cores will synchronise on the memory access. Algorithms like `std::copy`, `std::reverse` might be even slower than their serial versions - at least on common PC hardware[6]. It's best when your parallel tasks use CPU cycles for computation rather than waiting for memory access.

The **fourth** important thing is that the algorithms are very implementation-dependent. They might use different techniques to achieve parallelism. Not to mention the device that an algorithm might be executed on - on CPU or GPU.

Right now (as of July 2019) there are two implementations available in a popular compiler - starting with Visual Studio 2017 and GCC 9.1.

The Visual Studio Implementation is based on thread pools from Windows and only supports execution on the CPU and skips the vectorisation execution policy [7].

In GCC 9.1, we got parallel algorithms that are based on a popular Intel implementation - PSTL. Intel offered the implementation, update the licence so that the code could be reused in the Standard Library. Internally it requires OpenMP 4.0[8] support and Intel TBB[9] to be linked with the application.

Another thing is the GPU support. With hundreds of smaller computing cores, the algorithms might perform faster than their CPU version. It's important to remember that before executing something on the GPU, you have to usually copy the data to memory visible to the GPU (unless it's a shared memory like in integrated Graphics Cards). And sometimes the cost of the data transfer might reduce the total performance.

[6]See section "Current Limitations of the MSVC Implementation of Parallel Algorithms" in Using C++17 Parallel Algorithms for Better Performance | Visual C++ Team Blog where parallel `std::reverse` appeared to be 1.6 times slower.

[7]The VS implementation will usually process elements in blocks, so there's a chance that auto-vectorisation might still optimise the code and allow vector instruction for better performance.

[8]https://www.openmp.org/

[9]https://www.threadingbuildingblocks.org/

If you decide to use parallel algorithms, it's best to measure the performance against the sequential version. Sometimes, especially for a smaller number of elements, the performance might be even slower.

Examples

Until now, you've seen introductory code samples with parallel algorithms. This section will introduce a few more examples with more complex scenarios.

- You'll see a few benchmarks and see the performance gains over the sequential version.
- We'll discuss an example of how to process several containers in the same parallel algorithm.
- There will also be a sample of how to implement a parallel version of counting elements.

Benchmark

Finally, we can see the performance of the new algorithms.

Let's have a look at an example where we execute separate tasks on each element - using `std::transform`. In that example, the speed up vs the sequential version should be more visible.

Chapter Parallel Algorithms/par_benchmark.cpp

```cpp
#include <algorithm>
#include <execution>
#include <iostream>
#include <numeric>
#include <cmath>
#include "simpleperf.h"

int main(int argc, const char* argv[]) {
    const size_t vecSize = argc > 1 ? atoi(argv[1]) : 6000000;
    std::cout << vecSize << '\n';
    std::vector<double> vec(vecSize, 0.5);
    std::vector out(vec);
    RunAndMeasure("std::transform seq", [&vec, &out] {
        std::transform(std::execution::seq, vec.begin(), vec.end(), out.begin(),
            [](double v) {
                return std::sin(v)*std::cos(v);
            }
        );
        return out.size();
    });

    RunAndMeasure("std::transform par", [&vec, &out] {
        std::transform(std::execution::par, vec.begin(), vec.end(), out.begin(),
            [](double v) {
                return std::sin(v)*std::cos(v);
            }
        );
        return out.size();
    });

    return 0;
}
```

The code calculates `sin*cos` [10] and stores the result in the output vector. Those trigonometry functions will keep CPU busy with arithmetic instructions, rather than just fetching an element from memory.

The application was run on two machines and three modes:

- i7 4720H VS - means Win 10 64bit, i7 4720H, 2.60 GHz base frequency, 4 Cores/8 Threads, MSVC 2017 15.8, Release mode, x86.
- i7 8700 VS- means Win 10 64bit, i7 8700, 3,2 GHz base frequency, 6 Cores/12 Threads, MSVC 2017 15.8, Release Mode, x86.
- i7 8700 GCC - means NixOS 19.03 64bit, i7 8700, 3,2 GHz base frequency, 6 Cores/12 Threads, GCC 9.1, Intel TBB

`RunAndMeasure` is a helper function that runs a function and then prints the timings. The result is used later so that the compiler doesn't optimise the variable away:

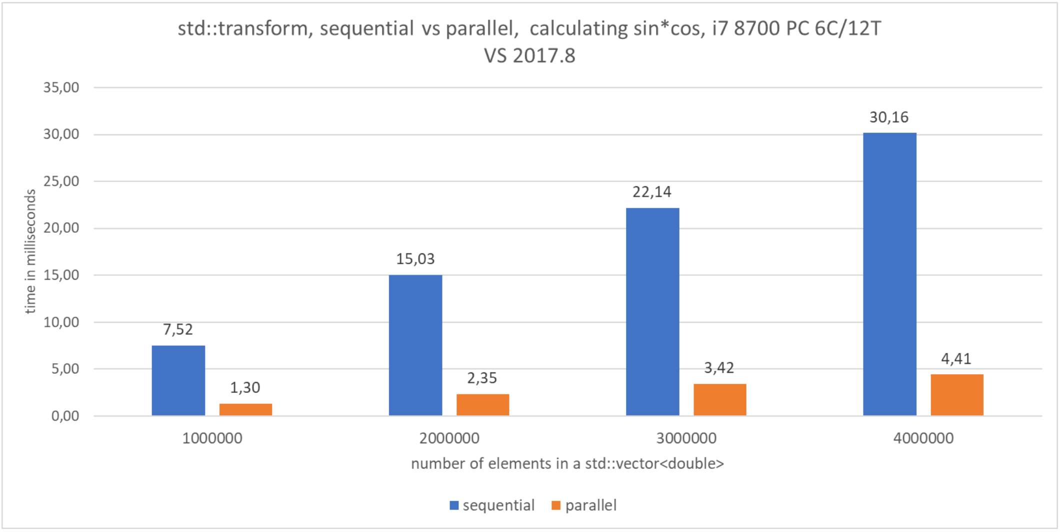
[10] You can also use more optimal computation of `sin*cos` as `sin(2x) = 2 sin(x) cos(x)`.

Helper Function that Measures Execution Time

```
template <typename TFunc> void RunAndMeasure(const char* title, TFunc func) {
    const auto start = std::chrono::steady_clock::now();
    ret = func();
    const auto end = std::chrono::steady_clock::now();
    std::cout << title
         << ": " << std::chrono::duration<double, std::milli>(end - start).count()
         << " ms " << ret << '\n';
}
```

Here are the results (time in milliseconds):

algorithm	vector size	i7 4720H VS	i7 8700 VS	i7 8700 GCC
`std::transform, seq`	1000000	10.9347	7.51991	19.8189
`std::transform, par`	1000000	2.67921	1.30245	3.14286
`std::transform, seq`	2000000	21.8466	15.028	37.3226
`std::transform, par`	2000000	5.29644	2.34634	6.22417
`std::transform, seq`	3000000	32.7403	22.1449	55.8141
`std::transform, par`	3000000	7.79366	3.42295	9.34034
`std::transform, seq`	4000000	44.2565	30.1643	74.2437
`std::transform, par`	4000000	11.7558	4.40974	12.4206



Benchmark of std::transform

The example above might be the perfect case for a parallelisation: we have an operation that requires a decent amount of instructions (trigonometry functions), and then all the tasks are separate. In this case, on a machine with 6 cores and 12 threads, the performance is almost 7X faster! On a computer with 4 cores and 8 threads, the performance is 4.2X faster.

GCC results are surprisingly slower than the Visual Studio version[11]. Still, we can also notice that with the parallel execution we get even 8x improvement on 6 cores/12 threads over the sequential execution.

It's also worth to notice that when the transformation instructions are simple like `return v*2.0` then the performance speed-up might be not seen. This is because all the code will be just waiting on the global memory, and it might perform the same as the sequential version.

Below there's a benchmark of computing the sum of all elements in a vector:

Chapter Parallel Algorithms/par_benchmark.cpp

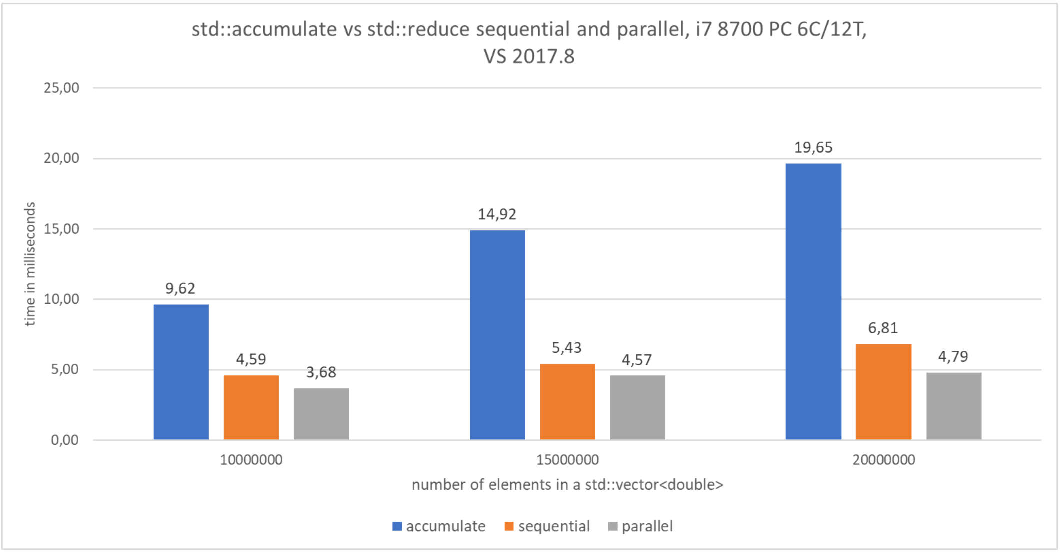
```
#include <algorithm>
#include <execution>
#include <iostream>
#include <numeric>
#include "simpleperf.h"

int main(int argc, const char* argv[]) {
    const size_t vecSize = argc > 1 ? atoi(argv[1]) : 6000000;
    std::cout << vecSize << '\n';
    std::vector<double> vec(vecSize, 0.5);

    RunAndMeasure("std::accumulate", [&vec] {
        return std::accumulate(vec.begin(), vec.end(), 0.0);
    });

    RunAndMeasure("std::reduce, seq", [&vec] {
            return std::reduce(std::execution::seq,
                vec.begin(), vec.end(), 0.0);
        }
    );

    RunAndMeasure("std::reduce, par", [&vec] {
            return std::reduce(std::execution::par,
                vec.begin(), vec.end(), 0.0);
        }
    );
```

[11] A similar machine was used, but the results were 2x slower. The full investigation is outside the scope of the book.

```
    return 0;
}
```

Here are the results:

algorithm	size	i7 4720H VS	i7 8700 VS	i7 8700 GCC
`std::accumulate`	10000000	10.5814	9.62405	9.65569
`std::reduce seq`	10000000	6.9556	4.58746	9.20017
`std::reduce par`	10000000	4.88708	3.67831	2.45625
`std::accumulate`	15000000	17.8769	14.9163	14.2885
`std::reduce seq`	15000000	11.5103	5.42508	13.7725
`std::reduce par`	15000000	9.99877	4.5679	3.79334
`std::accumulate`	20000000	21.8888	19.6507	18.8786
`std::reduce seq`	20000000	16.2142	6.80581	18.4035
`std::reduce par`	20000000	10.8826	4.79214	5.141



Benchmark std::accumulate vs std::reduce

During this execution, the `par` version was 2x..4x faster than the standard `std::accumulate`!

When looking at `par` and `accumulate`, this time, GCC results are almost the same as Visual Studio. It's also clear that the GCC version switches to regular `std::accumulate` when you use sequential mode for `std::reduce`.

Another reason to use sequential policy?

In Visual Studio the sequential version of `std::reduce` was also faster than `std::accumulate`. This might happen because in `std::reduce` the order of operations is not determined, while `std::accumulate` is a left fold. The compiler has more options to optimise the code.

Processing Several Containers At the Same Time

When using parallel algorithms, you might sometimes want to access other containers. For example, you might want to execute `for_each` on two containers.

The main technique is to get the index of the element currently being processed. Then you can use that index to access other containers (assuming the containers are of the same size).

We can do it in a few ways:

- by using a separate container of indices
- by using zip iterators/wrappers

Let's have a look at the techniques:

Separate Container of Indices

Chapter Parallel Algorithms/par_iterating_multiple.cpp

```
void Process(int a, int b) { }

std::vector<int> v(100);
std::vector<int> w(100);
std::iota(v.begin(), v.end(), 0);
std::iota(w.begin(), w.end(), 0);

std::vector<size_t> indexes(v.size());
std::iota(indexes.begin(), indexes.end(), 0);

std::for_each(std::execution::par, indexes.begin(), indexes.end(),
    [&v, &w](size_t& id) {
        Process(v[id], w[id]);
    }
);
```

Since the order of execution is not specified, we cannot iterate through `v` and `w` using some global `i` variable. That's why we have to generate a separate vector of indices and then use it to access our containers.

Zip Iterators

Chapter Parallel Algorithms/par_iterating_multiple.cpp

```
void Process(int a, int b) { }

std::vector<int> v(100);
std::vector<int> w(100);
std::iota(v.begin(), v.end(), 0);
std::iota(w.begin(), w.end(), 0);

vec_zipper<int, int> zipped{ v, w };
std::for_each(std::execution::seq, zipped.begin(), zipped.end(),
    [](std::pair<int&, int&>& twoElements) {
        Process(twoElements.first, twoElements.second);
    }
);
```

This is a more elegant approach as we combine two containers into a single sequence and then iterate at once. The example uses a custom implementation of a `vec_zipper` that works only with `std::vector`. You can improve the code and make it more general or use third-party zip iterators (like boost[12]).

Erroneous Technique

What's more, it's necessary to mention one aspect.

According to the Standard [algorithms.parallel.exec][13]:

> Unless otherwise stated, implementations may make arbitrary copies of elements (with type `T`) from sequences where `is_trivially_copy_constructible_v<T>` and `is_-trivially_destructible_v<T>` are true. [Note: This implies that user-supplied function objects should not rely on object identity of arguments for such input sequences. Users for whom the object identity of the arguments to these function objects is important should

[12]https://www.boost.org/doc/libs/1_70_0/libs/iterator/doc/zip_iterator.html
[13]https://timsong-cpp.github.io/cppwp/n4659/algorithms.parallel#exec-2

consider using a wrapping iterator that returns a non-copied implementation object such as `reference_wrapper<T>` or some equivalent solution. —end note]

Thus you cannot write:

Don't Rely on the Addresses

```
vector<int> vec;
vector <int> other;
vector <int> external;
int* beg = vec.data();
std::transform(std::execution::par,
               vec.begin(), vec.end(), other.begin(),
               [&beg, &external](const int& elem) {
                   // use pointer arithmetic
                   auto index = &elem - beg;
                   return elem * externalVec[index];
               }
);
```

The code above uses pointer arithmetic to find the current index of the element. Then we can use this index to access other containers.

The technique, however, uses the assumption that `elem` is the exact element from the container and not its copy! Since the implementations might copy elements, the addresses might be completely unrelated! This faulty technique also assumes that the container is storing the items in a contiguous chunk of memory.

Only `for_each` and `for_each_n` have a guarantee that the elements are not being copied during the execution [alg.foreach][14]:

Implementations do not have the freedom granted under [algorithms.parallel.exec] to make arbitrary copies of elements from the input sequence.

[14]https://timsong-cpp.github.io/cppwp/n4659/alg.foreach#9

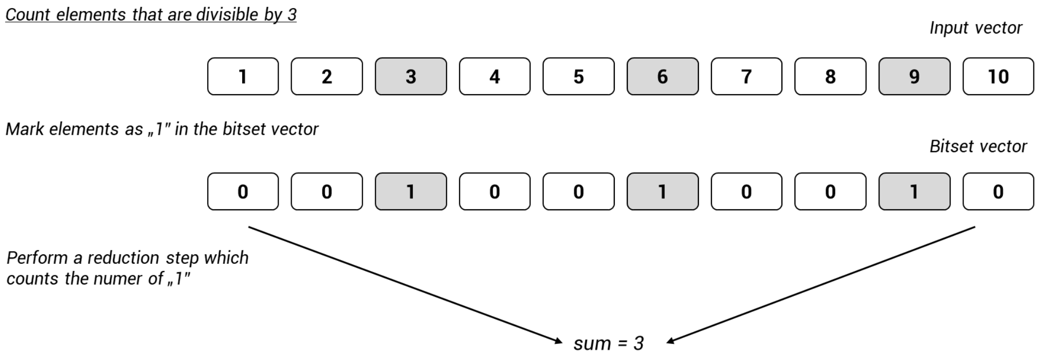
Counting Elements

To gain some practice, let's build an algorithm that counts the number of elements in a container. Our algorithm will be a version of another standard algorithm `count_if`.

The main idea is to use `transform_reduce` - a new "fused" algorithm. It first applies some unary function over an element and then performs a reduce operation.

To get the count of elements that satisfy some predicate, we can firstly filter each element (transform). We return `1` if the element passes the filter and `0` otherwise. Then, in the reduction step, we count how many elements returned `1`.

Here's a diagram that illustrates the algorithm for a simple case:


Count elements that are divisible by 3
Input vector
1 2 3 4 5 6 7 8 9 10
Mark elements as „1" in the bitset vector
Bitset vector
0 0 1 0 0 1 0 0 1 0
Perform a reduction step which counts the numer of „1"
sum = 3

Parallel Count IF Example

- The first step is to perform the `transform` step in `transform_reduce` algorithm. We return `1` for matching elements and `0` otherwise.
- Then the `reduce` step is used to compute the sum of all `1`. We have three values that satisfy the condition, so the output is `3`.

Here's the code:

Chapter Parallel Algorithms/par_count_if.cpp

```
template <typename Policy, typename Iter, typename Func>
std::size_t CountIf(Policy policy, Iter first, Iter last, Func predicate) {
    return std::transform_reduce(policy,
        first,
        last,
        std::size_t(0),
        std::plus<std::size_t>{},
        [&predicate](const Iter::value_type& v) {
            return predicate(v) ? 1 : 0;
        }
    );
}
```

We can run it on the following test containers:

Chapter Parallel Algorithms/par_count_if.cpp

```
std::vector<int> v(100);
std::iota(v.begin(), v.end(), 0);
auto NumEven = CountIf(std::execution::par, v.begin(), v.end(),
    [](int i) { return i % 2 == 0; }
);
std::cout << NumEven << '\n';
```

To get number of spaces in a string:

Chapter Parallel Algorithms/par_count_if.cpp

```
std::string_view sv = "Hello   Programming   World";
auto NumSpaces = CountIf(std::execution::seq, sv.begin(), sv.end(),
    [](char ch) { return ch == ' '; }
);
std::cout << NumSpaces << '\n';
```

Or even on a map:

Chapter Parallel Algorithms/par_count_if.cpp

```
std::map<std::string, int> CityAndPopulation{
    {"Cracow", 765000},
    {"Warsaw", 1745000},
    {"London", 10313307},
    {"New York", 18593220},
    {"San Diego", 3107034}
};
auto NumCitiesLargerThanMillion = CountIf(std::execution::seq,
    CityAndPopulation.begin(), CityAndPopulation.end(),
    [](const std::pair<const std::string, int>& p) {
        return p.second > 1000000;
    }
);
std::cout << CitiesLargerThanMillion << '\n';
```

The example uses simple test data and to have good performance over the sequential version the size of data would have to be significantly increased. For example, the cities and their population could be loaded from a database.

More Examples

Here's a list of a few other ideas where parallel algorithms could be beneficial:

- statistics - calculating various maths properties for a set of data
- processing CSV records line by line in parallel
- parsing files in parallel - one file per thread, or chunks of a file per thread
- computing summed-area tables
- parallel matrix operations
- parallel dot product

You can find a few more examples in the following articles:

- The Amazing Performance of C++17 Parallel Algorithms, is it Possible?[15]
- How to Boost Performance with Intel Parallel STL and C++17 Parallel Algorithms[16]

[15]https://www.bfilipek.com/2018/11/parallel-alg-perf.html
[16]https://www.bfilipek.com/2018/11/pstl.html

- Examples of Parallel Algorithms From C++17[17]
- Parallel STL And Filesystem: Files Word Count Example[18]

Chapter Summary

After reading the chapter, you should be equipped with the core knowledge about parallel algorithms. We discussed the execution policies, how they might be executed on hardware, what are the new algorithms.

At the moment, parallel algorithms show good potential. With only one extra parameter, you can easily parallelise your code. Previously that would require to use some third-party library or write a custom version of some thread pooling system.

For sure, we need to wait for more available implementations and experience. Currently, only Visual Studio and GCC 9.1 let you use parallel algorithms, and we're waiting for Clang's library to catch up. Executing code on GPU looks especially interesting.

It's also worth quoting the TS specification P0024[19]:

> The parallel algorithms and execution policies of the Parallelism TS are only a starting point. Already we anticipate opportunities for extending the Parallelism TS's functionality to increase programmer flexibility and expressivity. A fully-realised executors feature will yield new, flexible ways of creating execution, including the execution of parallel algorithms.

Things to remember:

- Parallel STL gives you a set of 69 algorithms that have overloads for the execution policy parameter.
- Execution policy describes how the algorithm might be executed.
- There are three execution policies in C++17 (`<execution>` header)
  - `std::execution::seq` - sequential
  - `std::execution::par` - parallel

[17]https://www.bfilipek.com/2018/06/parstl-tests.html
[18]https://www.bfilipek.com/2018/07/files-word-count.html
[19]https://wg21.link/P0024

  - `std::execution::par_unseq` - parallel and vectorised

- In parallel execution policy functors that are passed to algorithms cannot cause deadlocks and data races
- In parallel unsequenced policy functors cannot call vectorised unsafe instructions like memory allocations or any synchronisation mechanisms
- To handle new execution patterns there are also new algorithms: like `std::reduce`, `exclusive_scan` - They work out of order so the operations must be associative to generate deterministic results
- There are "fused" algorithms: `transform_reduce`, `transform_exclusive_scan`, `transform_inclusive_scan` that combine two algorithms together.
- Assuming there are no synchronisation points in the parallel execution, the parallel algorithms should be faster than the sequential version. Still, they perform more work - especially the setup and divisions into tasks.
- Implementations might usually use some thread pools to implement a parallel algorithm on CPU.

Compiler Support

As of today (July 2019) only two compilers/STL implementation support parallel algorithms: it's Visual Studio (since 2017 15.7) and GCC (since 9.1).

Visual Studio implements `par_unseq` as `par`, so you shouldn't expect any difference between code runs.

GCC implementation uses modified Intel PSTL and relies on OpenMP 4.0 and Intel TBB 2018. You need to install and link with `-ltbb` if you want to work with parallel algorithms.

For example:

```
g++-9.1 -std=c++17 -Wall -O2 sample.cpp -ltbb
```

For Building GCC 9.1 and Intel TBB you can check this guide @Solarian Programmer: C++17 STL Parallel Algorithms - with GCC 9.1 and Intel TBB on Linux and macOS[20].

Summary:

Feature	GCC	Clang	MSVC
Parallel Algorithms	9.1[21]	in progress	VS 2017 15.7[22]

There are also several other implementations out there:

- Codeplay - SYCL Parallel STL[23]
- STE||AR Group - HPX[24]
- Intel - Parallel STL[25] - based on OpenMP 4.0 and Intel® TBB.
- Parallel STL[26] - early Microsoft implementation for the Technical Specification.
- n3554 - proposal implementation (started by Nvidia)[27]
- Thibaut Lutz Implementation @Github[28] - early implementation

[20]https://solarianprogrammer.com/2019/05/09/cpp-17-stl-parallel-algorithms-gcc-intel-tbb-linux-macos/
[21]See in the article: GCC 9.1 Released and GCC 9 Release Series — Changes, New Features, and Fixes)
[22]See Announcing: MSVC Conforms to the C++ Standard | Visual C++ Team Blog
[23]http://github.com/KhronosGroup/SyclParallelSTL
[24]http://stellar-group.github.io/hpx/docs/html/hpx/manual/parallel.html
[25]https://software.intel.com/en-us/get-started-with-pstl
[26]https://parallelstl.codeplex.com/
[27]https://github.com/n3554/n3554
[28]http://github.com/t-lutz/ParallelSTL

15. Other Changes In The Library

C++17 is a significant update for the language, and it brings a lot of features in the Standard Library. So far, this book has covered the most important aspects, but there are many more things to describe!

This part of the book summaries briefly other changes in the Standard Library:

- What's `std::byte`?
- What are the new functionalities of maps and sets
- New algorithms: sampling
- Special Mathematical Functions
- Shared Pointers and Arrays
- Non-member `size()`, `data()` and `empty()`
- `constexpr` additions to the Standard Library
- How to lock multiple mutexes with `scoped_lock`?
- What's a polymorphic allocator? How can it help with memory management?

`std::byte`

`std::byte` is a small type that gives you a view of bytes and bits rather than numeric/char values (like `unsigned char`). It's defined as `enum`:

```
enum class byte : unsigned char {} ; // in <cstddef>
```

You can initialise `byte` from an unsigned char - which is in fact, another handy C++17 feature that allows you to init a scoped enum with the underlying type[1]. To convert from `byte` into a numeric type use `std::to_integer()`.

Let's see a basic example:

Chapter STL Other/byte.cpp

```
constexpr std::byte b{1};
// std::byte c{3535353}; // error: narrowing conversion from int

constexpr std::byte c{255};

// shifts:
constexpr auto b1 = b << 7;
static_assert(std::to_integer<int>(b)  == 0x01);
static_assert(std::to_integer<int>(b1) == 0x80);

// various bit operators, like &, |, &, etc
constexpr auto c1 = b1 ^ c;
static_assert(std::to_integer<int>(c)  == 0xff);
static_assert(std::to_integer<int>(c1) == 0x7f);

constexpr auto c2 = ~c1;
static_assert(std::to_integer<int>(c2)  == 0x80);
static_assert(c2 == b1);
```

The primary motivation behind `std::byte` is type-safety in the context of memory/byte access.

Extra Info

See the reference paper P0298R3[2].

[1]Read more in P0138 - https://wg21.link/P0138
[2]https://wg21.link/P0298R3

Improvements for Maps and Sets

In the Standard there are two notable features for maps and sets:

- Splicing Maps and Sets - P0083[3]
- New emplacement routines - N4279[4]

Splicing

You can now move nodes from one tree-based container (maps/sets) into other ones, without additional memory overhead/allocation.

For example:

Chapter STL Other/set_extract_insert.cpp

```
#include <set>
#include <string>
#include <iostream>

struct User {
    std::string name;

    User(std::string s) : name(std::move(s)) {
        std::cout << "User::User(" << name << ")\n";
    }
    ~User() {
        std::cout << "User::~User(" << name << ")\n";
    }
    User(const User& u) : name(u.name) {
        std::cout << "User::User(copy, " << name << ")\n";
    }

    friend bool operator<(const User& u1, const User& u2) {
        return u1.name < u2.name;
    }
};

int main() {
```

[3]https://wg21.link/P0083
[4]https://wg21.link/N4279

```
    std::set<User> setNames;
    setNames.emplace("John");
    setNames.emplace("Alex");
    setNames.emplace("Bartek");
    std::set<User> outSet;

    std::cout << "move John...\n";
    // move John to the outSet
    auto handle = setNames.extract(User("John"));
    outSet.insert(std::move(handle));

    for (auto& elem : setNames)
        std::cout << elem.name << '\n';

    std::cout << "cleanup...\n";
}
```

Output:

```
User::User(John)
User::User(Alex)
User::User(Bartek)
move John...
User::User(John)
User::~User(John)
Alex
Bartek
cleanup...
User::~User(John)
User::~User(Bartek)
User::~User(Alex)
```

In the above example, one element "John" is extracted from `setNames` into `outSet`. The `extract` method moves the found node out of the set and physically detaches it from the container. Later the extracted node can be inserted into a container of the same type.

Before C++17, if you wanted to move an object from one map (or set) and put it into another map (or set), you had to remove it from the first container and then copy/move into another. With the new functionality, you can manipulate tree nodes (by using implementation-defined type `node_type`) and leave objects unaffected. Such a technique allows you to handle non-movable elements and of course, is more efficient.

Emplace Enhancements for Maps and Unordered Maps

With C++17 you get two new methods for maps and unordered maps:

- `try_emplace()` - if the object already exists then it does nothing, otherwise it behaves like `emplace()`.
  - `emplace()` might move from the input parameter when the key is in the map, that's why it's best to use `find()` before such emplacement.
- `insert_or_assign()` - gives more information than `operator[]` - as it returns if the element was newly inserted or updated and also works with types that have no default constructor.

`try_emplace` Method

Here's an example:

Chapter STL Other/try_emplace_map.cpp

```
#include <iostream>
#include <string>
#include <map>

int main() {
    std::map<std::string, int> m;

    m["hello"] = 1;
    m["world"] = 2;

    // C++11 way:
    if (m.find("great") == std::end(m))
        m["great"] = 3;

    // the lookup is performed twice if "great" is not in the map

    // C++17 way:
    m.try_emplace("super", 4);
    m.try_emplace("hello", 5); // won't emplace, as it's
                               // already in the map

    for (const auto& [key, value] : m)
        std::cout << key << " -> " << value << '\n';
}
```

The behaviour of `try_emplace` is important in a situation when you move elements into the map:

Chapter STL Other/try_emplace_map_move.cpp

```cpp
#include <iostream>
#include <string>
#include <map>

int main() {
    std::map<std::string, std::string> m;
    m["Hello"] = "World";

    std::string s = "C++";
    m.emplace(std::make_pair("Hello", std::move(s)));

    // what happens with the string 's'?
    std::cout << s << '\n';
    std::cout << m["Hello"] << '\n';

    s = "C++";
    m.try_emplace("Hello", std::move(s));
    std::cout << s << '\n';
    std::cout << m["Hello"] << '\n';
}
```

The code tries to replace `["Hello", "World"]` into `["Hello", "C++"]`.

If you run the example the string `s` after `emplace` is empty and the value "World" is not changed into "C++"!

`try_emplace` does nothing in the case where the key is already in the container, so the `s` string is unchanged.

`insert_or_assign` Method

The second function `insert_or_assign`, inserts a new object in the map or assigns the new value. But as opposed to `operator[]` it also works with non-default constructible types.

For example:

Chapter STL Other/insert_or_assign.cpp

```cpp
#include <iostream>
#include <map>
#include <string>

struct User {
    std::string name;

    User(std::string s) : name(std::move(s)) {
        std::cout << "User::User(" << name << ")\n";
    }
    ~User() {
        std::cout << "User::~User(" << name << ")\n";
    }
    User(const User& u) : name(u.name) {
        std::cout << "User::User(copy, " << name << ")\n";
    }

    friend bool operator<(const User& u1, const User& u2) {
        return u1.name < u2.name;
    }
};

int main() {
    std::map<std::string, User> mapNicks;
    //mapNicks["John"] = User("John Doe"); // error: no default ctor for User()

    auto [iter, inserted] = mapNicks.insert_or_assign("John", User("John Doe"));
    if (inserted)
        std::cout << iter->first << " entry was inserted\n";
    else
        std::cout << iter->first << " entry was updated\n";
}
```

Output:

```
User::User(John Doe)
User::User(copy, John Doe)
User::~User(John Doe)
John entry was inserted
User::~User(John Doe)
```

In the example above we cannot use `operator[]` to insert a new value into the container, as it doesn't support types with non-default constructors. We can do it with the new function.

`insert_or_assign` returns a pair of `<iterator, bool>`. If the boolean value is true, it means the element was inserted into the container. Otherwise, it was reassigned.

Extra Info

See more information in Splicing Maps and Sets P0083[5] and New emplacement routines N4279[6].

[5]https://wg21.link/P0083
[6]https://wg21.link/N4279

Return Type of Emplace Methods

Since C++11 most of the standard containers got `.emplace*` methods. With those methods, you can create a new object in place, without additional object copies.

However, most of `.emplace*` methods didn't return any value - it was `void`. Since C++17 this is changed, and they now return the reference type of the inserted object.

For example:

```
// since C++11 and until C++17 for std::vector
template< class... Args >
void emplace_back( Args&&... args );

// since C++17 for std::vector
template< class... Args >
reference emplace_back( Args&&... args );
```

This modification should shorten the code that adds something to the container and then invokes some operation on that newly added object.

For example:

Chapter STL Other/emplace_return.cpp

```
#include <vector>
#include <string>

int main() {
    std::vector<std::string> stringVector;

    // in C++11/14:
    stringVector.emplace_back("Hello");
    // emplace doesn't return anything, so back() needed
    stringVector.back().append(" World");

    // in C++17:
    stringVector.emplace_back("Hello").append(" World");
}
```

Extra Info

See more information in the paper: P0084R2[7].

[7] http://wg21.link/p0084r2

Sampling Algorithms

New algorithm - `std::sample` - that selects `n` elements from the sequence:

Chapter STL Other/sample.cpp

```
#include <iostream>
#include <random>
#include <iterator>
#include <algorithm>

int main() {
    std::vector<int> v { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    std::vector<int> out;
    std::sample(v.begin(),               // range start
                v.end(),                 // range end
                std::back_inserter(out), // where to put it
                3,                       // number of elements to sample
                std::mt19937{std::random_device{}()});

    std::cout << "Sampled values: ";
    for (const auto &i : out)
        std::cout << i << ", ";
}
```

Possible output:

```
Sampled values: 1, 4, 9,
```

Extra Info

The new sampling algorithms come from the adoption of Library Fundamentals V1 TS Components, Sampling P0220R1[8].

[8]https://wg21.link/p0220

New Mathematical Functions

With C++17 we get lots of new mathematical functions like `std::gcd` (Greatest Common Divisor), `std::lcm` (Least Common Multiple), `std::clamp` and other special ones.

For example `std::gcm` and `std::lcm`, introduced in P0295R0[9], declared in `<numerics>` header:

Chapter STL Other/numeric_gcd_lcm.cpp

```
#include <iostream>
#include <numeric>  // for gcm, lcm

int main() {
    std::cout << std::gcd(24, 60) << ', ';
    std::cout << std::lcm(15, 50) << '\n';
}
```

Output:

```
12, 150
```

Another useful function is `std::clamp(v, min, max)`, declared in `<algorithm>`, from P0025[10]:

Chapter STL Other/clamp.cpp

```
#include <iostream>
#include <algorithm>  // clamp

int main() {
    std::cout << std::clamp(300, 0, 255) << ', ';
    std::cout << std::clamp(-10, 0, 255) << '\n';
}
```

The output: `255, 0`

[9]http://wg21.link/p0295r0
[10]http://wg21.link/p0025

What's more, there are newly available special functions, defined in the `<cmath>` header.

Function	Description
`assoc_laguerre`	Functions compute the associated Laguerre polynomials of their respective arguments n, m, and x
`assoc_legendre`	Functions compute the associated Legendre functions of their respective arguments l, m, and x
`beta`	Functions Compute the beta function of their respective arguments x and y
`comp_ellint_1`	Complete elliptic integral of the first kind of their respective arguments k
`comp_ellint_2`	Compute the complete elliptic integral of the second kind of their respective arguments k
`comp_ellint_3`	Compute the complete elliptic integral of the third kind of their respective arguments k and nu
`cyl_bessel_i`	Compute the regular modified cylindrical Bessel functions of their respective arguments nu and x
`cyl_bessel_j`	Compute the cylindrical Bessel functions of the first kind of their respective arguments nu and x
`cyl_bessel_k`	Compute the irregular modified cylindrical Bessel functions of their respective arguments nu and x
`cyl_neumann`	Compute the cylindrical Neumann functions, also known as the cylindrical Bessel functions of the second kind, of their respective arguments nu and x
`ellint_1`	Compute the incomplete elliptic integral of the first kind of their respective arguments k and phi (phi measured in radians)
`ellint_2`	Compute the incomplete elliptic integral of the second kind of their respective arguments k and phi (phi measured in radians)
`ellint_3`	Compute the incomplete elliptic integral of the third kind of their respective arguments k, nu, and phi (phi measured in radians)
`expint`	Compute the exponential integral of their respective arguments x
`hermite`	Compute the Hermite polynomials of their respective arguments n and x
`laguerre`	Compute the Laguerre polynomials of their respective arguments n and x
`legendre`	Compute the Legendre polynomials of their respective arguments l and x
`riemann_zeta`	Compute the Riemann zeta function of their respective arguments x
`sph_bessel`	Compute the spherical Bessel functions of the first kind of their respective arguments n and x
`sph_legendre`	Compute the spherical associated Legendre functions of their respective arguments l, m, and theta (theta measured in radians)
`sph_neumann`	Compute the spherical Neumann functions, also known as the spherical Bessel functions of the second kind, of their respective arguments n and x

Extra Info

The above special functions were introduced in N1542 ver 3[11].

[11]http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1542.pdf

Shared Pointers and Arrays

Before C++17, only `unique_ptr` was able to handle arrays out of the box (without the need to define a custom deleter). Now it's also possible with `shared_ptr`.

```
std::shared_ptr<int[]> ptr(new int[10]);
```

Please note that `std::make_shared` doesn't support arrays in C++17. But this will be fixed in C++20 (see P0674[12] which is already merged into C++20)

Another important remark is that raw arrays should be avoided. It's usually better to use standard containers. However, sometimes, you don't have the luxury to use vectors or lists - for example, in an embedded environment, or when you work with third-party API. In that situation, you might end up with a raw pointer to an array. With C++17, you'll be able to wrap those pointers into smart pointers (`std::unique_ptr` or `std::shared_ptr`) and be sure the memory is deleted correctly.

Extra Info

See the initial proposal: P0414R2[13].

[12]https://wg21.link/p0674
[13]https://wg21.link/P0414R2

Non-member `size()`, `data()` and `empty()`

Following the approach from C++11 regarding non-member `std::begin()` and `std::end()` C++17 brings three new functions.

Chapter STL Other/non_member_functions.cpp

```
#include <iostream>
#include <vector>

template <class Container>
void PrintBasicInfo(const Container& cont) {
    std::cout << typeid(cont).name() << '\n';
    std::cout << std::size(cont) << '\n';
    std::cout << std::empty(cont) << '\n';

    if (!std::empty(cont))
        std::cout << *std::data(cont) << '\n';
}

int main() {
    std::vector<int> iv { 1, 2, 3, 4, 5 };
    PrintBasicInfo(iv);
    float arr[4] = { 1.1f, 2.2f, 3.3f, 4.4f };
    PrintBasicInfo(arr);
}
```

Output (from GCC 8.2):

```
St6vectorIiSaIiEE
5
0
1
A4_f
4
0
1.1
```

Extra Info

The new functions are located in `<iterator>`, and the referencing paper is N4280[14].

[14]https://wg21.link/n4280

constexpr Additions to the Standard Library

With this enhancement you can work with iterators, `std::array`, range-based for loops in `constexpr` contexts.

The below example shows a basic implementation of `accumulate` algorithm as `constexpr` (C++11/14/17 version of `std::accumulate` is not `constexpr`):

Chapter STL Other/constexpr_additions.cpp

```
#include <array>

template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init) {
    for (auto &&obj : range) { // begin/end are constexpr
        init += func(obj);
    }
    return init;
}

constexpr int square(int i) { return i*i; }

int main() {
    constexpr std::array arr{ 1, 2, 3 }; // class deduction...

    // with constexpr lambda
    static_assert(SimpleAccumulate(arr, [](int i) constexpr {
            return i * i;
        }, 0) == 14);

    // with constexpr function
    static_assert(SimpleAccumulate(arr, &square, 0) == 14);

    return arr[0];
}
```

C++14 compilers would not compile the above example, but it's now possible with C++17 support.

There are several features used in this code:

- `SimpleAccumulate` is a `constexpr` template function and uses range access (hidden in range based for loop) to iterate over the input range.

- `arr` is deduced as `std::array<3, int>` - class template deduction works here.
- the code uses a `constexpr` lambda.
- `static_assert` without any message.
- `std::begin()` and `std::end()` are also `constexpr` since C++17

You can also see another example of `constexpr` additions in Chapter about General Language Features: Constexpr Lambda.

Each C++ Standard allows more and more code to be `constexpr`. In C++17, we can start using basic containers in constant expressions. In C++20 we'll get more standard algorithms that are declared as `constexpr`.

Extra Info

The main referencing paper is P0031 - Proposal to Add Constexpr Modifiers to reverse_iterator, move_iterator, array and Range Access[15].

[15]https://wg21.link/p0031

`std::scoped_lock`

With C++11 and C++14 we got the threading library and many support functionalities.

For example, with `std::lock_guard` you can take ownership of a mutex and lock it in RAII style:

```
std::mutex m;

std::lock_guard<std::mutex> lock_one(m);
// unlocked when lock_one goes out of scope...
```

The above code works, however, only for a single mutex. If you wanted to lock several mutexes, you had to use a different pattern, for example:

```
std::mutex first_mutex;
std::mutex second_mutex;

// ...

std::lock(fist_mutex, second_mutex);
std::lock_guard<std::mutex> lock_one(fist_mutex, std::adopt_lock);
std::lock_guard<std::mutex> lock_two(second_mutex, std::adopt_lock);
// ..
```

With C++17 things get a bit easier as with `std::scoped_lock` you can lock a variadic number of mutexes at the same time.

```
std::scoped_lock lck(first_mutex, second_mutex);
```

Due to compatibility `std::lock_guard` couldn't be extended with a variadic number of input mutexes and that's why a new type - `scoped_lock` - was needed.

Extra Info

You can read more information in P0156[16].

[16]https://wg21.link/p0156

Polymorphic Allocator, `pmr`

The polymorphic allocator is an enhancement to the standard allocator from the Standard Library.

In short, a polymorphic allocator conforms to the rules of an allocator from the Standard Library, but at its core, it uses a memory resource object to perform the memory management. Polymorphic Allocator contains a pointer to a memory resource class, and that's why it can use a virtual method dispatch. You can change the memory resource at runtime while keeping the type of the allocator.

All the types for polymorphic allocators live in a separate namespace `std::pmr` (PMR stands for Polymorphic Memory Resource), in the `<memory_resource>` header.

Core elements of `pmr`:

- `std::pmr::memory_resource` - is an abstract base class for all other implementations. It defines the following pure virtual methods: `do_allocate`, `do_deallocate` and `do_is_equal`.
- `std::pmr::polymorphic_allocator` - is an implementation of a standard allocator that uses `memory_resource` object to perform memory allocations and deallocations.
- global memory resources accessed by `new_delete_resource()` and `null_memory_resource()`
- a set of predefined memory pool resource classes:
  - `synchronized_pool_resource`
  - `unsynchronized_pool_resource`
  - `monotonic_buffer_resource`
- template specialisations of the standard containers with polymorphic allocator, for example `std::pmr::vector`, `std::pmr::string`, `std::pmr::map` and others. Each specialisation is defined in the same header file as the corresponding container.

Here's a short overview of the predefined memory resources:

Resource	Description
`new_delete_resource()`	a free function that returns a pointer to a global "default" memory resource. It manages memory with the global `new` and `delete`
`null_memory_resource()`	a free function that returns a pointer to a global "null" memory resource which throws `std::bad_alloc` on every allocation
`synchronized_pool_resource`	thread-safe allocator that manages pools of different sizes. Each pool is a set of chunks that are divided into blocks of uniform size.
`unsynchronized_pool_resource`	non-thread-safe `pool_resource`
`monotonic_buffer_resource`	non-thread-safe, fast, special-purpose resource that gets memory from a preallocated buffer, but doesn't release it with deallocation.

It's also worth mentioning that pool resources (including `monotonic_buffer_resource`) can be chained. So that if there's no available memory in a pool, the allocator will allocate from the "upstream" resource.

Below you can find a simple example of `monotonic_buffer_resource` and `pmr::vector`:

Chapter STL Other/pmr_monotonic_resource.cpp

```cpp
#include <iostream>
#include <memory_resource>   // pmr core types
#include <vector>            // pmr::vector

int main() {
    char buffer[64] = {}; // a small buffer on the stack
    std::fill_n(std::begin(buffer), std::size(buffer) - 1, '_');
    std::cout << buffer << '\n';

    std::pmr::monotonic_buffer_resource pool{std::data(buffer), std::size(buffer)};

    std::pmr::vector<char> vec{ &pool };
    for (char ch = 'a'; ch <= 'z'; ++ch)
        vec.push_back(ch);

    std::cout << buffer << '\n';
}
```

Possible output:

```
_______________________________________________________________
aababcdabcdefghabcdefghijklmnopabcdefghijklmnopqrstuvwxyz______
```

In the above example, we use a monotonic buffer resource initialised with a memory chunk from the stack. By using a simple `char buffer[]` array, we can easily print the contents of the "memory". The vector gets memory from the pool, and if there's no more space available, it will ask for memory from the "upstream" resource (which is `new_delete_resource` by default). The example shows vector reallocations when there's a need to insert more elements. Each time the vector gets more space, so it eventually fits all of the letters.

More Information

This section only touched upon the idea of polymorphic allocators and memory resources. If you want to learn more about this topic, there's a long chapter in C++17 The Complete Guide[17] by Nicolai Josuttis. There are also several conference talks, for example Allocators: The Good Parts by Pablo Halpern[18] from CppCon 2017.

Extra Info

See more information in P0220R1[19] and P0337R0[20].

[17]https://leanpub.com/cpp17
[18]https://channel9.msdn.com/Events/GoingNative/CppCon-2017/119
[19]https://wg21.link/p0220r1
[20]https://wg21.link/p0337r0

Compiler support

Feature	GCC	Clang	MSVC
`std::byte`	7.1	5.0	VS 2017 15.3
Improvements for Maps and Sets	7.0	3.9	VS 2017 15.5
`insert_or_assign()`/`try_emplace()` for maps	6.1	3.7	VS 2017 15
Emplace Return Type	7.1	4.0	VS 2017 15.3
Sampling algorithms	7.1	In Progress	VS 2017 15
`gcd` and `lcm`	7.1	4.0	VS 2017 15.3
`clamp`	7.1	3.9	VS 2015.3
Special Mathematical Functions	7.1	Not yet	VS 2017 15.7
Shared Pointers and Arrays	7.1	In Progress	VS 2017 15.5
Non-member `size()`, `data()` and `empty()`	6.1	3.6	VS 2015
`constexpr` Additions to the Standard Library	7.1	4.0	VS 2017 15.3
`scoped_lock`	7.1	5.0	VS 2017 15.3
Polymorphic Allocator & Memory Resource	9.1	In Progress	VS 2017 15.6

16. Removed And Deprecated Library Features

In the chapter about Removed or Fixed Language Features, the book focused only on the language side. But The Standard Library was also cleaned-up in C++17. This chapter shows the list of most [1] of the removed or deprecated types and utilities.

In this chapter, you'll learn:

- Why `auto_ptr` was removed and why you should stick with smart pointers
- Why `std::random_shuffle` algorithm was removed and what's a better alternative for it
- How to implement custom iterator and not derive from deprecated `std::iterator` type
- What are the other smaller elements that were deprecated or removed

[1] If you want to find the list of all deprecated elements, you can check "Annex D Compatibility features" of the Standard, for example under this link: https://timsong-cpp.github.io/cppwp/n4659/depr

Removing `auto_ptr`

Probably the best news of all!

C++98 added `auto_ptr` as a way to support basic RAII features for raw pointers. However, due to the lack of move semantics in the language, this smart pointer could be easily misused and cause runtime errors.

Here's an example where `auto_ptr` might cause a crash:

Chapter Removed Lib Features/auto_ptr_crash.cpp

```
void doSomething(std::auto_ptr<int> myPtr) {
    *myPtr = 11;
}

void AutoPtrTest() {
    std::auto_ptr<int> myTest(new int(10));
    doSomething(myTest);
    *myTest = 12; // uups!
}
```

`doSomething()` takes `auto_ptr` by value, but since it's not a shared pointer, it gets the unique ownership of the managed object. Later, when the function is completed, the copy of the pointer goes out of scope, and the object is deleted.

In `AutoPtrTest()` when `doSomething()` is finished the pointer is already cleaned up, and you'll get undefined behaviour when calling `*myTest = 12`.

In C++11 we got smart pointers: `unique_ptr`, `shared_ptr` and `weak_ptr`. With the move semantics, the language could finally support proper unique resource transfers. Also, new smart pointers can be stored in standard containers, which was not possible with `auto_ptr`. You should replace `auto_ptr` with `unique_ptr` as it's the direct and the best equivalent for `auto_ptr`.

We can rewrite the example so it uses `unique_ptr`:

Using unique_ptr

```
void doSomething(std::unique_ptr<int> myPtr) {
    *myPtr = 11;
}

void AutoPtrTest() {
    auto myTest = std::make_unique<int>(10);
    doSomething(myTest); // won't compile!
    *myTest = 12; // use after move ??
}
```

Now, the code won't compile as you need to move `unique_ptr` into `doSomething()`. Since the move is explicit it requires to possibly rething the solution. For example, in this case, maybe `doSomething()` doesn't need the ownership of the pointer? Perhaps it's better to pass a raw pointer, without the ownership?

Alternatively, you can use `shared_ptr` and then, the pointer won't be deleted after `doSomething()` is finished, as `shared_ptr` uses reference counting.

New smart pointers are much more powerful and safer than `auto_ptr`, so it has been deprecated since C++11. Compilers should report a warning:

```
warning: 'template<class> class std::auto_ptr' is deprecated
```

Now, when you compile with a conformant C++17 compiler, you'll get an error.

Here's the error from MSVC 2017 when using `/std:c++latest`:

```
error C2039: 'auto_ptr': is not a member of 'std'
```

If you need help with the conversion from `auto_ptr` to `unique_ptr` you can check Clang Tidy, as it provides auto conversion: Clang Tidy: modernize-replace-auto-ptr[2].

Extra Info

The change was proposed in: N4190[3].

[2]https://clang.llvm.org/extra/clang-tidy/checks/modernize-replace-auto-ptr.html
[3]https://wg21.link/n4190

Removed `std::random_shuffle`

The `random_shuffle(first, last)` and `random_shuffle(first, last, rng)` functions were marked already as deprecated in C++14. The reason was that in most cases it used the `rand()` function, which is considered inefficient and even error-prone (as it uses global state). Alternatively, you could provide the `rng` function parameter that appeared to be unusable in practice. If you need the same functionality, use `std::shuffle`:

```
template< class RandomIt, class URBG >
void shuffle( RandomIt first, RandomIt last, URBG&& g );
```

`std::shuffle` takes a random number generator as the third template argument, which is safer, easier to use and more scalable.

Have a look at the following example on how to convert from `random_shuffle` to `shuffle`:

Chapter Removed Lib Features/shuffle.cpp

```
std::vector<int> vec = { 0, 1, 2, 3, 4, 5 };

// Pre-C++17:
std::random_shuffle(begin(vec), end(vec));

for (auto& elem : vec)
    std::cout << elem << ", ";

// C++17 version:
std::random_device randDev;
std::mt19937 gen(randDev());

std::shuffle(begin(vec), end(vec), gen);

for (auto& elem : vec)
    std::cout << elem << ", ";
```

Extra Info

See more information in N4190[4].

[4]https://wg21.link/n4190

“Removing Old functional Stuff”

Functions like `bind1st()`/`bind2nd()`/`mem_fun()`, … were introduced in the C++98-era and are not needed now as you can apply a lambda. What’s more, the functions were not updated to handle perfect forwarding, `decltype` and other techniques from C++11. Thus it’s best not to use them in modern code.

Removed functions:

- `unary_function()`/`pointer_to_unary_function()`
- `binary_function()`/`pointer_to_binary_function()`
- `bind1st()`/`binder1st`
- `bind2nd()`/`binder2nd`
- `ptr_fun()`
- `mem_fun()`
- `mem_fun_ref()`

For example to replace `bind1st`/`bind2nd` you can use lambdas or `std::bind` (available since C++11) or `std::bind_front` that should be available since C++20.

Chapter Removed Lib Features/bind.cpp

```
auto onePlus = std::bind1st(std::plus<int>(), 1);
auto minusOne = std::bind2nd(std::minus<int>(), 1);
std::cout << onePlus(10) << ", " << minusOne(10) << '\n';

// with hardcoded lambdas:
auto lamOnePlus1 = [](int b) { return 1 + b; };
auto lamMinusOne1 = [](int b) { return b - 1; };
std::cout << lamOnePlus1(10) << ", " << lamMinusOne1(10) << '\n';

// a capture with an initializer
auto lamOnePlus = [a=1](int b) { return a + b; };
auto lamMinusOne = [a=1](int b) { return b - a; };
std::cout << lamOnePlus(10) << ", " << lamMinusOne(10) << '\n';
```

Extra Info

See more information in N4190[5].

[5]https://wg21.link/n4190

std::iterator Is Deprecated

The Standard Library API requires that each iterator type has to expose five typedefs:

- iterator_category - the type of the iterator
- value_type - type stored in the iterator
- difference_type - the type that is the result of subtracting two iterators
- pointer - pointer type of the stored type
- reference - the reference type of the stored type

iterator_category must be one of input_iterator_tag, forward_iterator_-tag, bidirectional_iterator_tag or random_access_iterator_tag.

Before C++17, if you wanted to define a custom iterator, you could use std::iterator as a base class. In C++14 it's defined as:

```
template<
    class Category,
    class T,
    class Distance = std::ptrdiff_t,
    class Pointer = T*,
    class Reference = T&
> struct iterator;
```

This helper class made defining the typedefs in a short way:

```
class ColumnIterator
 : public std::iterator<std::random_access_iterator_tag, Column>
{
    // ...
};
```

In C++17 you must not derive from std::iterator and you need to write the trait typedefs explicitly:

```
class ColumnIterator {
public:
    using iterator_category = std::random_iterator_tag;
    using value_type = Column;
    using difference_type = std::ptrdiff_t;
    using pointer = Column*;
    using reference = Column&;

    // ...
};
```

While you have to write more code, it's much easier to read, and it's less error-prone.

For example, the referencing paper mentions the following example from The Standard Library:

```
template <class T, class charT = char, class traits = char_traits<charT> >
class ostream_iterator:
  public iterator<output_iterator_tag, void, void, void, void>;
```

In the above code, it's not clear what all of those four `void` types mean in the definition.

Additionally, `std::iterator` could lead to complicated errors in the name lookup, especially if you happen to use the same name in the derived iterator as in the base class.

Extra Info

You can read more information in the paper P0174R2 - Deprecating Vestigial Library Parts in C++17[6].

[6]https://wg21.link/p0174r2

Other Smaller Removed or Deprecated Items

Let's have a look at some smaller elements that were altered in the Standard Library.

Deprecating `shared_ptr::unique()`

In C++14 `shared_ptr::unique()` was defined as `use_count() == 1`. But since `use_count()` is only approximation in multithreaded environments (as it doesn't imply any synchronisation access) then `unique()` it not reliable.

See more information in P0521[7].

Deprecating `<codecvt>`

The `<codecvt>` header declares several conversion utilities: `codecvt_utf8`, `codecvt_utf16` and `codecvt_utf8_utf16`. But those classes are hard to use, unsafe and not well specified.

Note: the class `std::codesvt` is not deprecated, as it's located in another header `<locale>`. So you can still use that.

See more information in P0618R0[8]

Removing Deprecated Iostreams Aliases

Since C++11 the following iostream types and methods were deprecated, and now they are removed from the Library.

```
typedef T1 io_state;  // T1 is integer
typedef T2 open_mode; // T2 is integer
typedef T3 seek_dir;  // T3 is integer
typedef implementation-defined streamoff;
typedef implementation-defined streampos;
basic_streambuf::stossc()
```

Also, the methods and overrides that depend on the above types were removed.

[7]http://wg21.link/P0521
[8]https://wg21.link/P0618R0

Previously they were all declared in Annex D of the Standard: [depr.ios.members][9].

See more information in P0004R1[10].

Deprecate C library headers

The following headers are now deprecated:

- `<ccomplex>`
- `<cstdalign>`
- `<cstdbool>`
- `<ctgmath>`

This is a result of cleaning up places that depend on the C99 specification. In C++17 the Standard relates to C11 rather than C99.

See more information in P0063R3[11].

Deprecate `std::result_of`

The type trait `std::result_of` used a non-variadic template declaration which limited its uses. It's advised to use enhanced traits, for example `std::invoke_result`.

See more information in P0604R0[12].

Deprecate `std::memory_order_consume` Temporarily

The memory model of `memory_order_consume` is hard to implement and not well specified in the Standard. The model is now temporarily deprecated and may reappear in the future.

See more information in P0371R1[13].

[9]https://timsong-cpp.github.io/cppwp/n4140/depr.ios.members
[10]https://wg21.link/P0004R1
[11]https://wg21.link/P0063R3
[12]https://wg21.link/P0604R0
[13]https://wg21.link/P0371R1

Remove allocator support from `std::function`

`std::function` uses type-erasure to handle callable objects. It's very complicated (or not implementable efficiently) to have allocator support for this type, so it was decided to remove this from the Standard.

See more information in P0302R1[14].

Compiler support

Feature	GCC	Clang	MSVC
Removing `auto_ptr`, `random_shuffle`, old `<functional>` stuff	No [15]	not yet	VS 2015
Deprecating `std::iterator`	not yet	not yet	VS 2017 15.5
Deprecating `shared_ptr::unique()`	not yet	not yet	VS 2017 15.5
Deprecating `<codecvt>`	not yet	not yet	VS 2017 15.5
Removing Deprecated Iostreams Aliases	not yet	3.8	VS 2015.2
Deprecate `result_of`	not yet	not yet	VS 2017 15.3
Removing Allocator Support In `std::function`	not yet	4.0	VS 2017 15.5
C++17 should refer to C11 instead of C99	9.1	7.0	VS 2015
Removing Deprecated Iostreams Aliases	not yet	3.8	VS 2015.2

[14] https://wg21.link/P0302R1
[15] Kept for compatibility.

Part 3 - More Examples and Use Cases

In the first two parts of the book, you've seen Language and the Standard Library features. Most of the time, the book presented them as isolated from each other. However, the real power of each new C++ Standard comes from the composition of the new building blocks. This part will lead you through several larger examples where multiple C++17 elements were used together.

In this part, you'll learn:

- How to refactor code with `std::optional` and `std::variant`
- How `if constexpr` simplifies the complex meta-programming code.
- What are the uses for the `[[nodiscard]]` attribute.
- How to work with the filesystem and parallel algorithms and improve the performance of CSV Reader application.

17. Refactoring with `std::optional` and `std::variant`

`std::variant` and `std::optional` are called “vocabulary” types because you can leverage them to convey more design information. Your code can be much more compact and more expressive.

This chapter will show you one example of how `std::optional` and `std::variant` can help with the refactoring of one function. We’ll start with some legacy code, and through several steps, we’ll arrive with a much better solution. To give you a better understanding, you’ll see the pros and cons of each step.

The Use Case

Consider a function that takes the current mouse selection for a game. The function scans the selected range and computes several outputs:

- the number of animating objects
- if there are any civil units in the selection
- if there are any combat units in the selection

The existing code looks like this:

```
class ObjSelection {
public:
    bool IsValid() const { return true; }
    // more code...
};

bool CheckSelectionVer1(const ObjSelection &objList,
                        bool *pOutAnyCivilUnits,
                        bool *pOutAnyCombatUnits,
                        int *pOutNumAnimating);
```

As you can see above, the function uses a lot of output parameters (in the form of raw pointers), and it returns `true/false` to indicate success (for example the input selection might be invalid).

The implementation of the function is not relevant now, but here's an example code that calls this function:

```
ObjSelection sel;

bool anyCivilUnits { false };
bool anyCombatUnits { false };
int numAnimating { 0 };
if (CheckSelectionVer1(sel, &anyCivilUnits, &anyCombatUnits, &numAnimating))
{
    // ...
}
```

How can we improve the function?

There might be several things:

- Look at the caller's code: we have to create all the variables that will hold the outputs. It definitely generates code duplication if you call the function in many places.
- Output parameters: Core guidelines suggests not to use them.
  - F.20: For "out" output values, prefer return values to output parameters[1]
- If you have raw pointers you have to check if they are valid. You might get away with the checks if you use references for the output parameters.
- What about extending the function? What if you need to add another output param?

Anything else?

How would you refactor this?

Motivated by Core Guidelines and new C++17 features, here's the plan for how we can improve this code:

1. Refactor output parameters into a tuple that will be returned.
2. Refactor tuple into a separate `struct` and reduce the tuple to a pair.
3. Use `std::optional` to express if the value was computed or not.
4. Use `std::variant` to convey not only the optional result but also the full error information.

The Tuple Version

The first step is to convert the output parameters into a tuple and return it from the function.

According to F.21: To return multiple "out" values, prefer returning a tuple or struct[2]:

> A return value is self-documenting as an "output-only" value. Note that C++ does have multiple return values, by the convention of using a `tuple` (including `pair`), possibly with the extra convenience of `tie` at the call site.

After the change the code might look like this:

[1]https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f20-for-out-output-values-prefer-return-values-to-output-parameters

[2]https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#f21-to-return-multiple-out-values-prefer-returning-a-tuple-or-struct

```
std::tuple<bool, bool, bool, int>
CheckSelectionVer2(const ObjSelection &objList) {
    if (!objList.IsValid())
        return {false, false, false, 0};

    // local variables:
    int numCivilUnits = 0;
    int numCombat = 0;
    int numAnimating = 0;

    // scan...

    return {true, numCivilUnits > 0, numCombat > 0, numAnimating };
}
```

A bit better… isn't it? The tuple version has the following advantages:

- There's no need to check raw pointers.
- Code is more expressive. We return everything in a single object.

What's more on the caller site, you can use Structured Bindings to wrap the returned tuple:

```
auto [ok, anyCivil, anyCombat, numAnim] = CheckSelectionVer2(sel);
if (ok) {
    // ...
}
```

Unfortunately, this version might not be the best one. For example, there's a risk of forgetting the order of outputs from the tuple.

The problem of function extensions is also still present. So when you'd like to add another output value, you have to extend this tuple and the caller site.

We can fix this with one further step: instead of a tuple, use a structure (as also suggested by the Core Guidelines).

A Separate Structure

The outputs seem to represent related data. That's why it's probably a good idea to wrap them into a `struct` called `SelectionData`.

```
struct SelectionData {
    bool anyCivilUnits { false };
    bool anyCombatUnits { false };
    int numAnimating { 0 };
};
```

And then you can rewrite the function into:

```
std::pair<bool, SelectionData> CheckSelectionVer3(const ObjSelection &objList) {
    SelectionData out;

    if (!objList.IsValid())
        return {false, out};

    // scan...

    return {true, out};
}
```

And the caller site:

```
if (auto [ok, selData] = CheckSelectionVer3(sel); ok)
{
    // ...
}
```

The code uses `std::pair` so we still preserve the success flag, it's not the part of the new `struct`.

The main advantage that we achieved here is the improved code structure and extensibility. If you want to add a new parameter, then extend the structure. Previously - with a list of output parameters in the function declaration - you'd have to update much more code.

But isn't `std::pair<bool, MyType>` similar to the concept of `std::optional`?

With `std::optional`

From the `std::optional` chapter:

> std::optional is a wrapper type to express "null-able" types. It either contains a value, or it's empty. It doesn't use any extra memory allocation.

That seems to be the right choice for our code. We can remove the `ok` variable and rely on the semantics of the optional.

The new version of the code:

```
std::optional<SelectionData> CheckSelectionVer4(const ObjSelection &objList) {
    if (!objList.IsValid())
        return std::nullopt;

    SelectionData out;

    // scan...

    return out;
}
```

And the caller site:

```
if (auto ret = CheckSelectionVer4(sel); ret.has_value()) {
    // access via *ret or even ret->
    // ret->numAnimating
}
```

What are the advantages of the optional version? Let's name a few:

- Clean and expressive form - `optional` expresses nullable types.
- Efficiency - implementations of `optional` are not permitted to use additional storage, such as dynamic memory. The contained value shall be allocated in a region of the optional storage suitably aligned for the type `T`.

With `std::variant`

The last implementation with `std::optional` omits one crucial aspect: error handling. There's no way to know the reason why a value wasn't computed. For example, with the version where `std::pair` was used, we were able to return an error code to indicate the reason. What can we do about that?

If you need full information about the error that might occur in the function, you can think about an alternative approach with `std::variant`.

```
enum class [[nodiscard]] ErrorCode {
    InvalidSelection,
    Undefined
};


variant<SelectionData, ErrorCode> CheckSelectionVer5(const ObjSelection &objList) {
    if (!objList.IsValid())
        return ErrorCode::InvalidSelection;

    SelectionData out;
    // scan...

    return out;
}
```

As you see the code uses `std::variant` with two possible alternatives: either `SelectionData` or `ErrorCode`. It's almost like a pair, except that you'll always see one active value.

You can use the above implementation:

```
if (auto retV5 = CheckSelectionVer5(sel);
        std::holds_alternative<SelectionData>(retV5)) {
    std::cout << "ok..."
              << std::get<SelectionData>(retV5).numAnimating << '\n';
}
else {
    switch (std::get<ErrorCode>(retV5))
    {
        case ErrorCode::InvalidSelection:
            std::cerr << "Invalid Selection!\n";
            break;
        case ErrorCode::Undefined:
            std::cerr << "Undefined Error!\n";
            break;
    }
}
```

As you can see, with `std::variant` you have even more information than when `std::optional` was used. You can return error codes and respond to possible failures.

`std::variant<ValueType, ErrorCode>` might be a possible implementation of `std::expected` - a new vocabulary type that might go into the future version of The Standard Library.

Wrap up

You can play with the code in:

`Chapter Refactoring With Optional And Variant/refactoring_optional_-variant.cpp`.

In this chapter, you've seen how to refactor lots of ugly-looking output parameters to a nicer `std::optional` version. The optional wrapper clearly expresses that the computed value might be absent. Also, you've seen how to wrap several function parameters into a separate `struct`. Having one separate type lets you easily extend the code while keeping the logical structure untouched.

And finally, if you need the full information about errors inside a function, then you might also consider an alternative with `std::variant`. This type gives you a chance to return a full error code.

18. Enforcing Code Contracts With `[[nodiscard]]`

C++17 brought a few more standard attributes. By using those extra annotations, you can make your code not only readable to other developers but also the compiler can use this knowledge. For example, it might produce more warnings about potential mistakes. Or the opposite: it might avoid a warning generation because it will notice a proper intention (for example with `[[maybe_unused]]`).

In this chapter, you'll see how one attribute - `[[nodiscard]]` - can be used to provide better safety in the code.

Introduction

The [[nodiscard]] attribute was mentioned in the Attributes Chapter, but here's a simple example to recall its properties.

The attribute is used to mark the return value of functions:

```
[[nodiscard]] int Compute();
```

When you call such function and don't assign the result:

```
void Foo() {
    Compute();
}
```

You should get the following (or a similar) warning:

```
warning: ignoring return value of 'int Compute()',
declared with attribute nodiscard
```

We can go further and not just mark the return value, but a whole type:

```
[[nodiscard]] struct ImportantType { }
ImportantType CalcSuperImportant();
ImportantType OtherFoo();
ImportantType Calculate();
```

and you'll get a warning whenever you call any function that returns `ImportantType`.

In other words, you can enforce the code contract for a function, so that the caller won't skip the returned value. Sometimes such omission might cause you a bug, so using `[[nodiscard]]` will improve code safety.

The compiler will generate a warning, but usually it's a good practice to enable "treat warnings as errors" when building the code. `/WX` in MSVC or `-Werror` in GCC. Errors stop the compilation process, so a programmer needs to take some action and fix the code.

Where Can It Be Used?

Attributes are a standardised way of annotating the code. They are optional, but they might help the compiler to optimize code, detect possible errors or just clearly express the programmer's intentions.

Here are a few places where `[[nodiscard]]` can be potentially handy:

Errors

One crucial use case for `[[nodiscard]]` are error codes.

How many times have you forgotten to check a returned error code from a function? (Crucial if you don't rely on exceptions).

Here's some code:

```
enum class [[nodiscard]] ErrorCode {
    OK,
    Fatal,
    System,
    FileIssue
};
```

And if we have several functions:

```
ErrorCode OpenFile(std::string_view fileName);
ErrorCode SendEmail(std::string_view sendto,
                    std::string_view text);
ErrorCode SystemCall(std::string_view text);
```

Now every time you'd like to call such functions, you're "forced" to check the return value.

Often you might see code where a developer checks only a few function calls, while other function invocations are left unchecked. That creates inconsistencies and can lead to some severe runtime errors.

You think your method is doing fine (because N (of M) called functions returned `OK`), but something still is failing. You verify it with the debugger, and you notice that the Y function returns `FAIL`, and you haven't checked it.

Should you use `[[nodiscard]]` to mark the error type or maybe some essential functions only?

For error codes that are visible through the whole application that might be the right thing to do. Of course when your function returns just `bool` then you can only mark the function, and not the type (or you can create a `typedef`/alias and then mark it with `[[nodiscard]]`).

Factories / Handles

Another important type of functions where `[[nodiscard]]` adds value are "factories".

Every time you call "make/create/build" you don't want to skip the returned value. Maybe that's a very obvious thing, but there's a possibility (especially when doing some refactoring), to forget, or comment out.

```
[[nodiscard]] Foo MakeFoo();
```

When Returning Non-Trivial Types?

What about such code:

```
std::vector<std::string> GenerateNames();
```

The returned type seems to be `heavy`, so usually, it means that you have to use it later. On the other hand, even `int` might be heavy regarding semantics of the given problem.

Code With No Side Effects?

The code in the previous section:

```
std::vector<std::string> GenerateNames(); // no side effects...
```

This is also an example of a function with no side effects - no global state is changed during the call. In that case, we need to do something with the returned value. Otherwise, the function call can be removed/optimised from the code.

Everywhere?!

There's a paper that might be a "guide" P0600R0 - [[nodiscard]] in the Library[1]. The proposal didn't make into C++17 but was voted into C++20. It suggests a few places were the attribute should be applied.

> **For existing API's:**
>
> - not using the return value always is a "huge mistake" (e.g. always resulting in resource leak)
> - not using the return value is a source of trouble and easily can happen (not obvious that something is wrong)
>
> **For new API's (not been in the C++ standard yet):**
>
> - not using the return value is usually an error.

Here are a few examples where the new attribute should be added:

- `malloc()/new/allocate` - expensive call, usually not using the return value is a resource leak
- `std::async()` - not using the return value makes the call synchronous, which might be hard to detect.

On the other hand such function as `top()` is questionable, as "not very useful, but no danger and such code might exist"

It's probably a good idea not to add `[[nodiscard]]` in all places of your code but focus on the critical places. Possibly, error codes and factories are a good place to start.

How to Ignore `[[nodiscard]]`

There might be rare situations where you might want to surpress "unused variable" warnings. To do that you can use another attribute from C++17: `[[maybe_unused]]`:

[1]https://wg21.link/p0600r0

```
[[nodiscard]] int Compute() { return 42; }
[[maybe_unused]] auto t = Compute();
```

Also, as described in the Attributes Chapter, you can cast the function call to `void` and the the compiler will think you "used" the value:

```
[[nodiscard]] int Compute();
static_cast<void>(Compute()); // used
```

Another good alternative might be to write a separate function that wraps the results and pretends to do use it[2]:

```
template <class T> inline void discard_on_purpose(T&&) {}
discard_on_purpose(Compute());
```

Be careful with the techniques to avoid warnings with `[[nodiscard]]`. It's better to follow the rules of the attribute rather than artificially prevent them.

Before C++17

Most of the attributes that went into the standardised `[[attrib]]` come from compiler extensions, same happened with `[[nodiscard]]`.

For example, in GCC/Clang, there's: `__attribute__((warn_unused_result))`

MSVC offers `_Check_return_` - see at MSDN: Annotating Function Behavior[3].

Summary

To sum up: `[[nodiscard]]` is an excellent addition to all the important code: public APIs, safety-critical systems, etc. Adding this attribute will at least enforce the code contract, and a compiler will help you detect bugs - at compile-time, rather than finding it in runtime.

[2]Suggested by Arne Mertz
[3]https://msdn.microsoft.com/en-us/library/jj159529.aspx

19. Replacing `enable_if` with `if constexpr` - Factory with Variable Arguments

One of the most powerful language features that we get with C++17 is the compile-time if in the form of `if constexpr`. It allows you to check, at compile-time, a condition and depending on the result the code is rejected from the further steps of the compilation.

In this chapter, you'll see one example of how this new feature can simplify the code.

The Problem

In the item 18 of Effective Modern C++ Scott Meyers described a method called makeInvestment:

```
template<typename... Ts>
std::unique_ptr<Investment>
makeInvestment(Ts&&... params);
```

There's a factory method that creates derived classes of Investment, and the main advantage is that it supports a variable number of arguments!

For example, here are the proposed derived types:

Chapter If Constexpr Factory/variable_factory.cpp

```
// base type:
class Investment {
public:
    virtual ~Investment() { }

    virtual void calcRisk() = 0;
};

class Stock : public Investment {
public:
    explicit Stock(const std::string&) { }

    void calcRisk() override { }
};

class Bond : public Investment {
public:
    explicit Bond(const std::string&, const std::string&, int) { }

    void calcRisk() override { }
};

class RealEstate : public Investment {
public:
    explicit RealEstate(const std::string&, double, int) { }
    void calcRisk() override { }
};
```

The code from the book was too idealistic, and it worked until all your classes have the same number and types of input parameters:

Scott Meyers: Modification History and Errata List for Effective Modern C++[1]:

> The makeInvestment interface is unrealistic because it implies that all derived object types can be created from the same types of arguments. This is especially apparent in the sample implementation code, where arguments are perfect-forwarded to all derived class constructors.

For example, if you had a constructor that needed two arguments and one constructor with three arguments, then the code might not compile:

```
// pseudo code:
Bond(int, int, int) { }
Stock(double, double) { }
make(args...)
{
  if (bond)
     new Bond(args...);
  else if (stock)
     new Stock(args...)
}
```

Now, if you write `make(bond, 1, 2, 3)` - then the `else` statement won't compile - as there no `Stock(1, 2, 3)` available! To make it work, we need a compile time if statement that rejects parts of the code that don't match a condition.

On my blog, with the help of one reader, we proposed one working solution (you can read more in Bartek's coding blog: Nice C++ Factory Implementation 2[2]).

[1]http://www.aristeia.com/BookErrata/emc++-errata.html
[2]http://www.bfilipek.com/2016/03/nice-c-factory-implementation-2.html

Here's the code that works:

```
template <typename... Ts>
unique_ptr<Investment>
makeInvestment(const string &name, Ts&&... params)
{
    unique_ptr<Investment> pInv;

    if (name == "Stock")
        pInv = constructArgs<Stock, Ts...>(forward<Ts>(params)...);
    else if (name == "Bond")
        pInv = constructArgs<Bond, Ts...>(forward<Ts>(params)...);
    else if (name == "RealEstate")
        pInv = constructArgs<RealEstate, Ts...>(forward<Ts>(params)...);

    // call additional methods to init pInv...

    return pInv;
}
```

As you can see the "magic" happens inside `constructArgs` function.

The main idea is to return `unique_ptr<Type>` when `Type` is constructible from a given set of attributes and `nullptr` when it's not.

Before C++17

In the previous solution (pre C++17) `std::enable_if` had to be used:

```
// before C++17
template <typename Concrete, typename... Ts>
enable_if_t<is_constructible<Concrete, Ts...>::value, unique_ptr<Concrete>>
constructArgsOld(Ts&&... params)
{
    return std::make_unique<Concrete>(forward<Ts>(params)...);
}

template <typename Concrete, typename... Ts>
enable_if_t<!is_constructible<Concrete, Ts...>::value, unique_ptr<Concrete> >
constructArgsOld(...)
{
    return nullptr;
}
```

`std::is_constructible` - allows us to test if a list of arguments could be used to create a given type.

Just a quick reminder about `enable_if`

`enable_if` (and `enable_if_t` since C++14). It has the following syntax:

```
template< bool B, class T = void >
struct enable_if;
```

`enable_if` will evaluate to `T` if the input condition `B` is true. Otherwise, it's SFINAE and a particular function overload is removed from the overload set.

What's more, in C++17 there's a helper:

```
is_constructible_v = is_constructible<T, Args...>::value;
```

Potentially, the code should be a bit shorter.

Still, using `enable_if` looks ugly and complicated. How about C++17 version?

With `if constexpr`

Here's the updated version:

```
template <typename Concrete, typename... Ts>
unique_ptr<Concrete> constructArgs(Ts&&... params)
{
   if constexpr (is_constructible_v<Concrete, Ts...>)
      return make_unique<Concrete>(forward<Ts>(params)...);
   else
       return nullptr;
}
```

We can even extend it with some little logging features, using fold expression:

```
template <typename Concrete, typename... Ts>
std::unique_ptr<Concrete> constructArgs(Ts&&... params)
{
    cout << __func__ << ": ";
    // fold expression:
    ((cout << params << ", "), ...);
    cout << '\n';

    if constexpr (std::is_constructible_v<Concrete, Ts...>)
        return make_unique<Concrete>(forward<Ts>(params)...);
    else
       return nullptr;
}
```

All the complicated syntax of `enable_if` went away; we don't even need a function overload for the `else` case. We can now wrap expressive code in just one function.

`if constexpr` evaluates the condition and only one block will be compiled. In our case, if a type is constructible from a given set of attributes, then we'll compile `make_unique` call. If not, then `nullptr` is returned (and `make_unique` is not even compiled).

You can play with the code in:

`Chapter If Constexpr Factory/variable_factory.cpp`

Summary

In this chapter, you've seen how `if constexpr` can make code much clearer and more expressive. Before C++17, you could use `enable_if` techniques (SFINAE) or tag dispatching. Those options usually generated complicated code which might be hard to read by novice and non-meta-programming experts. `if constexpr` lowers the expertise level needed to write template code effectively.

20. How to Parallelise CSV Reader

In the Parallel Algorithms chapter, we learned how to speed up code by running it automatically on multiple threads. That chapter showed a few smaller examples and benchmarks. It would also be a good idea to see more extensive applications and how they can benefit from parallelisation, so that's where we turn to now.

In the next pages, you'll see how to build a tool that works on CSV files, parses lines into sales records and then performs calculations on the data. You'll see how easy it is to add parallel execution to selected algorithms and have a performance improvement across the whole application. In the end, we'll discuss problems that we found along the way and possible future enhancements.

In this chapter, you'll learn:

- How to build an application that loads CSV files
- How to efficiently use parallel algorithms
- How to use `std::filesystem` library to gather required files
- How to use other C++17 library features like `std::optional`, conversion routines - `std::from_chars` and `string_view`

Introduction and Requirements

Imagine you're working with some sales data and one task is to calculate a sum of orders for some products. Your shopping system is elementary, and instead of a database, you have CSV files with the order data. There's one file per product.

Take this example of book sales:

date	coupon code	price	discount	quantity
5-12-2018		10.0	0	2
5-12-2018		10.0	0	1
6-12-2018	Santa	10.0	0.25	1
7-12-2018		10.0	0	1

Each line shows a book sale on a specific date. For example, 5th Dec there were three sales, 10$ each, and one person bought two books. On 6th Dec we had one transaction with a coupon code.

The data is encoded as a CSV file: `sales/book.csv`:

```
5-12-2018;;10.0;0;2;
5-12-2018;;10.0;0;1;
6-12-2018;Santa;10.0;0.25;1;
7-12-2018;;10.0;0;1;
```

The application should read the data and then calculate the sum, so in the above case we have

```
sum = 10*2+10*1+       // 5th Dec
      10*(1-0.25)*1 +  // 6th Dec with 25% coupon
      10*1;            // 7th Dec
```

For the above sales data, the final sum is `47.5$`.

Here are the requirements of the application we want to build:

- The application loads all CSV files in a given folder - read from the first argument in the command line
- The files might contain thousands of records but will fit into memory. There's no need to provide extra support for huge files

- Optionally, the application reads the start and end dates from the second and the third command-line argument
- Each CSV line has the following structure:
  - `date;coupon code;unit price;quantity;discount;`
- The application sums all orders between given dates and prints the sum to the standard output

We'll implement the serial version first, and then we'll try to make it parallel.

The Serial Version

For the first step, we'll cover a serial version of the application. This allows you to understand the core parts of the system and see how the tool works.

The code doesn't fit easily on a single page so you can have a look at it in the following file:

`CSV Chapter/csv_reader.cpp`

In the next sections, we'll explore the core parts of the application.

The Main

Let's start with the `main()` function.

CSV Chapter/csv_reader.cpp - main()

```cpp
int main(int argc, const char** argv) {
    if (argc <= 1)
        return 1;

    try {
        const auto paths = CollectPaths(argv[1]);

        if (paths.empty()) {
            std::cout << "No files to process...\n";
            return 0;
        }

        const auto startDate = argc > 2 ? Date(argv[2]) : Date();
        const auto endDate = argc > 3 ? Date(argv[3]) : Date();
15
16          const auto results = CalcResults(paths, startDate, endDate);
17
18          ShowResults(startDate, endDate, results);
19      }
20      catch (const std::filesystem::filesystem_error& err) {
21          std::cerr << "filesystem error! " << err.what() << '\n';
22      }
23      catch (const std::runtime_error& err) {
24          std::cerr << "runtime  error! " << err.what() << '\n';
25      }
26
27      return 0;
28  }
```

Once we're sure that there are enough arguments in the command line, we enter the main scope where all the processing happens:

- line 6 - gather all the files to process - in `CollectPaths()`
- line 16 - convert data from the files into record data and calculate the results - in `CalcResults()`
- line 18 - show the results on the output - in `ShowResults()`

The code relies on exceptions across the whole application.

The paths are collected using `directory_iterator` from the `std::filesystem` library:

CSV Chapter/csv_reader.cpp - CollectPaths()

```
bool IsCSVFile(const fs::path &p) {
    return fs::is_regular_file(p) && p.extension() == CSV_EXTENSION;
}

[[nodiscard]] std::vector<fs::path> CollectPaths(const fs::path& startPath) {
    std::vector<fs::path> paths;
    fs::directory_iterator dirpos{ startPath };
    std::copy_if(fs::begin(dirpos), fs::end(dirpos), std::back_inserter(paths),
                 IsCSVFile);
    return paths;
}
```

As in other filesystem examples, the namespace `fs` is an alias for `std::filesystem`.

With `directory_iterator` we can easily iterate over a given directory. By using `copy_-if`, we can filter out unwanted files and select only those with a CSV extension. Notice how easy it is to get the elements of the path and check files' properties.

Going back to `main()`, we check if there are any files to process (line 8).

Then, in lines 13 and 14, we parse the optional dates: `startDate` and `endDate` are read from `argv[2]` and `argv[3]`.

The dates are stored in a helper class `Date` that lets you convert from strings with a simple format of `Day-Month-Year` or `Year-Month-Day`. The class also supports comparison of dates. This will help us check whether a given order fits between selected dates.

Now, all of the computations and printouts are contained in lines:

```
const auto results = CalcResults(paths, startDate, endDate);
ShowResults(results, startDate, endDate);
```

`CalcResults()` implements the core requirements of the application:

- converting data from the file into a list of records to process
- calculating a sum of records between given dates

CSV Chapter/csv_reader.cpp - CalcResults()

```
struct Result {
    std::string mFilename;
    double mSum{ 0.0 };
};

[[nodiscard]] std::vector<Result>
CalcResults(const std::vector<fs::path>& paths, Date startDate, Date endDate) {
    std::vector<Result> results;
    for (const auto& p : paths) {
        const auto records = LoadRecords(p);

        const auto totalValue = CalcTotalOrder(records, startDate, endDate);
        results.push_back({ p.string(), totalValue });
    }
    return results;
}
```

The code loads records from each CSV file, then calculates the sum of those records. The results (along with the name of the file) are stored in the output vector.

We can now reveal the code behind the two essential methods `LoadRecords` and `CalcTotalOrder`.

Converting Lines into Records

`LoadRecords` is a function that takes a filename as an argument, reads the contents into `std::string` and then performs the conversion:

CSV Chapter/csv_reader.cpp - LoadRecords()

```
[[nodiscard]] std::vector<OrderRecord> LoadRecords(const fs::path& filename) {
    const auto content = GetFileContents(filename);

    const auto lines = SplitLines(content);

    return LinesToRecords(lines);
}
```

We assume that the files are small enough to fit into RAM, so there's no need to process them in chunks.

The core task is to split that one large string into lines and then convert them into a collection of Records.

If you look into the code, you can see that `content` is `std::string`, but `lines` is a vector of `std::string_view`. Views are used for optimisation. We guarantee to hold the large string - the file content - while we process chunks of it (views). This should give us better performance, as there's no need to copy string data.

Eventually, characters are converted into `OrderRecord` representation.

The `OrderRecord` Class

The main class that is used to compute results is `OrderRecord`. It's a direct representation of a line from a CSV file.

CSV Chapter/csv_reader.cpp - OrderRecord

```
class OrderRecord {
public:
    // constructors...

    double CalcRecordPrice() const noexcept;
    bool CheckDate(const Date& start, const Date& end) const noexcept;

private:
    Date mDate;
    std::string mCouponCode;
    double mUnitPrice{ 0.0 };
    double mDiscount{ 0.0 }; // 0... 1.0
    unsigned int mQuantity{ 0 };
};
```

The conversion

Once we have lines we can convert them one by one into objects:

CSV Chapter/csv_reader.cpp - LinesToRecord()

```
[[nodiscard]] std::vector<OrderRecord>
LinesToRecords(const std::vector<std::string_view>& lines) {
    std::vector<OrderRecord> outRecords;
    std::transform(lines.begin(), lines.end(),
                   std::back_inserter(outRecords), LineToRecord);

    return outRecords;
}
```

The code above is just a transformation, it uses `LineToRecord` to do the hard work:

CSV Chapter/csv_reader.cpp - LineToRecord()

```
[[nodiscard]] OrderRecord LineToRecord(std::string_view sv) {
    const auto cols = SplitString(sv, CSV_DELIM);
    if (cols.size() == static_cast<size_t>(OrderRecord::ENUM_LENGTH)) {
        const auto unitPrice = TryConvert<double>(cols[OrderRecord::UNIT_PRICE]);
        const auto discount = TryConvert<double>(cols[OrderRecord::DISCOUNT]);
        const auto quantity = TryConvert<unsigned int>(cols[OrderRecord::QUANTITY]);

        if (unitPrice && discount && quantity) {
            return { Date(cols[OrderRecord::DATE]),
                     std::string(cols[OrderRecord::COUPON]),
                     *unitPrice,
                     *discount,
                     *quantity };
        }
    }
    throw std::runtime_error("Cannot convert Record from " + std::string(sv));
}
```

Firstly, the line is split into columns, and then we can process each column.

If all elements are converted, then we can build a record.

For conversions of the elements we're using a small utility based on `std::from_chars`:

CSV Chapter/csv_reader.cpp - TryConvert()

```
template<typename T>
[[nodiscard]] std::optional<T> TryConvert(std::string_view sv) noexcept {
    T value{ };
    const auto last = sv.data() + sv.size();
    const auto res = std::from_chars(sv.data(), last, value);
    if (res.ec == std::errc{} && res.ptr == last)
        return value;

    return std::nullopt;
}
```

`TryConvert` uses `std::from_chars` and returns a converted value if there are no errors. As you remember, to guarantee that all characters were parsed, we also have to check `res.ptr == last`. Otherwise, the conversion might return success for input like "123xxx".

Calculations

Once all the records are available we can compute their sum:

CSV Chapter/csv_reader.cpp - CalcTotalOrder()

```
[[nodiscard]] double CalcTotalOrder(const std::vector<OrderRecord>& records,
                                    const Date& startDate, const Date& endDate) {
    return std::accumulate(std::begin(records), std::end(records), 0.0,
        [&startDate, &endDate](double val, const OrderRecord& rec) {
            if (rec.CheckDate(startDate, endDate))
                return val + rec.CalcRecordPrice();
            else
                return val;
        }
    );
}
```

The code runs on the vector of all records and then calculates the price of each element if they fit between `startDate` and `endDate`. Then they are all summed in `std::accumulate`.

Design Enhancements

The application calculates only the sum of orders, but we could think about adding other things. For example, minimal value, maximum, average order and other statistics.

The code uses a simple approach, loading a file into a string and then creating a temporary vector of lines. We could also enhance this by using a line iterator. It would take a large string and then return a line when you iterate.

Another idea relates to error handling. For example, rather than throwing exceptions, we could enhance the conversion step by storing the number of successfully processed records.

Running the Code

The application is ready to compile, and we can run it on the example data shown in the introduction.

```
CSVReader.exe sales/
```

This should read a single file `sales/book.csv` and sum up all the records (as no dates were specified):

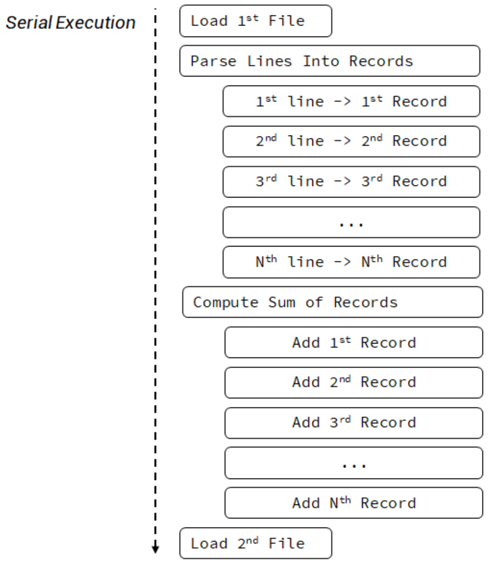
```
.\CalcOrdersSerial.exe .\sales\
Name Of File     | Total Orders Value
sales\book.csv   | 47.50
CalcResults: 3.13 ms
CalcTotalOrder: 0.01 ms
Parsing Strings: 0.01 ms
```

The full version of the code also includes timing measurement, so that's why you can see that the operation took around 3ms to complete. The file handling took the longest; calculations and parsing were almost immediate.

In the next sections, you'll see a few simple steps you can take to apply parallel algorithms.

Using Parallel Algorithms

Previously the code was executed sequentially. We can illustrate it in the following diagram:



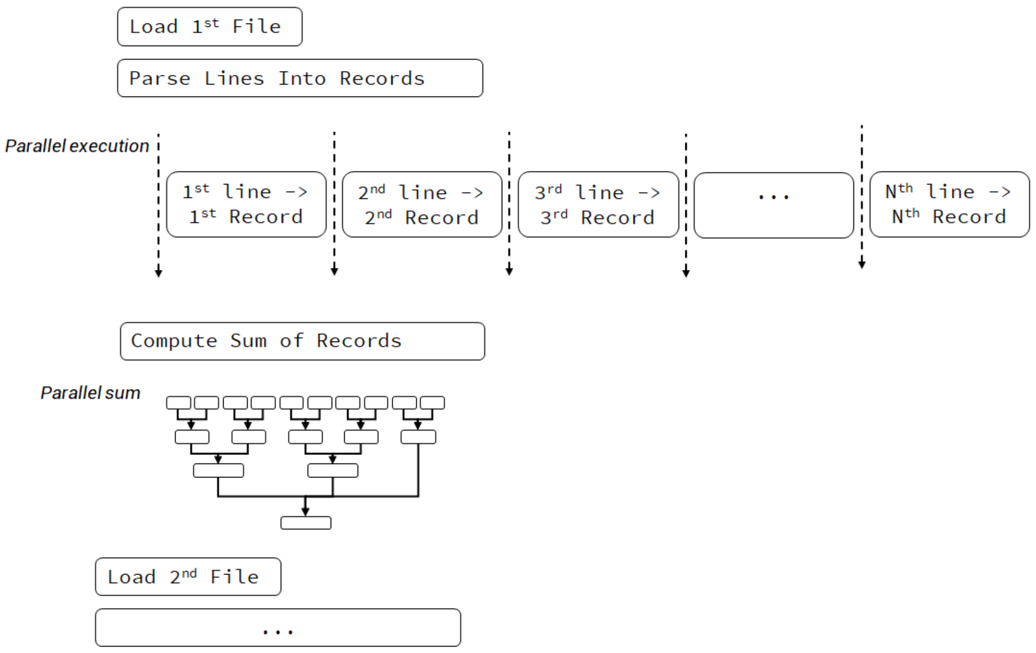
Serial Execution of CSV Reader

We open each file, process it, calculate, then we go to another file. All this happens on a single thread.

However, there are several places we can consider using parallel algorithms:

- Where each file can be processed separately
- Where each line of a file can be converted independently into the Record Data
- Where calculations can be enhanced with parallel execution

If we focus on the second and the third options, we can move into the following execution model:



Parallel Execution of CSV Reader

The above diagram shows that we're still processing file one by one, but we use parallel execution while parsing the strings and making the calculations.

When doing the conversion, we have to remember that exceptions won't be re-thrown from our code. Only `std::terminate` will be called.

As of July 2019, only the MSVC compiler (since Visual Studio 2017) and GCC (since 9.1) support parallel execution in the Standard Library. The parallel version of the example does not work with Clang. It's possible to use a third-party library like Intel Parallel STL or HPX.

Data Size & Instruction Count Matters

How to get the best performance with parallel algorithms?

You need two things:

- a lot of data to process
- instructions to keep the CPU busy

We also have to remember one rule:

> In general, parallel algorithms do more work, as they introduce the extra cost of managing the parallel execution framework as well as splitting tasks into smaller batches.

First and foremost, we have to think about the size of the data we're operating on. If we have only a few files, with a few dozen records, then we may not gain anything with parallel execution. But if we have lots of files, with hundreds of lines each, then the potential might increase.

The second thing is the instruction count. CPU cores need to compute and not just wait on memory. If your algorithms are memory-bound, then parallel execution might not give any speed-up over the sequential version. In our case, it seems that the parsing strings task is a good match here. The code performs searching on strings and does the numerical conversions, which keeps CPU busy.

Parallel Data Conversion

As previously discussed, we can add parallel execution to the place where we convert the data. We have lots of lines to parse, and each parsing is independent.

CSV Chapter/csv_reader.cpp - LinesToRecord()

```
[[nodiscard]] std::vector<OrderRecord>
LinesToRecords(const std::vector<std::string_view>& lines) {
    std::vector<OrderRecord> outRecords(lines.size());
    std::transform(std::execution::par, std::begin(lines), std::end(lines),
                   std::begin(outRecords), LineToRecord);

    return outRecords;
}
```

Two things need to be changed to the serial version:

- we need to preallocate the vector
- we have to pass `std::execution::par` (or `par_unseq`) as the first argument

The serial code also used `std::transform`, so why cannot we just pass the execution parameter?

We can even compile it… but you should see an error like:

`Parallel algorithms require forward iterators or stronger.`

The reason is simple: `std::back_inserter` is very handy, but it's not a forward iterator. It inserts elements into the vector, and that causes a vector to be changed (reallocated) by multiple threads. All of the insertions would have to be guarded by some critical section, and thus the overall performance could be weak.

Since we need to preallocate the vector, we have to consider two things:

- we pay for default construction of objects inside a vector, which probably isn't a big deal when objects are relatively small, and their creation is fast.
- on the other hand, the vector is allocated once, and there's no need to grow it (copy, reallocate) as in the case of `std::back_inserter`.

Parallel Calculations

Another place where we can leverage parallel algorithms is `CalcTotalOrder()`.

Instead of `std::accumulate` we can use `std::transform_reduce`.

As mentioned in the Parallel Algorithms chapter, the floating-point sum operation is not associative. However, in our case, the results should be stable enough to give 2 decimal places of precision. If you need better accuracy and numerical stability, you may be better off using a different method.

CSV Chapter/csv_reader.cpp - CalcTotalOrder()

```
double CalcTotalOrder(const std::vector<OrderRecord>& records,
                      const Date& startDate, const Date& endDate) {
    return std::transform_reduce(
        std::execution::par,
        std::begin(records), std::end(records),
        0.0,
        std::plus<>(),
        [&startDate, &endDate](const OrderRecord& rec) {
            if (rec.CheckDate(startDate, endDate))
                return rec.CalcRecordPrice();

            return 0.0;
        }
    );
}
```

We use the `transform` step of `std::transform_reduce` to “extract” values to sum. We cannot easily use `std::reduce` as it would require us to write a reduction operation that works with two `OrderRecord` objects.

Tests

We can run the two versions on a set of files and compare if the changes brought any improvements in the performance. The application was tested on a 6 core/12 thread PC - i7 8700, with a fast SSD drive, Windows 10.

Our applications access files, so it’s harder to make accurate benchmarks as we can quickly end up in the file system cache. Before major runs of applications, a tool called Use SysInternal’s RAMMap app[1] is executed to remove files from the cache. There are also Hard Drive hardware caches which are harder to release without a system reboot.

[1]http://technet.microsoft.com/en-us/sysinternals/ff700229.aspx

Mid Size Files 1k Lines 10 Files

Let's start with 10 files, 1k lines each. Files are not in the OS cache:

Step	Serial (ms)	Parallel (ms)
All steps	74.05	68.391
`CalcTotalOrder`	0.02	0.22
Parsing Strings	7.85	2.82

The situation when files are in the system cache:

Step	Serial (ms)	Parallel (ms)
All steps	8.59	4.01
`CalcTotalOrder`	0.02	0.23
Parsing Strings	7.74	2.73

The first numbers - 74ms and 68ms - come from reading uncached files, while the next two runs were executed without clearing the system cache so you can observe how much speed-up you get by system caches.

The parallel version still reads files sequentially, so we only get a few milliseconds of improvement. Parsing strings (line split and conversion to Records) is now almost 3x faster. The sum calculations are not better as a single-threaded version seem to handle sums more efficiently.

Large Set 10k Lines in 10 Files

How about larger input?

Uncached files:

Step	Serial (ms)	Parallel (ms)
All steps	239.96	178.32
`CalcTotalOrder`	0.2	0.74
Parsing Strings	70.46	15.39

Cached:

Step	Serial (ms)	Parallel (ms)
All steps	72.43	18.51
`CalcTotalOrder`	0.33	0.67
Parsing Strings	70.46	15.56

The more data we process, the better our results. The cost of loading uncached files "hides" slowly behind the time it takes to process the records. In the case of 10k lines, we can also see that the parsing strings step is 3.5 times faster; however, the calculations are still slower.

Largest Set 100k Lines in 10 Files

Let's do one more test with the largest files:

Uncached files:

Step	Serial (ms)	Parallel (ms)
All steps	757.07	206.85
`CalcTotalOrder`	3.03	2,47
Parsing Strings	699.54	143.31

Cached:

Step	Serial (ms)	Parallel (ms)
All steps	729.94	162.49
`CalcTotalOrder`	3.05	2.16
Parsing Strings	707.34	141.28

In a case of large files (each file is ∼2MB), we can see a clear win for the parallel version.

Wrap up & Discussion

The main aim of this chapter was to show how easy it is to use parallel algorithms.

The final code is located in two files:

`Chapter CSV Reader/csv_reader.cpp` and `Chapter CSV Reader/csv_reader_-par.cpp` for the parallel version.

In most of the cases, all we have to do to add parallel execution is to make sure there's no synchronisation required between the tasks and, if we can, provide forward iterators. That's why when doing the conversion we sometimes needed to preallocate `std::vector` (or other compliant collections) rather than using `std::back_inserter`. Another example is that we cannot iterate in a directory in parallel, as `std::filesystem::directory_-iterator` is not a forward iterator.

The next part is to select the proper parallel algorithm. In the case of this example, we replaced `std::accumulate` with `std::transform_reduce` for the calculations. There was no need to change `std::transform` for doing the string parsing - as you only have to use the extra `execution policy` parameter.

Our application performed a bit better than the serial version. Here are some thoughts we might have:

- Parallel execution needs independent tasks. If you have jobs that depend on each other, the performance might be lower than the serial version! This happens due to extra synchronisation steps.
- Your tasks cannot be memory-bound, otherwise CPU will wait for the memory. For example, the string parsing code performed better in parallel as it has many instructions to execute: string search, string conversions.
- You need a lot of data to process to see the performance gain. In our case, each file required several thousands of lines to show any gains over the sequential version.
- Sum calculations didn't show much improvement and there was even worse performance for smaller input. This is because the `std::reduce` algorithm requires extra reduction steps, and also our calculations were elementary. It's possible that, with more statistical computations in the code, we could improve performance.
- The serial version of the code is straightforward and there are places where extra performance could be gained. For example, we might reduce additional copies and temporary vectors. It might also be good to use `std::transform_reduce` with sequential execution in the serial version, as it might be faster than `std::accumulate`. You might consider optimising the serial version first and then making it parallel.
- If you rely on exceptions then you might want to implement a handler for `std::terminate`, as exceptions are not re-thrown in code that is invoked with execution policies.

Putting it all together, we can draw the following summary:

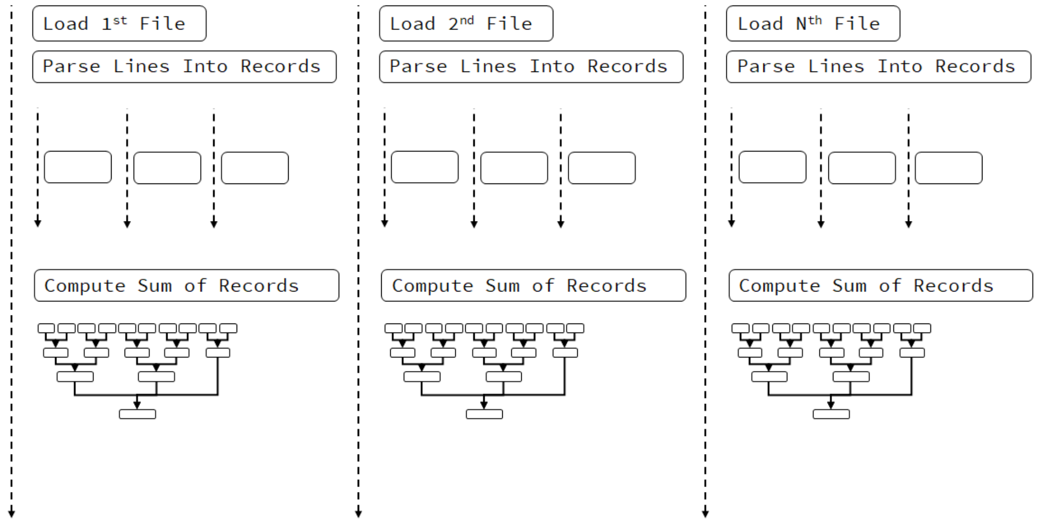
> *Parallel algorithms can bring extra speed to the application, but they have to be used wisely. They introduce an additional cost of parallel execution framework, and it's essential to have lots of tasks that are independent and good for parallelisation. As always, it's important to measure the performance between the versions to be able to select the final approach with confidence.*

Are there any other options to improve the project? Let's see a few other possibilities on the next page.

Additional Modifications and Options

The code in the parallel version skipped one option: parallel access to files. So far we read files one by one, but how about reading separate files from separate threads?

Here's a diagram that illustrates this option:



Parallel Execution of CSV Reader, Reading files in separate threads

In the above diagram, the situation is a bit complicated. If we assume that OS cannot handle multiple file access, then threads will wait on files. But once the files are available, the processing might go in parallel.

If you want to play around with this technique, you can replace `std::execution::seq` in `CalcResults()` with `std::execution::par`. That will allow the compiler to run `LoadRecords()` and `CalcTotalOrder()` in parallel.

Is your system capable of accessing files from separate threads?

In general, the answer might be tricky, as it depends on many elements: hardware, system, and cost of computations, etc. For example, on a machine with a fast SSD drive, the system can handle several files reads, while on a HDD drive, the performance might be slower. Modern drives also use Native Command Queues, so even if you access from multiple threads, the command to the drive will be serial and also rearranged into a more optimal way. We leave the experiments to the readers as this topic goes beyond the scope of this book.

Appendix A - Compiler Support

If you work with the latest version of a compiler like GCC, Clang or MSVC you may assume that C++17 is fully supported (with some exceptions to the STL implementations). GCC implemented the full support in the version 7.0, Clang did it in the version 6.0 and MSVC is conformant as of VS 2017 15.7 For completeness, here you have a list of features and versions of compilers where it was added.

The main resource to be up to date with the status of the features: CppReference - Compiler Support[2]

GCC

- Language[3]
- The Library - LibSTDC++[4]

Clang

- Language[5]
- The Library - LibC++[6]

VisualStudio - MSVC

- Announcing: MSVC Conforms to the C++ Standard[7]
- C++17/20 Features and Fixes in Visual Studio 2019[8]
- STL Features and Fixes in VS 2017 15.8[9]

[2]https://en.cppreference.com/w/cpp/compiler_support
[3]https://gcc.gnu.org/projects/cxx-status.html#cxx17
[4]https://gcc.gnu.org/onlinedocs/libstdc++/manual/status.html#status.iso.2017
[5]http://clang.llvm.org/cxx_status.html#cxx17
[6]http://libcxx.llvm.org/cxx1z_status.html
[7]https://blogs.msdn.microsoft.com/vcblog/2018/05/07/announcing-msvc-conforms-to-the-c-standard/
[8]https://devblogs.microsoft.com/cppblog/cpp17-20-features-and-fixes-in-vs-2019/
[9]https://devblogs.microsoft.com/cppblog/stl-features-and-fixes-in-vs-2017-15-8/

Compiler Support of C++17 Features

Fixes and Deprecation

Feature	GCC	Clang	MSVC
Removing `register` keyword	7.0	3.8	VS 2017 15.3
Remove Deprecated `operator++`(bool)	7.0	3.8	VS 2017 15.3
Removing Deprecated Exception Specifications	7.0	4.0	VS 2017 15.5
Removing trigraphs	5.1	3.5	VS 2010
New auto rules for direct-list-initialisation	5.0	3.8	VS 2015
`static_assert` with no message	6.0	2.5	VS 2017
Different begin and end types in range-based for	6.0	3.6	VS 2017

Clarification

Feature	GCC	Clang	MSVC
Stricter expression evaluation order	7.0	4.0	VS 2017
Guaranteed copy elision	7.0	4.0	VS 2017 15.6
Exception specifications part of the type system	7.0	4.0	VS 2017 15.5
Dynamic memory allocation for over-aligned data	7.0	4.0	VS 2017 15.5

General Language Features

Feature	GCC	Clang	MSVC
Structured Binding Declarations	7.0	4.0	VS 2017 15.3
Init-statement for if/switch	7.0	3.9	VS 2017 15.3
Inline variables	7.0	3.9	VS 2017 15.5
`constexpr` Lambda Expressions	7.0	5.0	VS 2017 15.3
Lambda Capture of `*this`	7.0	3.9	VS 2017 15.3
Nested namespaces	6.0	3.6	VS 2015
`has_include`	5	Yes	VS 2017 15.3

Templates

Feature	GCC	Clang	MSVC
Template argument deduction for class templates	7.0/8.0[10]	5.0	VS 2017 15.7
Deduction Guides in the Standard Library	8.0[11]	7.0/in progress[12]	VS 2017 15.7
Declaring non-type template parameters with auto	7.0	4.0	VS 2017 15.7
Fold expressions	6.0	3.9	VS 2017 15.5
`if constexpr`	7.0	3.9	VS 2017

Attributes

Feature	GCC	Clang	MSVC
`[[fallthrough]]`	7.0	3.9	VS 2017 15.0
`[[nodiscard]]`	7.0	3.9	VS 2017 15.3
`[[maybe_unused]]`	7.0	3.9	VS 2017 15.3
Attributes for namespaces and enumerators	4.9(namespaces)/6(enums)	3.4	VS 2015 14.0
Ignore unknown attributes	yes	3.9	VS 2015 14.0
Using attribute namespaces without repetition	7.0	3.9	VS 2017 15.3

[10]Additional improvements for Template Argument Deduction for Class Templates happened in GCC 8.0, P0512R0.

[11]Deduction Guides are not listed in the status pages of LibSTDC++, so we can assume they were implemented as part of Template argument deduction for class templates.

[12]The status page for LibC++ mentions that `<string>`, sequence containers, container adaptors and `<regex>` portions have been implemented so far.

The Standard Library

Feature	GCC	Clang	MSVC
`std::optional`	7.1	4.0	VS 2017 15.0
`std::variant`	7.1	4.0	VS 2017 15.0
`std::any`	7.1	4.0	VS 2017 15.0
`std::string_view`	7.1	4.0	VS 2017 15.0
String Searchers	7.1	5.0	VS 2017 15.3
String Conversions	8 (only integral types)	in progress	VS 2017 15.8
Parallel Algorithms	9.1	in progress	VS 2017 15.7
Filesystem	8.0	7.0	VS 2017 15.7

Other STL changes

Feature	GCC	Clang	MSVC
`std::byte`	7.1	5.0	VS 2017 15.3
Improvements for Maps and Sets	7.0	3.9	VS 2017 15.5
`insert_or_assign()`/`try_emplace()` for maps	6.1	3.7	VS 2017 15
Emplace Return Type	7.1	4.0	VS 2017 15.3
Sampling algorithms	7.1	In Progress	VS 2017 15
`gcd` and `lcm`	7.1	4.0	VS 2017 15.3
`clamp`	7.1	3.9	VS 2015.3
Special Mathematical Functions	7.1	Not yet	VS 2017 15.7
Shared Pointers and Arrays	7.1	In Progress	VS 2017 15.5
Non-member `size()`, `data()` and `empty()`	6.1	3.6	VS 2015
`constexpr` Additions to the Standard Library	7.1	4.0	VS 2017 15.3
`scoped_lock`	7.1	5.0	VS 2017 15.3
Polymorphic Allocator & Memory Resource	9.1	In Progress	VS 2017 15.6

Removed Or Deprecated Library Features

Feature	GCC	Clang	MSVC
Removing `auto_ptr`, `random_shuffle`, old `<functional>` stuff	No [13]	not yet	VS 2015
Deprecating `std::iterator`	not yet	not yet	VS 2017 15.5
Deprecating `shared_ptr::unique()`	not yet	not yet	VS 2017 15.5
Deprecating `<codecvt>`	not yet	not yet	VS 2017 15.5
Removing Deprecated Iostreams Aliases	not yet	3.8	VS 2015.2
Deprecate `result_of`	not yet	not yet	VS 2017 15.3
Removing Allocator Support In `std::function`	not yet	4.0	VS 2017 15.5
C++17 should refer to C11 instead of C99	9.1	7.0	VS 2015
Removing Deprecated Iostreams Aliases	not yet	3.8	VS 2015.2

[13]Kept for compatibility.

Appendix B - Resources and References

You can purchase the official C++17 Standard at the ISO site:

ISO/IEC 14882:2017 - Programming languages - C++[14]

However, you can also read the free draft that's very close to the published version:

N4687, 2017-07-30, **Working Draft, Standard for Programming Language C++**[15]

This PDF is from isocpp.org[16]. Also, have a look here for more information about the papers and the status of C++: isocpp/Standard C++[17].

For a quick overview of C++17 changes, here's a handy list located at:

P0636 - Changes between C++14 and C++17[18]

Books:

- **C++17 - The Complete Guide** by Nicolai Josuttis
- **C++17 STL Cookbook** by Jacek Galowicz
- **Modern C++ Programming Cookbook** by Marius Bancila
- **C++ Templates: The Complete Guide** (2nd Edition) by David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor
- **Professional C++, 4th Edition** by Marc Gregoire

General C++ Links:

- Compiler support: C++ compiler support
- ISO Standard C++

[14]https://www.iso.org/standard/68564.html
[15]https://wg21.link/n4687
[16]https://isocpp.org/
[17]https://isocpp.org/std/the-standard
[18]https://wg21.link/P0636

- Jason Turner: C++ Weekly channel, where he covered most (or even all!) of C++17 features.
- Simon Brand blog - with lot's of information about C++17
- Arne Mertz blog
- Rainer Grimm Blog
- CppCast
- FluentC++

General C++17 Language Features

- Simon Brand: Template argument deduction for class template constructors
- Class template deduction(since C++17) - cppreference.
- "Using fold expressions to simplify variadic function templates" in Modern C++ Programming Cookbook.
- Simon Brand: Exploding tuples with fold expressions
- Baptiste Wicht: C++17 Fold Expressions
- Fold Expressions - ModernesCpp.com
- Adding C++17 structured bindings support to your classes
- C++ Weekly Special Edition - Using C++17's constexpr if - YouTube - real examples from Jason and his projects.
- C++17: let's have a look at the constexpr if – FJ
- C++ 17 vs. C++ 14 — if-constexpr – LoopPerfect – Medium
- Two-phase name lookup support comes to MSVC
- What does the `carries_dependency` attribute mean? - Stack Overflow
- Value Categories in C++17 – Barry Revzin – Medium
- Rvalues redefined | Andrzej's C++ blog
- Guaranteed Copy Elision Does Not Elide Copies - MSVC C++ Team Blog

Expression Evaluation Order:

- GotW #56: Exception-Safe Function Calls
- Core Guidelines: ES.43: Avoid expressions with undefined order of evaluation
- Core Guidelines: ES.44: Don't depend on order of evaluation of function arguments

About `std::optional`:

- Andrzej's C++ blog: Efficient optional values
- Andrzej's C++ blog: Ref-qualifiers
- Clearer interfaces with `optional<T>` - Fluent C++
- Optional - Performance considerations - Boost 1.67.0
- Enhanced Support for Value Semantics in C++17 - Michael Park, CppCon 2017
- std::optional: How, when, and why | Visual C++ Team Blog

About `std::variant`:

- SimplifyC++ - Overload: Build a Variant Visitor on the Fly.
- Variant Visitation by Michael Park
- Sum types and state machines in C++17
- Implementing State Machines with std::variant
- Pattern matching in C++17 with std::variant, std::monostate and std::visit
- Another polymorphism | Andrzej's C++ blog
- Inheritance vs std::variant, C++ Truths

About `string_view`

- CppCon 2015 `string_view` — Marshall Clow
- string_view odi et amo - Marco Arena
- C++17 string_view – Steve Lorimer
- Modernescpp - string_view
- J. Müller - std::string_view accepting temporaries: good idea or horrible pitfall?
- abseil / Tip of the Week #1: string_view
- `std::string_view` is a borrow type – Arthur O'Dwyer – Stuff mostly about C++
- C++ Russia 2018: Victor Ciura, Enough string_view to hang ourselves - YouTube
- StringViews, StringViews everywhere! - Marc Mutz - Meeting C++ 2017 - YouTube
- Jacek's C++ Blog · Const References to Temporary Objects
- abseil / Tip of the Week #107: Reference Lifetime Extension
- C++17 - Avoid Copying with std::string_view - ModernesCpp.com

String Conversions and Searchers

- How to Convert a String to an int in C++ - Fluent C++
- How to *Efficiently* Convert a String to an int in C++ - Fluent C++
- How to encode char in 2-bits? - Stack Overflow

About Filesystem

- Chapter 7, "Working with Files and Streams" - of **Modern C++ Programming Cookbook**.
- examples like: Working with filesystem paths, Creating, copying, and deleting files and directories, Removing content from a file, Checking the properties of an existing file or directory, searching.
- Chapter 10 ""Filesystem" from "**C++17 STL Cookbook**"
- examples: path normalizer, Implementing a grep-like text search tool, Implementing an automatic file renamer, Implementing a disk usage counter, statistics about file types, Implementing a tool that reduces folder size by substituting duplicates with symlinks
- C++17- std::byte and std::filesystem - ModernesCpp.com
- How similar are Boost filesystem and the standard C++ filesystem libraries? - SO
- bfilipek.com: Converting from Boost to std::filesystem

Parallel Algorithms

- Bryce Adelstein's talk about parallel algorithms. Contains a lot of examples for map reduce (transform reduce) algorithm: CppCon 2016: Bryce Adelstein Lelbach "The C++17 Parallel Algorithms Library and Beyond" - YouTube
- Sean Parent – Better Code: Concurrency - code::dive 2016
- Simon Brand - std::accumulate vs. std::reduce
- Using C++17 Parallel Algorithms for Better Performance | Visual C++ Team Blog
- bfilipek.com: The Amazing Performance of C++17 Parallel Algorithms, is it Possible?

Index