BARTŁOMIEJ FILIPEK

C++17 IN DETAIL

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

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.

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[1]
- GoodReads Book's Page[2]

Example Code

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

cppindetail.com/data/cpp17indetail.zip[3]

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

[1]https://leanpub.com/cpp17indetail/feedback
[2]https://www.goodreads.com/book/show/41447221-c-17-in-detail
[3]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

```cpp
#include <iostream>

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

Or shorter snippets (without a corresponding file):

```cpp
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)[4]
- The first method of a class is not highlighted - #1084 - First method of class not highlighted in C++[5]
- Template method is not highlighted #1434 - C++ lexer doesn't recognize function if return type is templated[6]
- Modern C++ attributes are sometimes not recognised properly

Other issues for C++ and Pygments: issues C++[7].

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.

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

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

- Coliru[8] - uses GCC 8.2.0 (as of July 2019), offers link sharing and a basic text editor, it's simple but very effective.
- Wandbox[9] - offers a lot of compilers, including most Clang and GCC versions, can use boost libraries; offers link sharing and multiple file compilation.
- Compiler Explorer[10] - offers many compilers, shows compiler output, can execute the code.
- CppBench[11] - runs simple C++ performance tests (using google benchmark library).
- C++ Insights[12] - 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[13].

[8]http://coliru.stacked-crooked.com/
[9]https://wandbox.org/
[10]https://gcc.godbolt.org/
[11]http://quick-bench.com/
[12]https://cppinsights.io/
[13]https://arnemertz.github.io/online-compilers/

1. 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

```
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

```
// 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

```
#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.

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

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

```
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

2. 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)

3. 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

```
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`.

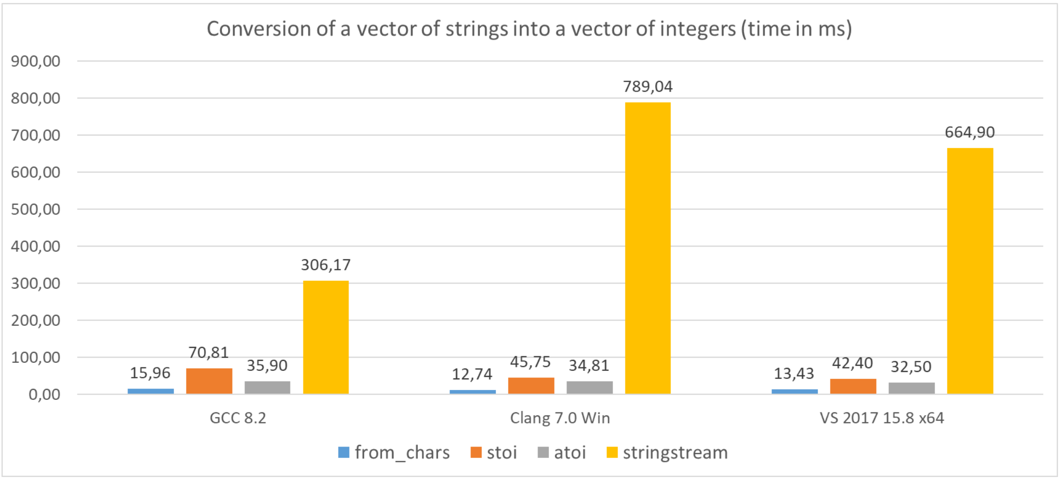
Looking now at `from_chars` :

- 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`!

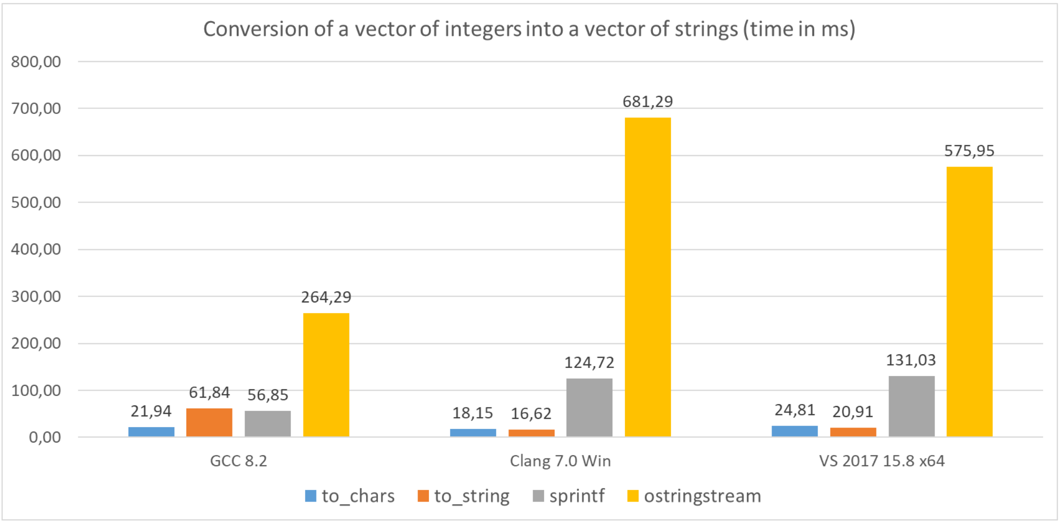
[2]https://nuwen.net/mingw.html
[3]http://releases.llvm.org/dow4

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.