BARTŁOMIEJ FILIPEK

C++17 IN DETAIL

LEARN WHAT ARE THE EXCITING FEATURES OF THE NEW STANDARD!

BFILIPEK.COM

C++17 in Detail

Learn what are 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 2018-08-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

for Wiola and Mikołaj

Contents

About the Author

Bartłomiej Filipek is a C++ software developer with more than 11 years of professional experience. He finished Jagiellonian University in Cracow with a Masters Degree in Computer Science.

Bartek currently works at Xara, where he develops features for advanced document editors. He has also 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 Cracow's Universities.

Since 2011 Bartek has been regularly blogging at his website: bfilipek.com. In the early days the topic revolved around graphics programming, and now the blog focuses on Core C++. He also helps as co-organizer at C++ User Group in Krakow. You can hear Bartek in one @CppCast episode where he talks about C++17, blogging and text processing.

In his spare time, he loves assembling trains and Lego with his little son. And he's a collector of large Lego models.

Preface

After long-awaited C++11, the C++ Committee made changes to the standardisation process and currently we can 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 this words, in the middle of 2018, we're in the process of preparing C++20.

As you see, the language and the Standard Library evolves quite fast! Since 2011 you get a set of new library modules and language features every three years. Thus staying up to date with all the state of the language becomes quite a challenging task, and this is why this book will help you.

The chapters of this book describe all the significant changes in C++17 and will give you essential knowledge to stay at the edge of the latest features. What's more, each section contains lots of practical examples and also compiler-specific notes to give you 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 with discovering C++17 as well!

Best regards

Bartek

1. General Language Features

Having finished the chapters on language fixes and clarifications, we're now ready to look at widespread features. Improvements described in this section also have the potential to make your code more compact and expressive.

For example, with Structured Bindings, you can leverage much easier syntax tuples (and tuple-like expressions). Something that was easy in other languages like Python is also possible with good-old C++!

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 expression that might be used in `constexpr` context

Structured Binding Declarations

Do you often work with tuples or pairs?

If not, then you should probably start looking into that handy types. Not only are tuples (or pairs) suggested for returning multiple values from a function, but they've also got dedicated language support, making code easier and cleaner to write.

If you have a function that returns a few results:

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

You can use `auto ret = InsertElement(...)` and then refer to `ret.first` or `ret.second`. Alternatively you can leverage `std::tie` which will unpack the tuple/pair into custom 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;

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

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

With C++17 the code can be more compact:

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

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

Now, instead of `pair.first` and `pair.second` you can use variables with more concrete names.

And, also you have one line instead of three and the code is easier to read and safer.

The Syntax

This part might be updated to become clearer

The basic syntax 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 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 hidden object member variables.

For example:

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

`x` binds to `int` stored in the hidden object that is a copy of `a`. And similarly `y` bind to a `float` value.

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 above example `x` binds to the element in the hidden object that is a reference to `a`.

You can also add `[[attribute]]`:

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

Structured Bindings or Decomposition Declaration?

For this feature, you might have seen another name "decomposition declaration" in use. During the standardisation process those two names were considered, but now the final version sticks with "Structured Bindings."

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;
```

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

```
auto [a, b] = myPair; // binds myPair.first/second
```

In other words, you can provide support for your classes, assuming you add `get<N>` interface implementation. See an example in the later section.

3. If the initializer's type contains only non static, public members:

```
struct S { int x1 : 2; volatile double y1; };
S f();
const auto [ x, y ] = f();
```

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

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

Note: In C++17, you could use structured bindings to bind to class members as long as they were public. This might be a problem when you want to access private members in the implementation of the class. With C++20 it should be fixed. See P0969R0[1].

Example

One of the **coolest use cases** - binding inside a range based for loop:

```
std::map<KeyType, ValueType> myMap;
for (const auto & [key,val] : myMap)
{
    // use key/value rather than iter.first/iter.second
}
```

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 is a pair `<first, second>`. Using the real names `key/value` is more expressive than the pair.

The above technique can be used in:

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

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

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

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

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:

```
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:

```
// 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> : std::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 simpler 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.

As you've seen `if constexpr` was used to implement `get<N>` functions, read more in the `if constexpr` chapter.

Extra Info

The change was proposed in: P0217[2](wording), P0144[3](reasoning and examples), P0615[4] (renaming "decomposition declaration" with "structured binding declaration").

[2] https://wg21.link/p0217
[3] https://wg21.link/p0144
[4] 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 and then check it 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...
}
```

Look, 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...
```

Notice that `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[5]):

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

Extra Info

The change was proposed in: P0305R1[6].

[5] https://herbsutter.com/2016/06/30/trip-report-summer-iso-c-standards-meeting-oulu/
[6] 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"};
};
```

Still, with static variables (or `const static`) you usually need to define it in some `cpp` file.

C++11 and `constexpr` keyword allow 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**[7]: 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:

[7] 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[8].

[8] 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 "constant expressions" - declared mainly with `constexpr`. In C++17 the two elements are allowed to exists together - so your lambda can be invoked in a constant expression context.

In C++11/14 the following code wouldn't compile:

```
auto SimpleLambda = [] (int n)  { return n; };
static_assert(SimpleLambda(3) == 3, "");
```

GCC compiled with the `-std=c++11` flag reports the following error:

```
error: call to non-'constexpr' function 'main()::<lambda(int)>'
     static_assert(SimpleLambda(3) == 3, "");
```

However, with the `-std=c++17` the code compiles! This is because since C++17 lambda expressions that follow the rules of standard `constexpr` functions are implicitly declared as `constexpr`.

What does that mean? What are the limitations?

Quoting the Standard - *10.1.5 The constexpr specifier [dcl.constexpr]*

> The definition of a constexpr function shall satisfy the following requirements:
>
> - it shall not be virtual (13.3);
> - its return type shall be a literal type; A >- each of its parameter types shall be a literal type; A >- its function-body shall be = delete, = default, or a compound-statement that does not contain:
>   - an asm-definition,
>   - a goto statement,
>   - an identifier label (9.1),
>   - a try-block, or
>   - a definition of a variable of non-literal type or of static or thread storage duration or for which no initialisation is performed.

Practically, 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.

Lambda can be also explicitly declared `constexpr`:

```
auto SimpleLambda = [] (int n) constexpr { return n; };
```

And if you violate the rules of `constexpr` functions, you'll get a compile-time error.

```
auto FaultyLeakyLambda = [] (int n) constexpr {
    int *p = new int(10);

    return n + (*p);
};
```

`operator new` is not `constexpr` so that won't compile.

Having `constexpr` lambdas will be a great feature combined with the `constexpr` standard algorithms that are coming in C++20.

Extra Info

The change was proposed in: P0170[9].

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

[9] http://wg21.link/p0170