Nicolai M. Josuttis

C++17

The Complete Guide

C++17 - The Complete Guide

Nicolai M. Josuttis

This book is for sale at **http://leanpub.com/cpp17**.

This version was published on **2018/07/15**.

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.

This book was typeset by Nicolai M. Josuttis using the LaTeX document processing system.

Contents

Preface

This book is an experiment in two ways:

- I write a book deeply covering language features without the direct help of a core language expert as a co-author. But I can ask and I do.
- I publish the book myself on Leanpub. That is, this book is written step-by-step and I will publish new versions as soon there is a significant improvement worth to publish it as a new version.

The good thing is:

- You get the view on the language features from an experienced application programmer. Somebody who feels the pain a feature might cause, and asks the relevant questions to be able to motivate and explain the design and its consequences for programming in practice.
- You can benefit from my experience on C++17 while I still learn and write.

But, you are part of the experiment. So help me out: Give feedback about flaws, errors, badly explained features, or gaps so that we all can benefit from these improvements.

Versions of This Book

Because this book is incrementally written, here is the history of its "releases" (newest first):

- **2018-07-15**: Filesystem library chapter fully written.
- **2018-05-30**: Add `scoped_lock)` and `shared_mutex`.
- **2018-05-29**: Add `is_always_lock_free8)` and hardware interference sizes.
- **2018-05-28**: Add variable templates with placeholders.
- **2018-05-27**: Add container support for incomplete types.
- **2018-05-11**: Add node handles for associative and unordered containers.
- **2018-05-11**: First full supported example of parallel algorithms on filesystems.
- **2018-04-04**: Start with a first introduction of (new) parallel algorithms.
- **2018-04-03**: Improvements on `std::optional<>` and more about the filesystem library.
- **2018-03-15**: Publish **new version** with a couple of small fixes.

- **2018-01-12**: Publish **new version** with several fixes and improvements.
- **2018-01-11**: Add new attribute features.
- **2018-01-03**: Add new attributes.
- **2018-01-02**: Add `new` and `delete` with over-aligned data.
- **2017-12-25**: Publish **new version** with several fixes and improvements.
- **2017-12-24**: Add that exception specifications became part of the type.
- **2017-12-23**: Add `u8` prefix for UTF-8 character literals.
- **2017-12-22**: Add hexadecimal floating-point literals.
- **2017-12-15**: Publish **first version**.

Acknowledgments

First I'd like to thank you, the C++ community, for making this book possible. The incredible design of new features, the helpful feedback, the curiousness are the base for an evolving successful language. Especially thanks for all the issues you told and explained me and the feedback you gave.

Especially, I'd like to thank everyone who reviewed drafts of this book or corresponding slides and provided valuable feedback and clarifications. These reviews brought the book to a significantly higher level of quality, and it again proved that good things need the input of many "wise guys." For this reason, so far (this list is still growing) huge thanks to Roland Bock, Matthew Dodkins, Andreas Fertig, Graham Haynes, Austin McCartney, Zachary Turner, Paul Reilly, Barry Revzin, and Vittorio Romeo.

In addition, I'd like to thank everyone in the C++ community and on the C++ standardization committee. In addition to all the work to add new language and library features, they spent many, many hours explaining and discussing their work with me, and they did so with patience and enthusiasm.

A special thanks goes to the LaTeX community for a great text system and to Frank Mittelbach for solving my LaTeX issues (it was almost always my fault).

About This Book

C++17 is the next evolution in modern C++ programming, which is already at least partially supported by the latest version of gcc, clang, and Visual C++. Although it is not as big a step as C++11, it contains a large number of small and valuable language and library features, which again will change the way we program in C++. This applies to both application programmers and programmers providing foundation libraries.

This book will present all the new language and library features in C++17. It will cover the motivation and context of each new feature with examples and background information. As usual for my books, the focus lies on the application of the new features in practice and will demonstrate how features impact day-to-day programming and how to benefit from them in projects.

What You Should Know Before Reading This Book

To get the most from this book, you should already know C++, ideally C++11 and/or C++14. However, you don't have to be an expert. My goal is to make the content understandable for the average C++ programmer, not necessarily familiar with the latest features. You should be familiar with the concepts of classes and inheritance, and you should be able to write C++ programs using components such as IOstreams and containers from the C++ standard library. You should also be familiar with the basic features of "Modern C++", such as `auto`, `decltype`, move semantics, and lambdas.

Nevertheless, I will discuss basic features and review more subtle issues as the need arises, even when such issues aren't directly related to C++17. This ensures that the text is accessible to experts and intermediate programmers alike.

Note that I usually use the modern way of initialization (introduced in C++11 as *uniform initialization*) with curly braces:

```
int i{42};
std::string s{"hello"};
```

This form of initialization, which is called *list initialization*, has the advantage that it can be used with fundamental types, class types, aggregates (extended with C++17), enumeration types (added with C++17), and `auto` (fixed with C++17) and is able to detect narrowing errors (e.g., initializing

an `int` by a floating-point value). If the braces are empty the default constructors of (sub)objects are called and fundamental data types are initialized with `0`/`false`/`nullptr`.[1]

Overall Structure of the Book

The intent of the book is to cover all changes to C++17. This applies to both language and library features as well as both features that affect day-to-day application programming and for the sophisticated implementation of (foundation) libraries. However, the more general cases and examples usually come first.

The different chapters are grouped, but the grouping has no deeper didactic reasoning other than that it makes sense to first introduce language features, because they might be used by the following library features. In principle, you can read the chapters in any order. If features of different chapters are combined, corresponding cross references exist.

As a result the book has the following parts:

- **Part I** covers the new non-template language features.
- **Part II** covers the new language features for generic programming with templates.
- **Part III** introduces the new standard library components.
- **Part IV** covers the extensions and modifications to the existing components of the standard library.

How to Read This Book

In my experience, the best way to learn something new is to look at examples. Therefore, you'll find a lot of examples throughout the book. Some are just a few lines of code illustrating an abstract concept, whereas others are complete programs that provide a concrete application of the material. The latter kind of examples will be introduced by a C++ comment describing the file containing the program code. You can find these files at the Web site of this book at `http://www.cppstd17.com`.

Note that I often talk about programming errors. If there is no special hint, a comment such as

```
... // ERROR
```

means a compile-time error. The corresponding code should not compile (with a conforming compiler).

If I talk about run-time errors, the program might compile but not behave correctly or have undefined behavior (thus, it might or might not do what is expected).

[1] The only exception are atomic data types (type `std::atomic<>`), where even list initialization does not guarantee proper initialization. This will hopefully get fixed with C++20.

The C++17 Standard

The original C++ standard was published in 1998 and subsequently amended by a *technical corrigendum* in 2003, which provided minor corrections and clarifications to the original standard. This "old C++ standard" is known as C++98 or C++03.

The world of "Modern C++" was entered with C++11 and extended with C++14. The international C++ committee now aims at issuing a new standard roughly every 3 years. Clearly, that leaves less time for massive additions, but it brings the changes more quickly to the broader programming community. The development of larger features, then, spans time and might cover multiple standards.

C++17 is just the next step. It is not a revolution, but it brings a huge amount of improvements and extensions.

At the time of this writing, C++17 is already at least partially supported by major compilers. But as usual, compilers differ greatly in their support of new different language features. Some will compile most or even all of the code in this book, while others may only be able to handle a significant subset. However, I expect that this problem will soon be resolved as programmers everywhere demand standard support from their vendors.

Example Code and Additional Information

You can access all example programs and find more information about this book from its Web site, which has the following URL:

```
http://www.cppstd17.com
```

Feedback

I welcome your constructive input—both the negative and the positive. I worked very hard to bring you what I hope you'll find to be an excellent book. However, at some point I had to stop writing, reviewing, and tweaking to "release the new revision." You may therefore find errors, inconsistencies, presentations that could be improved, or topics that are missing altogether. Your feedback gives me a chance to fix this, inform all readers through the book's Web site, and improve any subsequent revisions or editions.

The best way to reach me is by email. You will find the email address at the Web site of this book:

```
http://www.cppstd17.com
```

Please, be sure to have the latest version of this book (remember it is written and published incrementally) and refer to the publishing date of this version when giving feedback. The current publishing date is **2018/07/15** (you can also find it on page ii right after the cover and on top of each page with the PDF format).

Many thanks.

This page is intentionally left blank

Part I

Basic Language Features

This part introduces the new core language features of C++17 not specific for generic programming (i.e., templates). They especially help application programmers in their day-to-day programming. So every C++ programmer using C++17 should know them.

Core language features specific for programming with templates are covered in Part II.

This page is intentionally left blank

Chapter 1

Structured Bindings

Structured bindings allow you to initialize multiple entities by the elements or members of an object.

For example, suppose you have defined a structure of two different members:

```
struct MyStruct {
  int i = 0;
  std::string s;
};

MyStruct ms;
```

You can bind members of this structure directly to new names by using the following declaration:

```
auto [u,v] = ms;
```

Here, the *names* `u` and `v` are what is called *structured bindings*. To some extent they *decompose* the objects passed for initialization (at some point they were called *decomposing declarations*).

Structured bindings are especially useful for functions returning structures or arrays. For example, consider you have a function returning a structure

```
MyStruct getStruct() {
  return MyStruct{42, "hello"};
}
```

You can directly assign the result to two entities giving local names to the returned data members:

```
auto[id,val] = getStruct(); // id and val name i and s of returned struct
```

Here, `id` and `val` are names for the members `i` and `s` of the returned structure. They have the corresponding types, `int` and `std::string`, and can be used as two different objects:

```
if (id > 30) {
  std::cout << val;
}
```

The benefit is direct access and the ability to make the code more readable by binding the value directly to names that convey semantic meaning about their purpose.[1]

The following code demonstrates how code can significantly improve with structured bindings. To iterate over the elements of a `std::map<>` without structured bindings you'd have to program:

```
for (const auto& elem : mymap) {
  std::cout << elem.first << ": " << elem.second << '\n';
}
```

The elements are `std::pairs` of the key and value type and as the members of a `std::pair` are `first` and `second`, you have to use these names to access the key and the value. By using structured bindings the code gets a lot more readable:

```
for (const auto& [key,val] : mymap) {
  std::cout << key << ": " << val << '\n';
}
```

We can directly use the key and value member of each element, using names that clearly demonstrate their semantic meaning.

1.1 Structured Bindings in Detail

In order to understand structured bindings, it is important to be aware that there is a new anonymous variable involved. The new names introduced as structure bindings refer to members/elements of this anonymous variable.

Binding to an Anonymous Entity

The exact behavior of an initialization

```
auto [u,v] = ms;
```

behaves as if we'd initialize a new entity *e* with `ms` and let the structured bindings `u` and `v` become alias names for the members of this new object, similar to defining:

```
auto e = ms;
auto& u = e.i;
auto& v = e.s;
```

The only difference is that we don't have a name for *e*, so that we can't access the initialized entity directly by name.

As a result,

```
std::cout << u << ' ' << v << '\n';
```

prints the values of *e*`.i` and *e*`.s`, which are copies of `ms.i` and `ms.s`.

[1] Thanks to Zachary Turner for pointing that out.

e exists as long as the structured bindings to it exist. Thus, it is destroyed when the structured bindings go out of scope.

As a consequence, unless references are used, modifying the value used for initialization has no effect on the names initialized by a structured binding (and vice versa):

```
MyStruct ms{42,"hello"};
auto [u,v] = ms;
ms.i = 77;
std::cout << u;      // prints 42
u = 99;
std::cout << ms.i;   // prints 77
```

`u` and `ms.i` also have different addresses.

When using structured bindings for return values, the same principle applies. An initialization such as

```
auto [u,v] = getStruct();
```

behaves as if we'd initialize a new entity *e* with the return value of `getStruct()` so that the structured bindings `u` and `v` become alias names for the two members/elements of *e*, similar to defining:

```
auto e = getStruct();
auto& u = e.i;
auto& v = e.s;
```

That is, structured bindings bind to a *new* entity, which is initialized from a return value, instead binding to the return value directly.

To the anonymous entity *e* the usual address and alignment guarantees apply, so that the structured bindings are aligned as the corresponding members they bind to. For example:

```
auto [u,v] = ms;
assert(&((MyStruct*)&u)->s == &v);  // OK
```

Here, `((MyStruct*)&u)` yields a pointer to the anonymous entity as a whole.

Using Qualifiers

We can use qualifiers, such as `const` and references. Again, these qualifiers apply to the anonymous entity *e* as a whole. Usually, the effect is similar to applying the qualifiers to the structured bindings directly, but beware that this is not always the case (see below).

For example, we can declare structured bindings to a `const` reference:

```
const auto& [u,v] = ms;  // a reference, so that u/v refer to ms.i/ms.s
```

Here, the anonymous entity is declared as a `const` reference, which means that `u` and `v` are the names of the members `i` and `s` of the initialized `const` reference to `ms`. As a consequence, any change to the members of `ms` affect the value of `u` and/or `v`.

```
ms.i = 77;                // affects the value of u
std::cout << u;           // prints 77
```

Declared as a non-`const` reference, you can even modify the members of the object/value used for initialization:

```
MyStruct ms{42,"hello"};
auto& [u,v] = ms;          // the initialized entity is a reference to ms
ms.i = 77;                 // affects the value of u
std::cout << u;            // prints 77
u = 99;                    // modifies ms.i
std::cout << ms.i;         // prints 99
```

If the value used to initialize a structured bindings reference is a temporary object, as usual the lifetime of the temporary is extended to the lifetime of the bound structure:

```
MyStruct getStruct();
...
const auto& [a,b] = getStruct();
std::cout << "a: " << a << '\n';      // OK
```

Qualifiers don't Necessarily Apply to the Structured Bindings

As written, the qualifiers apply to the new anonymous entity. They don't necessarily apply to the new names introduced as structured bindings. Where this makes a difference can be demonstrated by specifying an alignment:

```
alignas(16) auto [u,v] = ms;   // align the object, not v
```

Here, we align the initialized anonymous entity and not the structured bindings `u` and `v`. This means that `u` as the first member is forced to be aligned to 16 while `v` is not.

For the same reason, structured bindings do not *decay*[2] although `auto` is used. For example, if we have a structure of raw arrays:

```
struct S {
  const char x[6];
  const char y[3];
};
```

then after

```
S s1{};
auto [a, b] = s1;   // a and b get the exact member types
```

the type of `a` still is `const char[6]`. Again, the `auto` applies to the anonymous entity, which as a whole doesn't decay. This is different from initializing a new object with `auto`, where types decay:

```
auto a2 = a;        // a2 gets decayed type of a
```

[2] The term *decay* describes the type conversions when arguments are passed by value, which means that raw arrays convert to pointers and top-level qualifiers, such as `const` and references, are ignored.

Move Semantics

Move semantics is supported following the rules just introduced. In the following declarations:

```
MyStruct ms = { 42, "Jim" };
auto&& [v,n] = std::move(ms);              // entity is rvalue reference to ms
```

the structured bindings `v` and `n` refer to an anonymous entity being an rvalue reference to `ms`. `ms` still holds its value:

```
std::cout << "ms.s: " << ms.s << '\n';   // prints "Jim"
```

but you can move assign `n`, which refers to `ms.s`:

```
std::string s = std::move(n);              // moves ms.s to s
std::cout << "ms.s: " << ms.s << '\n';   // prints unspecified value
std::cout << "n:    " << n << '\n';      // prints unspecified value
std::cout << "s:    " << s << '\n';      // prints "Jim"
```

As usual, moved-from objects are in a valid state with an unspecified value. Thus, it is fine to print the value but not to make any assumptions about what is printed.[3]

This is slightly different from initializing the new entity with the moved values of `ms`:

```
MyStruct ms = { 42, "Jim" };
auto [v,n] = std::move(ms);                // new entity with moved-from values from ms
```

Here, the initialized anonymous entity is a new object initialized with the moved values from `ms`. So, `ms` already lost its value:

```
std::cout << "ms.s: " << ms.s << '\n';   // prints unspecified value
std::cout << "n:    " << n << '\n';      // prints "Jim"
```

You can still move assign the value of `n` or assign a new value there, but this does not affect `ms.s`:

```
std::string s = std::move(n);              // moves n to s
n = "Lara";
std::cout << "ms.s: " << ms.s << '\n';   // prints unspecified value
std::cout << "n:    " << n << '\n';      // prints "Lara"
std::cout << "s:    " << s << '\n';      // prints "Jim"
```

1.2 Where Structured Bindings can be Used

In principle, structured bindings can be used for structures with public data members, raw C-style arrays, and "tuple-like objects:"

- If in **structures and classes** all non-static data members are public, you can bind each non-static data member to exactly one name.
- For **raw arrays**, you can bind a name to each element.

[3] For strings, moved-from objects are usually empty, but this is *not* guaranteed.

- For any type you can use a **tuple-like API** to bind names to whatever the API defines as "elements." The API roughly consists out of the following elements for a type *type*:
  - `std::tuple_size<`*type*`>::value` has to return the number of elements.
  - `std::tuple_element<`*idx*`,`*type*`>::type` has to return the type of the *idx*th element.
  - A global or member `get<`*idx*`>()` has to yield the value if the *idx*th element.

  The standard library types `std::pair<>`, `std::tuple<>`, and `std::array<>` already provide this API.

If structures or classes provide the tuple-like API, the API is used.

In all cases the number of elements or data members has to fit the number of names in the declaration of the structured binding. You can't skip name and you can't use a name twice. However, you could use a very short name such as `'_'` (as some programmers prefer but others hate and is not allowed in the global namespace), but this works only once in the same scope:

```
auto [_,val1] = getStruct();  // OK
auto [_,val2] = getStruct();  // ERROR: name _ already used
```

Nested or non-flat decomposition is not supported.

The following subsections discuss all these cases in detail.

1.2.1 Structures and Classes

The examples introduced in the sections above demonstrate a couple of simple cases for structured bindings for structures and classes.

Note that there is only limited usage of inheritance possible. All non-static data members must be members of the same class definition (thus, they have to be direct members of the type or of the same unambiguous public base class):

```
struct B {
    int a = 1;
    int b = 2;
};

struct D1 : B {
};
auto [x, y] = D1{};       // OK

struct D2 : B {
    int c = 3;
};
auto [i, j, k] = D2{};    // Compile-Time ERROR
```

1.2.2 Raw Arrays

The following code initializes x and y by the two elements of the raw C-style array:

```
int arr[] = { 47, 11 };
auto [x, y] = arr;          // x and y are ints initialized by elems of arr
auto [z] = arr;             // ERROR: number of elements doesn't fit
```

This of course only is possible as long as the array still has a known size. For an array passed as argument, this is not possible because it *decays* to the corresponding pointer type.

Note that C++ allows us to return arrays with size by reference, so that this feature also applies to functions returning an array provided its size is part of the return type:

```
auto getArr() -> int(&)[2];   // getArr() returns reference to raw int array
...
auto [x, y] = getArr();       // x and y are ints initialized by elems of returned array
```

You can also use structured bindings for a `std::array`, but this uses the tuple-like API approach, which is described next.

1.2.3 `std::pair`, `std::tuple`, and `std::array`

The structured binding mechanism is extensible, so that you can add support for structured bindings to any type. The standard library uses this for `std::pair<>`, `std::tuple<>`, and `std::array<>`.

`std::array`

For example, the following code initializes `i`, `j`, `k`, and `l` by the four elements of the `std::array<>` returned by a function `getArray()`:

```
std::array<int,4> getArray();
...
auto [i,j,k,l] = getArray();    // i,j,k,l name the 4 elements of the copied return value
```

Here, `i`, `j`, `k`, and `l` are structured bindings to the elements of the `std::array` returned by `getArray()`.

Write access is also supported, provided the value for initialization is not a temporary return value. For example:

```
std::array<int,4> stdarr { 1, 2, 3, 4 };
...
auto& [i,j,k,l] = stdarr;
i += 10;                    // modifies std::array[0]
```

`std::tuple`

The following code initializes `a`, `b`, and `c` by the three elements of the `std::tuple<>` returned by `getTuple()`:

```
std::tuple<char,float,std::string> getTuple();
```

```
...
auto [a,b,c] = getTuple();      // a,b,c have types and values of returned tuple
```

That is, `a` gets type `char`, `b` gets type `float`, and `c` gets type `std::string`.

`std::pair`

As another example, the code to handle the return value of calling `insert()` on an associative/unordered container can be made more readable by binding the value directly to names that convey semantic meaning about their purpose, rather than relying on the generic names `first` and `second` from the resulting `std::pair<>` object:

```
std::map<std::string, int> coll;
...
auto [pos,ok] = coll.insert({"new",42});
if (!ok) {
  // if insert failed, handle error using iterator pos:
  ...
}
```

Before C++17, the corresponding check has to be formulated as follows:

```
auto ret = coll.insert({"new",42});
if (!ret.second){
  // if insert failed, handle error using iterator ret.first
  ...
}
```

Note that in this particular case, C++17 provides a way to improve this even further using `if` with initializers.

1.3 Providing a Tuple-Like API for Structured Bindings

As written, you can add support for structured bindings to any type by providing a *tuple-like API* just as the standard library does for `std::pair<>`, `std::tuple<>`, and `std::array<>`.

Enable Read-Only Structured Bindings

The following example demonstrates how to enable structured bindings for a type `Customer`, which might be defined as follows:

lang/customer1.hpp

```
#include <string>
#include <utility>  // for std::move()

class Customer {
```

```
 private:
  std::string first;
  std::string last;
  long val;
 public:
  Customer (std::string f, std::string l, long v)
   : first(std::move(f)), last(std::move(l)), val(v) {
  }
  std::string getFirst() const {
    return first;
  }
  std::string getLast() const {
    return last;
  }
  long getValue() const {
    return val;
  }
};
```

We can provide a tuple-like API as follows:

lang/structbind1.hpp

```
#include "customer1.hpp"
#include <utility>  // for tuple-like API

// provide a tuple-like API for class Customer for structured bindings:
template<>
struct std::tuple_size<Customer> {
  static constexpr int value = 3;    // we have 3 attributes
};

template<>
struct std::tuple_element<2, Customer> {
  using type = long;                 // last attribute is a long
};
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
  using type = std::string;          // the other attributes are strings
};

// define specific getters:
template<std::size_t> auto get(const Customer& c);
template<> auto get<0>(const Customer& c) { return c.getFirst(); }
```

```
template<> auto get<1>(const Customer& c) { return c.getLast(); }
template<> auto get<2>(const Customer& c) { return c.getValue(); }
```

Here, we define a tuple-like API for 3 attributes of a customer, which we essentially map to the three getters of a customer (any other user-defined mapping is possible):

- The first name as `std::string`
- The last name as `std::string`
- The value as `long`

The number of attributes is defined as specialization of `std::tuple_size` for type `Customer`:

```
template<>
struct std::tuple_size<Customer> {
  static constexpr int value = 3;    // we have 3 attributes
};
```

The types of the attributes are defined as specializations of `std::tuple_element`:

```
template<>
struct std::tuple_element<2, Customer> {
  using type = long;                 // last attribute is a long
};
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
  using type = std::string;          // the other attributes are strings
};
```

The type of the third attribute is defined as full specialization for index 2. For the other attributes we use a partial specialization, which has lower priority than the full specialization.

Finally, we define the corresponding getters as overloads of a function `get<>()` in the same namespace as type `Customer`:[4]

```
template<std::size_t> auto get(const Customer& c);
template<> auto get<0>(const Customer& c) { return c.getFirst(); }
template<> auto get<1>(const Customer& c) { return c.getLast(); }
template<> auto get<2>(const Customer& c) { return c.getValue(); }
```

In this case, we have a primary function template declaration and full specializations for all cases. Note that all full specializations of function templates have to use the same signature (including the exact same return type). The reason is that we only provide specific "implementations," no new declarations. The following will not compile:

```
template<std::size_t> auto get(const Customer& c);
template<> std::string get<0>(const Customer& c) { return c.getFirst(); }
template<> std::string get<1>(const Customer& c) { return c.getLast(); }
```

[4] The C++17 standard also allows us to define these `get<>()` functions as member functions, but this is probably an oversight and should not be used.

```
template<> long get<2>(const Customer& c) { return c.getValue(); }
```

By using the new compile-time `if` feature, we can combine the `get<>()` implementations into one function:

```
template<std::size_t I> auto get(const Customer& c) {
  static_assert(I < 3);
  if constexpr (I == 0) {
    return c.getFirst();
  }
  else if constexpr (I == 1) {
    return c.getLast();
  }
  else {  // I == 2
    return c.getValue();
  }
}
```

With this API, we can use structured bindings as usual for objects of type `Customer`:

lang/structbind1.cpp

```
#include "structbind1.hpp"
#include <iostream>

int main()
{
  Customer c("Tim", "Starr", 42);
  auto [f, l, v] = c;
  std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';

  // modify structured bindings:
  std::string s = std::move(f);
  l = "Waters";
  v += 10;
  std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';
  std::cout << "c:        " << c.getFirst() << ' '
            << c.getLast() << ' ' << c.getValue() << '\n';
  std::cout << "s:        " << s << '\n';
}
```

As usual, the structured bindings `f`, `l`, and `v` are references to the "members" of an new anonymous entity initialized with `c`. The initialization calls the corresponding getter once for each member/attribute. Thus, after the initialization of the structured bindings modifying `c` has no effect for them (and vice versa). So, the program has the following output:

```
f/l/v:    Tim Starr 42
```

```
f/l/v:     Waters 52
c:        Tim Starr 42
s:        Tim
```

Using structured binding you could also iterate over the `Customer` elements of a vector:

```
std::vector<Customer> coll;
...
for (const auto& [first, last, val] : coll) {
  std::cout << first << ' ' << last << ": " << val << '\n';
}
```

Enable Structured Bindings with Write Access

The tuple-like API can use functions that yield references. This enables structured bindings with write access. Consider class `Customer` provides an API to read and modify its members:

lang/customer2.hpp

```
#include <string>
#include <utility>  // for std::move()

class Customer {
 private:
  std::string first;
  std::string last;
  long val;
 public:
  Customer (std::string f, std::string l, long v)
   : first(std::move(f)), last(std::move(l)), val(v) {
  }
  const std::string& firstname() const {
    return first;
  }
  std::string& firstname() {
    return first;
  }
  const std::string& lastname() const {
    return last;
  }
  std::string& lastname() {
    return last;
  }
  long value() const {
    return val;
  }
```

```
  long& value() {
    return val;
  }
};
```

For read-write access, we have to overload the getters for constant and non-constant references:

lang/structbind2.hpp

```
#include "customer2.hpp"
#include <utility>  // for tuple-like API

// provide a tuple-like API for class Customer for structured bindings:
template<>
struct std::tuple_size<Customer> {
  static constexpr int value = 3;    // we have 3 attributes
};

template<>
struct std::tuple_element<2, Customer> {
  using type = long;                 // last attribute is a long
};
template<std::size_t Idx>
struct std::tuple_element<Idx, Customer> {
  using type = std::string;          // the other attributes are strings
};

// define specific getters:
template<std::size_t I> decltype(auto) get(Customer& c) {
  static_assert(I < 3);
  if constexpr (I == 0) {
    return c.firstname();
  }
  else if constexpr (I == 1) {
    return c.lastname();
  }
  else {  // I == 2
    return c.value();
  }
}
template<std::size_t I> decltype(auto) get(const Customer& c) {
  static_assert(I < 3);
  if constexpr (I == 0) {
    return c.firstname();
```

```
  }
  else if constexpr (I == 1) {
    return c.lastname();
  }
  else {  // I == 2
    return c.value();
  }
}
template<std::size_t I> decltype(auto) get(Customer&& c) {
  static_assert(I < 3);
  if constexpr (I == 0) {
    return std::move(c.firstname());
  }
  else if constexpr (I == 1) {
    return std::move(c.lastname());
  }
  else {  // I == 2
    return c.value();
  }
}
```

Note that you should have all three overloads, to be able to deal with constant, non-constant, and movable objects.[5] To enable the return value to be a reference, you should use `decltype(auto)`.[6]

Again, we use the new compile-time `if` feature, which makes the implementation simple if the getters have different return types. Without, we would need full specializations again, such as:

```
template<std::size_t> decltype(auto) get(Customer& c);
template<> decltype(auto) get<0>(Customer& c) { return c.firstname(); }
template<> decltype(auto) get<1>(Customer& c) { return c.lastname(); }
template<> decltype(auto) get<2>(Customer& c) { return c.value(); }
...
```

Again, note that the primary function template declaration and the full specializations must have the same signature (including the same return type). The following will not compile:

```
template<std::size_t> decltype(auto) get(Customer& c);
template<> std::string& get<0>(Customer& c) { return c.firstname(); }
template<> std::string& get<1>(Customer& c) { return c.lastname(); }
template<> long& get<2>(Customer& c) { return c.value(); }
```

[5] The standard library provides a fourth `get<>()` overload for `const&&`, which is provided for other reasons (see https://wg21.link/lwg2485) and not necessary to support structured bindings

[6] `decltype(auto)` was introduced with C++14 to be able to deduce a (return) type from the value category of an expression. By using this as a return type, roughly speaking, references are returned by reference, but temporaries are returned by value.

...

Now, you can use structured bindings for read access and to modify the members:

lang/structbind2.cpp

```
#include "structbind2.hpp"
#include <iostream>

int main()
{
  Customer c("Tim", "Starr", 42);
  auto [f, l, v] = c;
  std::cout << "f/l/v:    " << f << ' ' << l << ' ' << v << '\n';

  // modify structured bindings via references:
  auto&& [f2, l2, v2] = c;
  std::string s = std::move(f2);
  f2 = "Ringo";
  v2 += 10;
  std::cout << "f2/l2/v2: " << f2 << ' ' << l2 << ' ' << v2 << '\n';
  std::cout << "c:        " << c.firstname() << ' '
            << c.lastname() << ' ' << c.value() << '\n';
  std::cout << "s:        " << s << '\n';
}
```

The program has the following output:

```
f/l/v:    Tim Starr 42
f2/l2/v2: Ringo Starr 52
c:        Ringo Starr 52
s:        Tim
```

1.4 Afternotes

Structured bindings were first proposed by Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis in https://wg21.link/p0144r0 by using curly braces instead of square brackets. The finally accepted wording for this feature was formulated by Jens Maurer in https://wg21.link/p0217r3.

This page is intentionally left blank

Chapter 2

`if` and `switch` with Initialization

The `if` and `switch` control structures now allow us to specify an initialization clause beside the usual condition or selection clause.

For example, you can write:

```
if (status s = check(); s != status::success) {
  return s;
}
```

where the initialization

```
status s = check();
```

initializes `s`, which is then valid for the whole `if` statement.

2.1 `if` with Initialization

Any value initialized inside an `if` statement is valid until the end of the *then* and the *else* part (if any). For example:

```
if (std::ofstream strm = getLogStrm(); coll.empty()) {
  strm << "<no data>\n";
}
else {
  for (const auto& elem : coll) {
    strm << elem << '\n';
  }
}
// strm no longer declared
```

The destructor for `strm` is called at the end of the then-part or at the end of the else-part.

Another example would be the use of a lock while performing some tasks depending on a condition:

```
if (std::lock_guard<std::mutex> lg{collMutex}; !coll.empty()) {
  std::cout << coll.front() << '\n';
}
```

which due to class template argument deduction now also can be written as:

```
if (std::lock_guard lg{collMutex}; !coll.empty()) {
  std::cout << coll.front() << '\n';
}
```

In any case, this code is equivalent to:

```
{
  std::lock_guard<std::mutex> lg{collMutex};
  if (!coll.empty()) {
    std::cout << coll.front() << '\n';
  }
}
```

with the minor difference that lg is defined in the scope of the if statement so that the condition is in the same scope (*declarative region*), as it is the case for the initialization in a for loop.

Note that any object being initialized must have a name. Otherwise, the initialization creates and immediately destroys a temporary. For example, initializing a lock guard without a name would no longer lock, when the condition is checked:

```
if (std::lock_guard<std::mutex>{collMutex};   // run-time ERROR:
    !coll.empty()) {                          // - no longer locked
  std::cout << coll.front() << '\n';          // - no longer locked
}
```

In principle, a single _ as a name would be enough (as some programmers prefer but others hate and is not allowed in the global namespace):

```
if (std::lock_guard<std::mutex> _{collMutex};   // OK, but...
    !coll.empty()) {
  std::cout << coll.front() << '\n';
}
```

As a third example, consider to insert a new element into a map or unordered map. You can check whether this was successful, as follows:

```
std::map<std::string, int> coll;
...
if (auto [pos,ok] = coll.insert({"new",42}); !ok) {
  // if insert failed, handle error using iterator pos:
  const auto& [key,val] = *pos;
  std::cout << "already there: " << key << '\n';
}
```

Here, we also use structured bindings, to give both the return value and the element at the return position pos useful names instead of just first and second. Before C++17, the corresponding check has to be formulated as follows:

```
auto ret = coll.insert({"new",42});
if (!ret.second){
  // if insert failed, handle error using iterator ret.first
  const auto& elem = *(ret.first);
  std::cout << "already there: " << elem.first << '\n';
}
```

Note that the extension also applies to the new *compile-time* if feature.

2.2 switch with Initialization

Using the switch statement with an initialization allows us to initialize an object/entity for the scope of the switch before formulating the condition to decide where to continue the control flow.

For example, we can initialize a filesystem path before we deal with it according to its path type:

```
using namespace std::filesystem;
...
switch (path p(name); status(p).type()) {
 case file_type::not_found:
   std::cout << p << " not found\n";
   break;
 case file_type::directory:
   std::cout << p << ":\n";
   for (auto& e : std::filesystem::directory_iterator(p)) {
      std::cout << "- " << e.path() << '\n';
   }
   break;
 default:
   std::cout << p << " exists\n";
   break;
}
```

Here, the initialized path p can be used throughout the whole switch statement.

2.3 Afternotes

if and switch with initialization was first proposed by Thomas Köppe in https://wg21.link/p0305r0, initially only extending the if statement. The finally accepted wording was formulated by Thomas Köppe in https://wg21.link/p0305r1.

This page is intentionally left blank

Chapter 3

Inline Variables

One strength of C++ is its ability to support the development of header-only libraries. However, up to C++17, this was only possible if no global variables/objects were needed or provided by such a library.

Since C++17 you can define a variable/object in a header file as `inline` and if this definition is used by multiple translation units, they all refer to the same unique object:

```
class MyClass {
  static inline std::string name = "";   // OK since C++17
  ...
};

inline MyClass myGlobalObj;   // OK even if included/defined by multiple CPP files
```

3.1 Motivation of Inline Variables

In C++, it is not allowed to initialize a non-`const` static member inside the class structure:

```
class MyClass {
  static std::string name = "";   // Compile-Time ERROR
  ...
};
```

Defining the variable outside the class structure is also an error, if this definitions is part of a header file, included by multiple CPP files:

```
class MyClass {
  static std::string name;        // OK
  ...
};
MyClass::name = "";    // Link ERROR if included by multiple CPP files
```

According to the *one definition rule* (ODR), a variable or entity had to be defined in exactly one translation unit.

Even preprocessor guards do not help:

```
#ifndef MYHEADER_HPP
#define MYHEADER_HPP

class MyClass {
  static std::string name;        // OK
  ...
};
MyClass.name = "";                // Link ERROR if included by multiple CPP files

#endif
```

The problem is not that the header file might be included multiple times, the problem is that two different CPP files include the header so that both define `MyClass.name`.

For the same reason, you get a link error if you define an object of your class in a header file:

```
class MyClass {
  ...
};
MyClass myGlobalObject;   // Link ERROR if included by multiple CPP files
```

Workarounds

For some cases, there are workarounds:

- You can initialize `static const` integral data members in a `class`/`struct`:

```
class MyClass {
  static const bool trace = false;
  ...
};
```

- You can define an inline function returning a `static` local variable:

```
inline std::string getName() {
  static std::string name = "initial value";
  return name;
}
```

- You can define a `static` member function returning the value:

```
std::string getMyGlobalObject() {
  static std::string myGlobalObject = "initial value";
  return myGlobalObject;
}
```

- You can use variable templates (since C++14):

```
template<typename T = std::string>
T myGlobalObject = "initial value";
```

- You can derive from a base class template for the static member(s):

```
template<typename Dummy>
class MyClassStatics
{
  static std::string name;
};

template<typename Dummy>
std::string MyClassStatics<Dummy>::name = "initial value";

class MyClass : public MyClassStatics<void>
{
  ...
};
```

But all these approaches lead to significant overhead, less readability and/or different ways to use the global variable. In addition, the initialization of a global variable might be postponed until it's first usage, which disables applications where we want to initialize objects at program start (such as when using an object to monitor the process).

3.2 Using Inline Variables

Now, with `inline`, you can have a single globally available object by defining it only in a header file, which might get included by multiple CPP files:

```
class MyClass {
  static inline std::string name = "";   // OK since C++17
  ...
};

inline MyClass myGlobalObj;   // OK even if included/defined by multiple CPP files
```

The initializations are performed when the first translation unit that includes the header or contains these definitions gets entered.

Formally the `inline` used here has the same semantics as a function declared inline:

- It can be defined in multiple translation units, provided all definitions are identical.
- It must be defined in every translation unit in which it is used.

Both is given by including the definition from the same header file. The resulting behavior of the program is as if there is exactly one variable.

You can even apply this to define atomic types in header files only:

```
inline std::atomic<bool> ready{false};
```

Note that as usual for `std::atomic` you always have to initialize the values when you define them.

Note that still you have to ensure that types are *complete* before you can initialize them. For example, if a `struct` or `class` has a static member of its own type, the member can only be inline defined after the type declaration:

```
struct MyValue {
  int value;
  MyValue(int i) : value{i} {
  }
  // one static object to hold the maximum value of this type:
  static const MyValue max;   // can only be declared here
  ...
};
inline const MyValue MyValue::max = 1000;
```

See the header file to track all `new` calls for another example of using inline variables.

3.3 `constexpr` now implies `inline`

For static data members, `constexpr` implies `inline` now, such that the following declaration since C++17 *defines* the static data member `n`:

```
struct D {
  static constexpr int n = 5; // C++11/C++14: declaration
                              // since C++17: definition
};
```

That is, it is the same as:

```
struct D {
  inline static constexpr int n = 5;
};
```

Note that before C++17 you already could declare without a corresponding definition:

```
struct D {
  static constexpr int n = 5;
};
```

But this only worked if no definition of `D::n` was needed, which, for example, was the case if `D::n` was passed by value:

```
std::cout << D::n;        // OK (ostream::operator<<(int) gets D::n by value)
```

If `D::n` was passed by reference to a non-inlined function and/or the call was not optimized away this was invalid. For example:

```
int inc(const int& i);

std::cout << inc(D::n);   // usually an ERROR
```

This code violates the *one definition rule* (ODR). When built with an optimizing compiler, it may work as expected or may give a link error due to the missing definition. When built without optimizations, it will almost certainly be rejected due to the missing definition of `D::n`.[1]

Thus, before C++17, you had to define `D::n` in exactly one translation unit:

```
constexpr int D::n;       // C++11/C++14: definition
                          // since C++17: redundant declaration (deprecated)
```

When built in C++17, the declaration inside the class is a definition by itself, so the code is now valid without the former definition, which is still valid but deprecated.

3.4 Inline Variables and `thread_local`

By using `thread_local` you can also make an inline variable unique for each thread:

```
struct ThreadData {
  inline static thread_local std::string name;        // unique name per thread
  ...
};

inline thread_local std::vector<std::string> cache;   // one cache per thread
```

As a complete example, consider the following header file:

lang/inlinethreadlocal.hpp

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

struct MyData {
  inline static std::string gName = "global";             // unique in program
  inline static thread_local std::string tName = "tls";   // unique per thread
  std::string lName = "local";                            // for each object
  ...
  void print(const std::string& msg) const {
    std::cout << msg << '\n';
    std::cout << "- gName: " << gName << '\n';
    std::cout << "- tName: " << tName << '\n';
    std::cout << "- lName: " << lName << '\n';
  }
};

inline thread_local MyData myThreadData;   // one object per thread
```

[1] Thanks to Richard Smith for pointing that out.

You can use it in the translation unit having `main()`:

lang/inlinethreadlocal1.cpp

```
#include "inlinethreadlocal.hpp"
#include <thread>

void foo();

int main()
{
  myThreadData.print("main() begin:");

  myThreadData.gName = "thread1 name";
  myThreadData.tName = "thread1 name";
  myThreadData.lName = "thread1 name";
  myThreadData.print("main() later:");

  std::thread t(foo);
  t.join();
  myThreadData.print("main() end:");
}
```

And you can use the header file in another translation unit defining `foo()`, which is called in a different thread:

lang/inlinethreadlocal2.cpp

```
#include "inlinethreadlocal.hpp"

void foo()
{
  myThreadData.print("foo() begin:");

  myThreadData.gName = "thread2 name";
  myThreadData.tName = "thread2 name";
  myThreadData.lName = "thread2 name";
  myThreadData.print("foo() end:");
}
```

The program has the following output:

```
main() begin:
- gName: global
- tName: tls
- lName: local
```

```
main() later:
- gName: thread1 name
- tName: thread1 name
- lName: thread1 name
foo() begin:
- gName: thread1 name
- tName: tls
- lName: local
foo() end:
- gName: thread2 name
- tName: thread2 name
- lName: thread2 name
main() end:
- gName: thread2 name
- tName: thread1 name
- lName: thread1 name
```

3.5 Afternotes

Inline variables were motivated by David Krauss in https://wg21.link/n4147 and first proposed by Hal Finkel and Richard Smith in https://wg21.link/n4424. The finally accepted wording was formulated by Hal Finkel and Richard Smith in https://wg21.link/p0386r2.

This page is intentionally left blank

Chapter 4

Aggregate Extensions

One way to initialize objects in C++ is *aggregate initialization*, which allows the initialization of an aggregate[1] from multiple values with curly braces:

```
struct Data {
  std::string name;
  double value;
};

Data x{"test1", 6.778};
```

Since C++17, aggregates can have base classes, so that for such structures being derived from other classes/structures list initialization is allowed:

```
struct MoreData : Data {
  bool done;
};

MoreData y{{"test1", 6.778}, false};
```

As you can see, aggregate initialization now supports nested braces to pass values to the derived members of the base class.

And as for initialization of subobjects with members, you can skip the nested braces if a base type or subobject only gets one value:

```
MoreData y{"test1", 6.778, false};
```

[1] Aggregates are either arrays or simple, C-like classes that have no user-provided constructors, no private or protected non-static data members, no virtual functions, and before C++17 no base classes.

4.1 Motivation for Extended Aggregate Initialization

Without this feature, deriving a structure from another disabled aggregate initialization, so that you had to define a constructor:

```
struct Cpp14Data : Data {
  bool done;
  Cpp14Data (const std::string& s, double d, bool b)
   Data{s,d}, done{b} {
  }
};

Cpp14Data y{"test1", 6.778, false};
```

Now we have this ability for free with the syntax using nested braces, which can be omitted if only one value is passed:

```
MoreData x{{"test1", 6.778}, false};   // OK since C++17
MoreData y{"test1", 6.778, false};     // OK
```

Note that because this is an aggregate now, other initializations are possible:

```
MoreData u;        // OOPS: value/done are uninitialized
MoreData z{};      // OK: value/done have values 0/false
```

If this is too dangerous, you still better provide a constructor.

4.2 Using Extended Aggregate Initialization

One typical application is the ability to list initialize members of a C style structure derived by a class to add additional data members or operations. For example:

```
struct Data {
  const char* name;
  double value;
};

struct PData : Data {
  bool critical;
  void print() const {
    std::cout << '[' << name << ',' << value << "]\n";
  }
};

PData y{{"test1", 6.778}, false};
y.print();
```

Here, the arguments in the inner parentheses are passed to the base type `Data`.

Note that you can skip initial values. In that case the elements are *zero initialized* (calling the default constructor or initializing fundamental data types with `0`, `false`, or `nullptr`). For example:

```
PData a{};          // zero-initialize all elements
PData b{{"msg"}};   // same as {{"msg",0.0},false}
PData c{{}, true};  // same as {{nullptr,0.0},true}
PData d;            // values of fundamental types are unspecified
```

Note the difference between using empty curly braces and no braces at all:

- The definition of `a` zero-initializes all members so that the string `name` is default constructed, the `double value` is initialized by `0.0`, and the `bool flag` is initialized by `false`.
- The definition of `d` only initializes the string `name` by calling the default constructor; all other members are not initialized and have a unspecified value.

You can also derive aggregates from non-aggregate classes. For example:

```
struct MyString : std::string {
  void print() const {
    if (empty()) {
      std::cout << "<undefined>\n";
    }
    else {
      std::cout << c_str() << '\n';
    }
  }
};

MyString x{{"hello"}};
MyString y{"world"};
```

You can even derive aggregates from multiple base classes and/or aggregates:

```
template<typename T>
struct D : std::string, std::complex<T>
{
   std::string data;
};
```

which you could then use and initialize as follows:

```
D<float> s{{"hello"}, {4.5,6.7}, "world"};          // OK since C++17
D<float> t{"hello", {4.5, 6.7}, "world"};           // OK since C++17
std::cout << s.data;                                // outputs: "world"
std::cout << static_cast<std::string>(s);           // outputs: "hello"
std::cout << static_cast<std::complex<float>>(s);   // outputs: (4.5,6.7)
```

The inner initializer lists are passed to the base classes in the order of the base class declarations.

The new feature also helps defining an overload of lambdas with very little code.

4.3 Definition of Aggregates

To summarize, since C++17 an *aggregate* is defined as

- either an array
- or a *class type* (`class`, `struct`, or `union`) with:
  - no user-declared or `explicit` constructor
  - no constructor inherited by a `using` declaration
  - no `private` or `protected` non-static data members
  - no `virtual` functions
  - no `virtual`, `private`, or `protected` base classes

To be able to *use* an aggregate it is also required that no `private` or `protected` base class members or constructors are used during initialization.

C++17 also introduces a new type trait `is_aggregate<>` to test, whether a type is an aggregate:

```
template<typename T>
struct D : std::string, std::complex<T> {
   std::string data;
};
D<float> s{{"hello"}, {4.5,6.7}, "world"};          // OK since C++17
std::cout << std::is_aggregate<decltype(s)>::value; // outputs: 1 (true)
```

4.4 Backward Incompatibilities

Note that the following example no longer compiles:

lang/aggr14.cpp

```
struct Derived;

struct Base {
 friend struct Derived;
 private:
  Base() {
  }
};

struct Derived : Base {
};

int main()
{
  Derived d1{};     // ERROR since C++17
```

```
  Derived d2;       // still OK (but might not initialize)
}
```

Before C++17, `Derived` was not an aggregate. Thus,

```
Derived d1{};
```

was calling the implicitly defined default constructor of `Derived`, which by default called the default constructor of the base class `Base`. Although the default constructor of the base class is `private`, it was valid to be called via the default constructor of the derived class, because the derived class was defined to be a `friend` class.

Since C++17, `Derived` in this example is an aggregate, not having an implicit default constructor at all (the constructor is not inherited by a `using` declaration). So the initialization is an aggregate initialization, for which it is not allowed to call `private` constructors of bases classes. Whether the base class is a `friend` doesn't matter.

4.5 Afternotes

Extended aggregate initialization was first proposed by Oleg Smolsky in https://wg21.link/n4404. The finally accepted wording was also formulated by Oleg Smolsky in https://wg21.link/p0017r1.

The type trait `std::is_aggregate<>` was introduced as a US national body comment for the standardization of C++17 (see https://wg21.link/lwg2911).

This page is intentionally left blank

Chapter 5

Mandatory Copy Elision or Passing Unmaterialized Objects

The topic of this chapter can be seen from two points of view:

- Technically, C++17 introduces a new rule for *mandatory copy elision* under certain conditions: The former option to eliminate copying temporary objects, when passing or returning them by value, now becomes mandatory.
- As a result we deal with passing around the values of *unmaterialized objects* for initialization.

I will introduce this feature technically, coming later to the effect and terminology of *materialization*.

5.1 Motivation for Mandatory Copy Elision for Temporaries

Since the first standard, C++ permits certain copy operations to be omitted (*elided*) even if this might impact the behavior of a program as a side effect that no copy constructor gets called. One case is when a temporary object is used to initialize a new object. This especially happens when a temporary is passed to or returned from a function by value. For example:

```
class MyClass
{
  ...
};

void foo(MyClass param) {   // param is initialized by passed argument
  ...
}

MyClass bar() {
  return MyClass();         // returns temporary
}
```

```
int main()
{
  foo(MyClass());       // pass temporary to initialize param
  MyClass x = bar();    // use returned temporary to initialize x
  foo(bar());           // use returned temporary to initialize param
}
```

However, because these optimizations were not mandatory, copying the objects had to be *possible* by providing an implicit or explicit copy or move constructor. That is, although the copy/move constructor was usually not called, it had to exist. Code like this didn't compile when no copy/move constructor was defined.

Thus, with the following definition of class `MyClass` the code above did not compile:

```
class MyClass
{
 public:
  ...
  // no copy/move constructor defined:
  MyClass(const MyClass&) = delete;
  MyClass(MyClass&&) = delete;
  ...
};
```

It was enough not to have the copy constructor, because the move constructor is only implicitly available, when no copy constructor (or assignment operator or destructor) is user-declared.

The copy elision to initialize objects from temporaries is mandatory since C++17. In fact, what we will see later is that we simply pass a value for initialization as argument or return value that is used then to *materialize* a new object.

This means that even with a definition of class `MyClass` not enabling copying at all, the example above compiles.

However, note that all other optional copy elisions still are optional and require a callable copy or move constructor. For example:

```
MyClass foo()
{
  MyClass obj;
  ...
  return obj;   // still requires copy/move support
}
```

Here, inside `foo()` `obj` is a variable with a name (which is an *lvalue*). So the ***named*** *return value optimization* (NRVO) is used, which still requires copy/move support. This would even be the case if `obj` is a parameter:

```
MyClass bar(MyClass obj)   // copy elision for passed temporaries
{
```

```
  ...
  return obj;   // still requires copy/move support
}
```

While passing a temporary (which is a *prvalue*) to the function is no longer a copy/move, returning the parameter requires copy/move support, because the returned object has a name.

As part of this change a couple of modifications and clarifications in the terminology of value categories were made.

5.2 Benefit of Mandatory Copy Elision for Temporaries

One benefit of this feature is, of course, guaranteed better performance when returning a value that is expensive to copy. Although move semantics helps to reduce copying costs significantly, it still can become a key improvement not to perform copies even if they are pretty cheap (e.g., if the objects have many fundamental data types as members). This might reduce the need to use out-parameters rather than simply returning a value (provided the return value is created with the return statement).

Another benefit is the ability now to define a factory function that *always* works, because it can now also return an object even when neither copying nor moving is allowed. For example, consider the following generic factory function:

lang/factory.hpp

```
#include <utility>

template <typename T, typename... Args>
T create(Args&&... args)
{
  ...
  return T{std::forward<Args>(args)...};
}
```

This function can now even be used for a type such as `std::atomic<>`, where neither the copy nor the move constructor is defined:

lang/factory.cpp

```
#include "factory.hpp"
#include <memory>
#include <atomic>

int main()
{
  int i = create<int>(42);
  std::unique_ptr<int> up = create<std::unique_ptr<int>>(new int{42});
  std::atomic<int> ai = create<std::atomic<int>>(42);
```

```
}
```

As another effect, for classes with explicitly deleted move constructors, you can now return temporaries by value and initialize objects with them:

```
class CopyOnly {
 public:
   CopyOnly() {
   }
   CopyOnly(int) {
   }
   CopyOnly(const CopyOnly&) = default;
   CopyOnly(CopyOnly&&) = delete;   // explicitly deleted
};

CopyOnly ret() {
  return CopyOnly{};   // OK since C++17
}

CopyOnly x = 42;       // OK since C++17
```

The initialization of `x` was invalid before C++17, because the *copy initialization* (initialization using the `=`) needed the conversion of `42` to a temporary and that temporary in principle needed the move constructor, although it was never called. (The fact that the copy constructor serves as fallback for a move constructor only applies, if the move constructor is *not* user-declared.)

5.3 Clarified Value Categories

As a side effect of the proposed change to require copy elision for temporaries when initializing new objects, some adjustments to *value categories* were made.

5.3.1 Value Categories

Each expression in a C++ program has a value category. The category especially describes what can be done with an expression.

History of Value Categories

Historically (taken from C), we only had *lvalue* and *rvalue*, based on an assignment:

```
x = 42;
```

where `x` used as expression was an *lvalue*, because it could stand on the left side of an assignment, and `42` used as an expression was an *rvalue*, because it could only stand on the right side. But already with ANSI-C things became more complicated, because an `x` declared as `const int` could not stand on the left side of an assignment but still was an (nonmodifyable) lvalue.

And in C++11 we got movable objects, which were semantically objects for the right side of an assignment only, but could be modified, because an assignment operator could steal their value. For this reason, the category *xvalue* was introduced and the former category *rvalue* got a new name *prvalue*.

Value Categories Since C++11

Since C++11, the value categories are as described in Figure 5.1: We have the core categories *lvalue*, *prvalue* ("pure rvalue"), and *xvalue* ("eXpiring value"). The composite categories are: *glvalue* ("generalized lvalue," which is the union of *lvalue* and *xvalue*) and *rvalue* (the union of *xvalue* and *prvalue*).

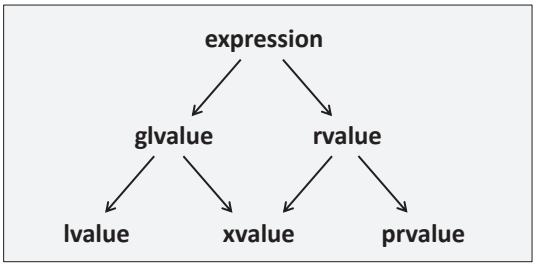

Figure 5.1. Value Categories since C++11

Examples of ***lvalues*** are:

- An expression that is just the name of a variable, function, or member
- An expression that is just a string literal
- The result of the built-in unary * operator (i.e., what dereferencing a raw pointer yields)
- The result of a function returned by lvalue reference (*type*&)

Examples of ***prvalues*** are:

- Expressions that consist of a literal that is not a string literal (or a user-defined literal, where the return type of the associated literal operator defines the category)
- The result of the built-in unary & operator (i.e., what taking the address of an expression yields)
- The result of built-in arithmetic operators
- The result of a function returned by value
- A lambda expression

Examples of ***xvalues*** are:

- The result of a function returned by rvalue reference (*type*&&, especially returned by `std::move()`)
- A cast to an rvalue reference to an object type

Roughly speaking:

- All names used as expressions are *lvalues*.
- All string literals used as expression are *lvalues*.
- All other literals (`4.2`, `true`, or `nullptr`) are *prvalues*.

- All temporaries (especially objects returned by value) are *prvalues*.
- The result of `std::move()` is an *xvalue*.

For example:

```
class X {
};

X v;
const X c;

void f(const X&);   // accepts an expression of any value category
void f(X&&);        // accepts prvalues and xvalues only, but is a better match

f(v);               // passes a modifiable lvalue to the first f()
f(c);               // passes a non-modifiable lvalue to the first f()
f(X());             // passes a prvalue to the second f()
f(std::move(v));    // passes an xvalue to the second f()
```

It's worth emphasizing that strictly speaking glvalues, prvalues, and xvalues are terms for expressions and *not* for values (which means that these terms are misnomers). For example, a variable itself is not an lvalue; only an expression denoting the variable is an lvalue:

```
int x = 3;   // x here is a variable, not an lvalue
int y = x;   // x here is an lvalue
```

In the first statement `3` is a prvalue initializing the variable (not the lvalue) `x`. In the second statement `x` is an lvalue (its evaluation designates an object containing the value `3`). The lvalue `x` is converted to a prvalue, which is what initializes the variable `y`.

5.3.2 Value Categories Since C++17

C++17 didn't change these value categories but clarified their semantic meaning (as described in Figure 5.2).

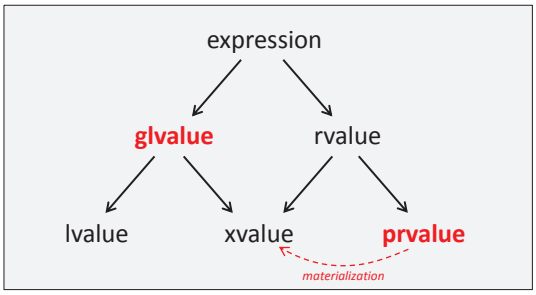

Figure 5.2. Value Categories since C++17

The key approach to explain value categories now is that in general we have two kinds of expressions

- **glvalues**: expressions for *locations* of objects or functions
- **prvalues:** expressions for *initializations*

An *xvalue* is then considered a special location, representing an object whose resources can be reused (usually because it is near the end of its lifetime).

C++17 then introduces a new term, called *materialization* (of a temporary) for the moment a prvalue becomes a temporary object. Thus, a *temporary materialization conversion* is a prvalue-to-xvalue conversion.

Any time a prvalue validly appears where a glvalue (lvalue or xvalue) is expected, a temporary object is created and initialized with the prvalue (recall that prvalues are primarily "initializing values"), and the prvalue is replaced by an *xvalue* designating the temporary. So in the example above, we strictly speaking have:

```
void f(const X& p);   // accepts an expression of any value category,
                      // but expects a glvalue

f(X());               // passes a prvalue materialized as xvalue
```

Because `f()` in this example has a reference parameter, it expects a glvalue argument. However, the expression `X()` is a prvalue. The "temporary materialization" rule therefore kicks in, and the expression `X()` is "converted" to an xvalue designating a temporary object initialized with the default constructor.

Note that materialization does not mean that we create a new/different object. The lvalue reference `p` still *binds* to both an xvalue and a prvalue, although the latter now always involves a conversion to an xvalue.

With this modification (that prvalues are no longer objects but are instead expressions that can be used to initialize objects), the required copy elision makes perfect sense, because the prvalues no longer need to be movable in order to initialize a variable using assignment syntax. We only pass an initial value around that is sooner or later *materialized* to initialize an object.[1]

5.4 Unmaterialized Return Value Passing

Unmaterialized return value passing applies to all forms of returning a temporary object (prvalue) by value:

- When we return a literal that is not a string literal:

```
int f1() {     // return int by value
  return 42;
}
```

[1] Thanks to Richard Smith and Graham Haynes for pointing that out.

- When we return a temporary object by its type or `auto`:

```
auto f2() {   // return deduced type by value
  ...
  return MyType{...};
}
```

- When we return a temporary object by `decltype(auto)`:

```
decltype(auto) f3() {   // return temporary from return statement by value
  ...
  return MyType{...};
}
```

  Remember that a declaration with `decltype(auto)` operates *by value* if the expression used for initialization (here the return statement) is an expression that creates a temporary (a prvalue).

Because we return a prvalue in all these cases by value, we don't require any copy/move support at all.

5.5 Afternotes

The mandatory copy elision for initializations from temporaries was first proposed by Richard Smith in https://wg21.link/p0135r0. The finally accepted wording was also formulated by Richard Smith in https://wg21.link/p0135r1.

Chapter 6

Lambda Extensions

Lambdas, introduced with C++11, and generic lambdas, introduced with C++14, are a success story. They allow us to specify functionality as arguments, which makes it a lot easier to specify behavior right where it is needed.

C++17 improved their abilities to allow the use of lambdas in even more places:

- in constant expressions (i.e., at compile time)
- in places where you need a copy of the current object (e.g., when calling lambdas in threads)

6.1 `constexpr` Lambdas

Since C++17, lambdas are implicitly `constexpr` if possible. That is, any lambda can be used in compile-time contexts provided the features it uses are valid for compile-time contexts (e.g., only literal types, no static variables, no `virtual`, no `try/catch`, no `new/delete`).

For example, you can use the result of calling a lambda computing the square of a passed value as compile-time argument to the declaration of the size of a `std::array<>`:

```
auto squared = [](auto val) {       // implicitly constexpr since C++17
  return val*val;
};
std::array<int,squared(5)> a;       // OK since C++17 => std::array<int,25>
```

Using features that are not allowed in `constexpr` contexts disable this ability, but you can still use the lambda in run-time contexts:

```
auto squared2 = [](auto val) {      // implicitly constexpr since C++17
  static int calls = 0;    // OK, but disables lambda for constexpr contexts
  ...
  return val*val;
};
std::array<int,squared2(5)> a;      // ERROR: static variable in compile-time context
std::cout << squared2(5) << '\n';   // OK
```

To find out at compile time whether a lambda is valid for a compile-time context, you can declare it as `constexpr`:

```
auto squared3 = [](auto val) constexpr {    // OK since C++17
  return val*val;
};
```

With specified return types the syntax looks as follows:

```
auto squared3i = [](int val) constexpr -> int {    // OK since C++17
  return val*val;
};
```

The usual rules regarding `constexpr` for functions apply: If the lambda is used in a run-time context, the corresponding functionality is performed at run time.

However, using features in a `constexpr` lambda that are not valid in a compile-time context results in a compile-time error:[1]

```
auto squared4 = [](auto val) constexpr {
  static int calls=0;              // ERROR: static variable in compile-time context
  ...
  return val*val;
};
```

For an implicit or explicit `constexpr` lambda, the function call operator is `constexpr`. That is, the definition of

```
auto squared = [](auto val) {              // implicitly constexpr since C++17
  return val*val;
};
```

converts into the *closure type*:

```
class CompilerSpecificName {
  public:
    ...
    template<typename T>
    constexpr auto operator() (T val) const {
      return val*val;
    }
};
```

Note that the function call operator of the generated closure type is automatically `constexpr` here. In general since C++17, the generated function call operator is `constexpr` if either the lambda is explicitly defined to be `constexpr` or it is implicitly `constexpr` (as it is the case here).

[1] Features not allowed in compile-time context are, for example, `static` variables, `virtual` functions, `try` and `catch`, and `new` and `delete`.

6.2 Passing Copies of `this` to Lambdas

When using lambdas in member functions, you have no implicit access to the object the member function is called for. That is, inside the lambda, without capturing `this` in any form, you can't use members of the object (independent from whether you qualify them with `this->`):

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
      auto l1 = [] { std::cout << name << '\n'; };        // ERROR
      auto l2 = [] { std::cout << this->name << '\n'; };  // ERROR
      ...
    }
};
```

In C++11 and C++14, you have to pass `this` either by value or by reference:

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
      auto l1 = [this] { std::cout << name << '\n'; };  // OK
      auto l2 = [=] { std::cout << name << '\n'; };     // OK
      auto l3 = [&] { std::cout << name << '\n'; };     // OK
      ...
    }
};
```

However, the problem here is that even copying `this` captures the underlying object by reference (as only the *pointer* was copied). This can become a problem if the lifetime of the lambda exceeds the lifetime of the object upon which the member function is invoked. One critical example is when the lambda defines the task of a new thread, which should use its own copy of the object to avoid any concurrency or lifetime issues. Another reason might simply be to pass a copy of the object with its current state.

There was a workaround possible since C++14, but it doesn't read and work well:

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
```

```
        auto l1 = [thisCopy=*this] { std::cout << thisCopy.name << '\n'; };
        ...
    }
};
```

For example, programmers could still accidentally use `this`, when also using `=` or `&` to capture other objects:

```
auto l1 = [&, thisCopy=*this] {
             thisCopy.name = "new name";
             std::cout << name << '\n'; // OOPS: still the old name
          };
```

Since C++17, you can explicitly ask to capture a copy of the current object by capturing `*this`:

```
class C {
  private:
    std::string name;
  public:
    ...
    void foo() {
      auto l1 = [*this] { std::cout << name << '\n'; };
      ...
    }
};
```

That is, the capture `*this` means that a *copy* of the current object is passed to the lambda.

Still you can combine capturing `*this` with other captures, as long as there is no contradiction for handling `this`:

```
auto l2 = [&, *this] { ... };      // OK
auto l3 = [this, *this] { ... };   // ERROR
```

Here is a complete example:

lang/lambdathis.cpp

```
#include <iostream>
#include <string>
#include <thread>

class Data {
  private:
    std::string name;
  public:
    Data(const std::string& s) : name(s) {
    }
    auto startThreadWithCopyOfThis() const {
        // start and return new thread using this after 3 seconds:
```

```
        using namespace std::literals;
        std::thread t([*this] {
                                std::this_thread::sleep_for(3s);
                                std::cout << name << '\n';
                            });
        return t;
    }
};

int main()
{
    std::thread t;
    {
      Data d{"c1"};
      t = d.startThreadWithCopyOfThis();
    } // d is no longer valid
    t.join();
}
```

The lambda takes a copy of `*this`, which means that a copy of `d` is passed. Therefore, it is no problem that probably the thread uses the passed object after the destructor of `d` was called.

If we'd have captured `this` with `[this]`, `[=]`, or `[&]`, the thread runs into undefined behavior, because when printing the name in the lambda passed to the thread the lambda would use a member of a destroyed object.

6.3 Afternotes

`constexpr` lambdas were first proposed by Faisal Vali, Ville Voutilainen, and Gabriel Dos Reis in https://wg21.link/n4487. The finally accepted wording was formulated by Faisal Vali, Jens Maurer, and Richard Smith in https://wg21.link/p0170r1.

Capturing `*this` in lambdas was first proposed by H. Carter Edwards, Christian Trott, Hal Finkel Jim Reus, Robin Maffeo, and Ben Sander in https://wg21.link/p0018r0. The finally accepted wording was formulated by H. Carter Edwards, Daveed Vandevoorde, Christian Trott, Hal Finkel, Jim Reus, Robin Maffeo, and Ben Sander in https://wg21.link/p0180r3.

This page is intentionally left blank

Chapter 7

New Attributes and Attribute Features

Since C++11 you can specify *attributes* (formal annotations that enable or disable warnings). With C++17, new attributes were introduced. In addition, attributes can now be used at a few some more places and with some additional convenience.

7.1 Attribute `[[nodiscard]]`

The new attribute `[[nodiscard]]` can be used to encourage warnings by the compiler if a return value of a function is not used (issuing a warning is not required, though).

Usually, this can be used to signal misbehavior when return values are not used. The misbehavior might be:

- **memory leaks**, such as not using returned allocated memory,
- **unexpected or non-intuitive behavior**, such as getting different/unexpected behavior when not using the return value;
- **unnecessary overhead** such as calling something that is a no-op if the return value is not used.

Here are some examples, where using the attribute is useful:

- Functions allocating resources that have to get freed by another function call should be marked with `[[nodiscard]]`. A typical example would be function to allocate memory, such as `malloc()` or the member function `allocate()` of allocators.

  Note however, that some function *might* return a value so that no compensating call is necessary. For example, programmers call the C function `realloc()` with a size of zero bytes to free memory, so that the return values has not to be saved to call `free()` later.
- A good example of a function changing its behavior non-intuitively when not using the return value is `std::async()` (introduced with C++11). It has the purpose to start a functionality asynchronously and returns a handle to wait for its end (and use any outcome). When the return value is not used the call becomes a synchronous call, because the destructor of the unused return

value gets called immediately. which waits for the end of the started functionality. So not using the return value silently contradicts the whole purpose why `std::async()` was called. With `[[nodiscard]]` the compilers warns about this.

- Another example is the member function `empty()`, which checks whether an object (container/string) has no elements. Programmers surprisingly often call this to "empty" the container (remove all elements):

  ```
  cont.empty();
  ```

  This wrong application of `empty()` can often be detected, because it doesn't use the return value. So, marking the member function accordingly:

  ```
  class MyContainer {
    ...
    public:
      [[nodiscard]] bool empty() const noexcept;
      ...
  };
  ```

  helps to detect such an error.

Although the language feature was introduced with C++17, it is not used yet in the standard library. The proposal to apply this feature there simply came too late for C++17. So one of the key motivations for this feature, adding it to `std::async()` was not done yet. However, for all the examples discussed above, corresponding fixes will come with the next C++ standard (see https://wg21.link/p0600r1 for the already accepted proposal).

When defining `operator new()`, you should mark the functions with `[[nodiscard]]` as it is done, for example, when defining a header file to track all calls of `new`.

Compilers already had non-portable ways to mark functions accordingly, such as `[[gnu:warn_unused_result]]` for gcc or clang.

7.2 Attribute `[[maybe_unused]]`

The new attribute `[[maybe_unused]]` can be used to avoid warnings by the compiler for not using a name or entity.

The attribute may be applied to the declaration of a class, a type definition with `typedef` or `using`, a variable, a non-static data member, a function, an enumeration type, or an enumerator (enumeration value).

One application is to name a parameter without (necessarily) using it:

```
void foo(int val, [[maybe_unused]] std::string msg)
{
#ifdef DEBUG
  log(msg);
#endif
  ...
}
```

Another example would be to have a member without using it:

```
class MyStruct {
  char c;
  int i;
  [[maybe_unused]] char makeLargerSize[100];
  ...
};
```

Note that you can't apply [[maybe_unused]] to a statement. For this reason, you cannot counter [[nodiscard]] with [[maybe_unused]] directly:[1]

```
[[nodiscard]] void* foo();

int main()
{
  foo();                              // WARNING: return value not used
  [[maybe_unused]] foo();             // ERROR: attribute not allowed here
  [[maybe_unused]] auto x = foo();    // OK
}
```

7.3 Attribute [[fallthrough]]

The new attribute [[fallthrough]] can be used to avoid warnings by the compiler for not having a break statement after a sequence of one or more case labels inside a switch statement.

For example:

```
void commentPlace(int place)
{
  switch (place) {
     case 1:
       std::cout << "very ";
       [[fallthrough]];
     case 2:
       std::cout << "well\n";
       break;
     default:
       std::cout << "OK\n";
       break;
   }
}
```

Here, passing the place 1 will print:

[1] Thanks to Roland Bock for pointing that out.

```
very well
```

using a statement of `case 1` and `case 2`.

Note that the attribute has to be used in an empty statement. Thus, you need a semicolon at its end.

Using the attribute as last statement in a `switch` statement is not allowed.

7.4 General Attribute Extensions

The following features were enabled for attributes in general with C++17:

1. Attributes are now allowed to mark namespaces. For example, you can now deprecate a namespace as follows:

   ```
   namespace [[deprecated]] DraftAPI {
     ...
   }
   ```

   This is also possible for inline and unnamed namespaces.

2. Attributes are now allowed to mark enumerators (values of enumeration types).

   For example, you can introduce a new enumeration value as a replacement of an existing (now deprecated) enumeration value as follows:

   ```
   enum class City { Berlin = 0,
                     NewYork = 1,
                     Mumbai = 2, Bombay [[deprecated]] = Mumbai,
                     ... };
   ```

   Here, both `Mumbai` and `Bombay` represent the same numeric code for a city, but using `Bombay` is marked as deprecated. Note that for enumeration values the attribute is placed *behind* the identifier.

3. For user-defined attributes, which usually should be defined in their own namespace, you can now use a `using` prefix to avoid the repetition of the attribute namespace for each attribute. That is, instead of:

   ```
   [[MyLib::WebService, MyLib::RestService, MyLib::doc("html")]] void foo();
   ```

   you can just write

   ```
   [[using MyLib: WebService, RestService, doc("html")]] void foo();
   ```

   Note that with a `using` prefix using the namespace again is an error:

   ```
   [[using MyLib: MyLib::doc("html")]] void foo();  // ERROR
   ```

7.5 Afternotes

The three new attributes were first proposed by Andrew Tomazos in https://wg21.link/p0068r0. The finally accepted wording for the `[[nodiscard]]` attribute was formulated by Andrew Tomazos in https://wg21.link/p0189r1. The finally accepted wording for the `[[maybe_unused]]`

attribute was formulated by Andrew Tomazos in https://wg21.link/p0212r1. The finally accepted wording for the [[fallthrough]] attribute was formulated by Andrew Tomazos in https://wg21.link/p0188r1.

Allowing attributes for namespaces and enumerators was first proposed by Richard Smith in https://wg21.link/n4196. The finally accepted wording was formulated by Richard Smith in https://wg21.link/n4266.

The using prefix for attributes was first proposed by J. Daniel Garcia, Luis M. Sanchez, Massimo Torquati, Marco Danelutto, and Peter Sommerlad in https://wg21.link/p0028r0. The finally accepted wording was formulated by J. Daniel Garcia and Daveed Vandevoorde in https://wg21.link/P0028R4.

This page is intentionally left blank

Chapter 8

Other Language Features

There are a couple of minor or small changes to the C++ core language, which are described in this chapter.

8.1 Nested Namespaces

Proposed in 2003 for the first time, the C++ standard committee finally accepted to define nested namespaces as follows:

```
namespace A::B::C {
  ...
}
```

which is equivalent to:

```
namespace A {
  namespace B {
    namespace C {
      ...
    }
  }
}
```

Note that there is no support to nest `inline` namespaces. This is simply because it is not obvious whether the `inline` applies to the last or to all namespaces (both could equally be useful).

8.2 Defined Expression Evaluation Order

Many code bases and C++ books contain code that looks valid according to intuitive assumptions, but strictly speaking has undefined behavior. One example is finding and replacing multiple substrings in a string:[1]

```
std::string s = "I heard it even works if you don't believe";

s.replace(0,8,"").replace(s.find("even"),4,"sometimes")
                 .replace(s.find("you don't"),9,"I");
```

The usual assumption is that this code is valid replacing the first 8 characters by nothing, `"even"` by `"sometimes"`, and `"you don't"` by `"I"` so that we get:

```
it sometimes works if I believe
```

However, before C++17, this outcome is not guaranteed, because the `find()` calls, returning where to start with a replacement, might be performed at any time while the whole statement gets processed and before their result is needed. In fact, all `find()` calls, computing the starting index of the replacements, might be processed *before* any of the replacements happens, so that the resulting string becomes:

```
it sometimes works if I believe
```

But other outcomes are also possible:

```
it sometimes workIdon't believe
it even worsometiIdon't believe
it even worsometimesf youIlieve
```

As another example, consider using the output operator to print values computed by expressions that depend on each other:

```
std::cout << f() << g() << h();
```

The usual assumption is that `f()` is called before `g()` and both are called before `h()`. However, this assumption is wrong. `f()`, `g()`, and `h()` might be called in any order, which might have surprising or even nasty effects when these calls depend on each other.

As a concrete example, up to C++17 the following code has undefined behavior:

```
i = 0;
std::cout << ++i << ' ' << --i << '\n';
```

Before C++17, it ***might*** print `1 0`; but it might also print `0 -1` or even `0 0`. It doesn't matter whether `i` is `int` or a user-defined type (for fundamental types, some compilers at least warn about this problem).

[1] A similar example is part of the motivation in the paper proposing the new feature with the comment *This code has been reviewed by C++ experts world-wide, and published (The C++ Programming Language, 4th edition.)*

To fix all this unexpected behavior, for *some* operators the evaluation guarantees were refined so that they now specify a guaranteed evaluation order:

- For

  e1 **[** *e2* **]**
  e1 **.** *e2*
  e1 **.*** *e2*
  e1 **->*** *e2*
  e1 **<<** *e2*
  e1 **>>** *e2*

  e1 is guaranteed to get evaluated before *e2* now, so that the evaluation order is left to right.

  However, note that the evaluation order of different arguments of the same function call is still undefined. That is, in

  ```
  e1.f(a1,a2,a3)
  ```

  `e1` is guaranteed to get evaluated before `a1`, `a2`, and `a3` now. However, the evaluation order of `a1`, `a2`, and `a3` is still undefined.
- In all assignment operators

  e2 **=** *e1*
  e2 **+=** *e1*
  e2 ***=** *e1*
  ...

  the right-hand side *e1* is guaranteed to get evaluated before the left-hand side *e2* now.
- Finally, in `new` expressions like

  new *Type* **(***e***)**

  the allocation is now guaranteed to be performed before the evaluation *e*, and the initialization of the new value is guaranteed to happen before any usage of the allocated and initialized value.

All these guarantees apply to both fundamental types and user-defined types.

As a consequence, since C++17

```
std::string s = "I heard it even works if you don't believe";
s.replace(0,8,"").replace(s.find("even"),4,"always")
                 .replace(s.find("don't believe"),13,"use C++17");
```

is guaranteed to change the value of `s` to:

```
it always works if you use C++17
```

Thus, each replacement in front of a `find()` expression is done before the `find()` expression is evaluated.

As another consequence, for the statements

```
i = 0;
std::cout << ++i << ' ' << --i << '\n';
```

the output is now guaranteed to be `1 0` for any type of `i` that supports these operands.

However, the undefined order for most of the other operators still exists. For example:

```
i = i++ + i;   // still undefined behavior
```

Here, the `i` on the right might be the value of `i` before or after it was incremented.

Another application of the new expression evaluation order is the function that inserts a space before passed arguments.

Backward Incompatibilities

The new guaranteed evaluation order might impact the output of existing programs. This is not just theory. Consider, for example, the following program:

lang/evalexcept.cpp

```
#include <iostream>
#include <vector>

void print10elems(const std::vector<int>& v) {
  for (int i=0; i<10; ++i) {
    std::cout << "value: " << v.at(i) << '\n';
  }
}

int main()
{
  try {
    std::vector<int> vec{7, 14, 21, 28};
    print10elems(vec);
  }
  catch (const std::exception& e) {      // handle standard exception
    std::cerr << "EXCEPTION: " << e.what() << '\n';
  }
  catch (...) {                          // handle any other exception
    std::cerr << "EXCEPTION of unknown type\n";
  }
}
```

Because the `vector<>` in this program only has 4 elements, the program throws an exception in the loop in `print10elems()` when calling `at()` as part of an output statement for an invalid index:

```
std::cout << "value: " << v.at(i) << "\n";
```

Before C++17 the output could be:

```
value: 7
value: 14
value: 21
```

```
value: 28
EXCEPTION: ...
```

because `at()` was allowed to be evaluated before `"value "` was written, so that for the wrong index the output was skipped at all.[2]

Since C++17, the output is guaranteed to be:

```
value: 7
value: 14
value: 21
value: 28
value: EXCEPTION: ...
```

because the output of `"value "` has to be performed before `at()` gets evaluated.

8.3 Relaxed Enum Initialization from Integral Values

For enumerations with a fixed underlying type, since C++17 you can use an integral value of that type for direct list initialization. This applies to unscoped enumerations with a specified type and all scoped enumerations, because they always have an underlying default type:

```
// unscoped enum with underlying type:
enum MyInt : char { };
MyInt i1{42};           // OK since C++17 (ERROR before C++17)
MyInt i2 = 42;          // still ERROR
MyInt i3(42);           // still ERROR
MyInt i4 = {42};        // still ERROR

// scoped enum with default underlying type:
enum class Salutation { mr, mrs };
Salutation s1{0};      // OK since C++17 (ERROR before C++17)
Salutation s2 = 0;     // still ERROR
Salutation s3(0);      // still ERROR
Salutation s4 = {0};   // still ERROR
```

The same applies if `Salutation` has a specified underlying type:

```
// scoped enum with specified underlying type:
enum class Salutation : char { mr, mrs };
Salutation s1{0};      // OK since C++17 (ERROR before C++17)
Salutation s2 = 0;     // still ERROR
Salutation s3(0);      // still ERROR
Salutation s4 = {0};   // still ERROR
```

[2] This was, for example, the behavior of older GCC or Visual C++ versions.

For unscoped enumerations (`enum` without `class`) having *no* specified underlying type, you still can't use list initialization for numeric values:

```
enum Flag { bit1=1, bit2=2, bit3=4 };
Flag f1{0};          // still ERROR
```

Note also that list initialization still doesn't allow narrowing, so you can't pass a floating-point value:

```
enum MyInt : char { };
MyInt i5{42.2};      // still ERROR
```

This feature was motivated to support the trick of defining new integral types just by defining an enumeration type mapping to an existing integral type as done here with `MyInt`. Without the feature, there is no way to initialize a new object without a cast.

In fact, since C++17 the C++ standard library also provides `std::byte`, which directly uses this feature.

8.4 Fixed Direct List Initialization with `auto`

After introducing *uniform initialization* with braces in C++11, it turned out that there were some unfortunate and non-intuitive inconsistencies when using `auto` instead of a specific type:

```
int x{42};       // initializes an int
int y{1,2,3};    // ERROR
auto a{42};      // initializes a std::initializer_list<int>
auto b{1,2,3};   // OK: initializes a std::initializer_list<int>
```

These inconsistencies were fixed for *direct list initialization* (brace initialization without =) so that we now have the following behavior:

```
int x{42};       // initializes an int
int y{1,2,3};    // ERROR
auto a{42};      // initializes an int now
auto b{1,2,3};   // ERROR now
```

Note that this is a **breaking change** that might even silently result in a different program behavior (e.g., when printing `a`). For this reason, compilers that adopt this change usually also adopt this change even in C++11 mode. For the major compilers, the fix was adopted for all modes with Visual Studio 2015, g++ 5, and clang 3.8.

Note also that *copy list initialization* (brace initialization with =) still has the behavior initializing always a `std::initializer_list<>` when `auto` is used:

```
auto c = {42};       // still initializes a std::initializer_list<int>
auto d = {1,2,3};    // still OK: initializes a std::initializer_list<int>
```

Thus, we now have another significant difference between direct initialization (without =) and copy initialization (with =):

```
auto a{42};      // initializes an int now
```

```
auto c = {42};  // still initializes a std::initializer_list<int>
```

The recommended way to initialize variables and objects should always be to use direct list initialization (brace initialization without =).

8.5 Hexadecimal Floating-Point Literals

C++17 standardizes the ability to specify hexadecimal floating-point literals (as some compilers supported already even before C++17). This notation is especially useful when an exact floating-point representation is desired (for decimal floating-point values there is no general guarantee that the exact value exists).

For example:

lang/hexfloat.cpp

```
#include <iostream>
#include <iomanip>

int main()
{
  // init list of floating-point values:
  std::initializer_list<double> values {
                    0x1p4,        // 16
                    0xA,          // 10
                    0xAp2,        // 40
                    5e0,          // 5
                    0x1.4p+2,     // 5
                    1e5,          // 100000
                    0x1.86Ap+16,  // 100000
                    0xC.68p+2,    // 49.625
  };

  // print all values both as decimal and hexadecimal value:
  for (double d : values) {
    std::cout << "dec: " << std::setw(6) << std::defaultfloat << d
              << "  hex: " << std::hexfloat << d << '\n';
  }
}
```

The program defines different floating-point values by using different existing notations and the new hexadecimal floating-point notation. The new notation is a base-2 scientific notation:

- The significand/mantissa is written in hexadecimal format.
- The exponent is written in decimal format and interpreted with respect to base 2.

For example, `0xAp2` is a way to specify the decimal value 40 (10 times 2 to the power of 2). The value could also be expressed as `0x1.4p+5`, which is 1.25 times 32 (0.4 is a hexadecimal quarter and 2 to the power of 5 is 32).

The program has the following output:

```
dec:      16  hex: 0x1p+4
dec:      10  hex: 0x1.4p+3
dec:      40  hex: 0x1.4p+5
dec:       5  hex: 0x1.4p+2
dec:       5  hex: 0x1.4p+2
dec:  100000  hex: 0x1.86ap+16
dec:  100000  hex: 0x1.86ap+16
dec:  49.625  hex: 0x1.8dp+5
```

As you can see in the example program, support for hexadecimal floating-point notation already existed for output streams using the `std::hexfloat` manipulator (available since C++11).

8.6 UTF-8 Character Literals

Since C++11, C++ supports the prefix `u8` for UTF-8 string literals. However, the prefix was not enabled for character literals. C++17 fixes this gap, so that you can write:

```
char c = u8'6';   // character 6 with UTF-8 encoding value
```

This way you guarantee that your character value is the value of the character `'6'` in UTF-8. You can use all 7-bit US-ASCII characters, for which the UTF-8 code has the same value. That is, this specifies to have the correct character value for 7-bit US-ASCII, ISO Latin-1, ISO-8859-15, and the basic Windows character set.[3] Usually, your source code interprets characters in US-ASCII/UTF-8 anyway so that the prefix isn't necessary. The value of `c` will almost always be `54` (hexadecimal `36`).

To give you some background where the prefix might be necessary: For character and string literals in source code, C++ standardizes the characters you can use but not their values. The values depend on the *source character set*. And when the compiler generates the code for the executable program it uses the *execution character set*. The source character set is almost always 7-bit US-ASCII and usually the execution character set is the same; so that in any C++ program all character and string literals (with and without the `u8` prefix) have the same value.

But in very rare scenarios this might not be the case. For example, on old IBM hosts, which (still) use the EBCDIC character set, the character `'6'` would have the value `246` (hexadecimal `F6`) instead. In a program using an EBCDIC character set, the value of the character `c` above would therefore be 246 instead of 54 and running the program on a UTF-8 encoding platform might therefore print the character ö, which is the character with the value of 246 in ASCII (if available). In situations like this the prefix might be necessary.

[3] ISO Latin-1 is formally named ISO-8859-1, while the ISO character set with the European Euro symbol €, ISO-8859-15, is also named ISO Latin-9 (yes, this is not a spelling error).

Note that `u8` can only be used for single characters and characters that have a single byte (code unit) in UTF-8. An initialization such as:

```
char c = u8'ö';
```

is not allowed because the value of the German umlaut `ö` in UTF-8 is a sequence of two bytes, `195` and `182` (hexadecimal `C3 B6`).

As a result both character and string literals now accept the following prefixes:

- `u8` for single-byte US-ASCII and UTF-8 encoding.
- `u` for two-byte UTF-16 encoding.
- `U` for four-byte UTF-32 encoding.
- `l` for wide characters without specific encoding, which might have two or four bytes.

8.7 Exception Specifications as Part of the Type

Since C++17 exception handling specifications became part of the type of a function. That is, the following two functions now have two different types:

```
void f1();
void f2() noexcept;       // different type
```

Before C++17, both functions would have the same type.

As a consequence, at compiler now will detect if you use a function throwing an exception where a function not throwing any exception is required:

```
void (*fp)() noexcept;    // pointer to function that doesn't throw
fp = f2;                  // OK
fp = f1;                  // ERROR since C++17
```

Using function that doesn't throw where functions are allowed to throw is still valid, of course:

```
void (*fp2)();            // pointer to function that might throw
fp2 = f2;                 // OK
fp2 = f1;                 // OK
```

So, the new feature doesn't break programs that didn't use `noexcept` for function pointers yet, but ensures now that you can no longer violate `noexcept` requirements in function pointers (which might break existing programs for a good reason).

It is not allowed to overload a function name for the same signature with a different exception specification (as it is not allowed to overload functions with different return types only):

```
void f3();
void f3() noexcept;       // ERROR
```

Note that all other rules are not affected. For example, it is still the case that you are not allowed to ignore a `noexcept` specification of a base class:

```
class Base {
 public:
  virtual void foo() noexcept;
```

```
  ...
};

class Derived : public Base {
 public:
  void foo() override;  // ERROR: does not override
  ...
};
```

Here, the member function `foo()` in the derived class has a different type so that it does not override the `foo()` of the base class. This code still does not compile. Even without the `override` specifier this code would not compile, because we still can't overload with a looser throw specification.

Using Conditional Exception Specifications

When using conditional noexcept specifications, the type of the functions depends on whether the condition is true or false:

```
void f1();
void f2() noexcept;
void f3() noexcept(sizeof(int)<4);    // same type as either f1() or f2()
void f4() noexcept(sizeof(int)>=4);   // different type than f3()
```

Here, the type of `f3()` depends on the type of the condition when the code gets compiled:

- If `sizeof(int)` yields 4 (or more), the resulting signature is

  ```
  void f3() noexcept(false);    // same type as f1()
  ```

- If `sizeof(int)` yields a value less than 4, the resulting signature is

  ```
  void f3() noexcept(true);     // same type as f2()
  ```

Because the exception condition of `f4()` uses the negated expression of `f3()`, `f4()` always has a different type (i.e., it guarantees to throw if `f3()` doesn't and vice versa).

The "old-fashioned" empty throw specification can still be used but is deprecated since C++17:

```
void f5() throw();    // same as void f5() noexcept but deprecated
```

Dynamic throw specifications are no longer supported (they were deprecated since C++11):

```
void f6() throw(std::bad_alloc);  // ERROR: invalid since C++17
```

Consequences for Generic Libraries

Making `noexcept` declarations part of the type might have some consequences for generic libraries. For example, the following program was valid up to C++14 but no longer compiles with C++17:

lang/noexceptcalls.cpp

```
#include <iostream>
```

```
template<typename T>
void call(T op1, T op2)
{
  op1();
  op2();
}

void f1() {
  std::cout << "f1()\n";
}
void f2() noexcept {
  std::cout << "f2()\n";
}

int main()
{
  call(f1, f2);  // ERROR since C++17
}
```

The problem is that since C++17 `f1()` and `f2()` have different types so that the compiler no longer finds a common type `T` for both types when instantiating the function template `call()`.

With C++17 you have to use two different types if this should still be possible:

```
template<typename T1, typename T2>
void call(T1 op1, T2 op2)
{
  op1();
  op2();
}
```

If you want or have to overload on all possible function types, you also have to double the overloads now. This, for example, applies to the definition of the standard type trait `std::is_function<>`. The primary template is defined so that in general a type `T` is no function:

```
// primary template (in general type T is no function):
template<typename T> struct is_function : std::false_type { };
```

The template derives from `std::false_type` so that `is_function<T>::value` in general yields `false` for any type `T`.

For all types that *are* functions, partial specializations exist, which derive from `std::true_type` so that the member `value` yields `true` for them:

```
// partial specializations for all function types:
template<typename Ret, typename... Params>
struct is_function<Ret (Params...)> : std::true_type { };

template<typename Ret, typename... Params>
```

```
struct is_function<Ret (Params...) const> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) &> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const &> : std::true_type { };
...
```

Before C++17, there were already 24 partial specializations, because function types can have `const` and `volatile` qualifiers as well as lvalue (`&`) and rvalue (`&&`) reference qualifiers, and you need overloads for functions with a variadic list of arguments.

Now, with C++17, the number of partial specialization is doubled by adding a `noexcept` qualifier to all these partial specializations so that we get 48 partial specializations now:

```
...
// partial specializations for all function types with noexcept:
template<typename Ret, typename... Params>
struct is_function<Ret (Params...) noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) & noexcept> : std::true_type { };

template<typename Ret, typename... Params>
struct is_function<Ret (Params...) const& noexcept> : std::true_type { };
...
```

Libraries not implementing the `noexcept` overloads might no longer compile code that uses them to pass functions or function pointers to places where `noexcept` is required.

8.8 Single-Argument `static_assert`

Since C++17, the previously required message argument for `static_assert()` is now optional. This means that the resulting diagnostic message is completely platform specific. For example:

```
#include <type_traits>

template<typename T>
class C {
  // OK since C++11:
  static_assert(std::is_default_constructible<T>::value,
                "class C: elements must be default-constructible");
```

```
  // OK since C++17:
  static_assert(std::is_default_constructible_v<T>);
  ...
};
```

The new assertion without the message also uses the new type traits suffix _v.

8.9 Preprocessor Condition __has_include

C++17 extends the preprocessor to be able to check, whether a specific header file could be included. For example:

```
#if __has_include(<filesystem>)
#  include <filesystem>
#  define HAS_FILESYSTEM 1
#elif __has_include(<experimental/filesystem>)
#  include <experimental/filesystem>
#  define HAS_FILESYSTEM 1
#  define FILESYSTEM_IS_EXPERIMENTAL 1
#elif __has_include("filesystem.hpp")
#  include "filesystem.hpp"
#  define HAS_FILESYSTEM 1
#  define FILESYSTEM_IS_EXPERIMENTAL 1
#else
#  define HAS_FILESYSTEM 0
#endif
```

The conditions inside __has_include(...) evaluate to 1 (true) if a corresponding #include command would be valid. Nothing else matters (e.g., the answer does not depend on whether the file already was included).

8.10 Afternotes

Nested namespace definitions were first proposed in 2003 by Jon Jagger in https://wg21.link/n1524. Robert Kawulak brought up a new proposal in 2014 in https://wg21.link/n4026. The finally accepted wording was formulated by Robert Kawulak and Andrew Tomazos in https://wg21.link/n4230.

The refined expression evaluation order was first proposed by Gabriel Dos Reis, Herb Sutter, and Jonathan Caves in https://wg21.link/n4228. The finally accepted wording was formulated by Gabriel Dos Reis, Herb Sutter, and Jonathan Caves in https://wg21.link/p0145r3.

Relaxed enum initialization was first proposed by Gabriel Dos Reis in https://wg21.link/p0138r0. The finally accepted wording was formulated by Gabriel Dos Reis in https://wg21.link/p0138r2.

Fixing list initialization with `auto` was first proposed by Ville Voutilainen in https://wg21.link/n3681 and https://wg21.link/3912. The final fix for list initialization with `auto` was proposed by James Dennett in https://wg21.link/n3681.

Hexadecimal Floating-Point Literals were first proposed by Thomas Köppe in https://wg21.link/p0245r0. The finally accepted wording was formulated by Thomas Köppe in https://wg21.link/p0245r1.

The prefix for UTF-8 character literals was first proposed by Richard Smith in https://wg21.link/n4197. The finally accepted wording was formulated by Richard Smith in https://wg21.link/n4267.

Making exception specifications part of the function type was first proposed by Jens Maurer in https://wg21.link/n4320. The finally accepted wording was formulated by Jens Maurer in https://wg21.link/p0012r1.

Single-argument `static_assert` was accepted as proposed by Walter E. Brown in https://wg21.link/n3928.

The preprocessor clause `__has_include()` was first proposed by Clark Nelson and Richard Smith as part of https://wg21.link/p0061r0. The finally accepted wording was formulated by Clark Nelson and Richard Smith in https://wg21.link/p0061r1.

Part II

Template Features

This part introduces the new language features C++17 provides for generic programming (i.e., templates).

While we start with class template argument deduction, which also impacts just the usage of templates, the later chapters especially provide feature for programmers of generic code (function templates, class templates, and generic libraries).

This page is intentionally left blank

Chapter 9

Class Template Argument Deduction

Before C++17, you always have to explicitly specify all template parameter types for class templates. For example, you can't omit the `double` here:

```
std::complex<double> c{5.1,3.3};
```

or omit the need to specify `std::mutex` here a second time:

```
std::mutex mx;
std::lock_guard<std::mutex> lg(mx);
```

Since C++17, the constraint that you always have to specify the template arguments explicitly was relaxed. You can skip defining the templates arguments explicitly if the constructor is able to *deduce* all template parameters.

For example:

- You can declare now:

```
std::complex c{5.1,3.3};
```

- You can implement now:

```
std::mutex mx;
std::lock_guard lg(mx);
```

9.1 Usage of Class Template Argument Deduction

Class template argument deduction can be used whenever the arguments passed to a constructor can be used to deduce the class template parameters. The deduction supports all ways of initialization (provided the initialization itself is valid):

```
std::complex c1{1.1, 2.2};  // deduces std::complex<double>
std::complex c2(2.2, 3.3);  // deduces std::complex<double>
```

```
std::complex c3 = 3.3;       // deduces std::complex<double>
std::complex c4 = {4.4};     // deduces std::complex<double>
```

The initialization of `c3` and `c4` is possible, because you can initialize a `std::complex<>` by passing only one argument, which is enough to deduce the template parameter `T`, which is then used for both the real and the imaginary part:

```
namespace std {
  template<typename T>
  class complex {
    constexpr complex(const T& re = T(), const T& im = T());
    ...
  }
};
```

With a declaration such as

```
std::complex c1{1.1, 2.2);
```

the compiler finds the constructor

```
constexpr complex(const T& re = T(), const T& im = T());
```

as possible function to call. Because for both arguments `T` is `double`, the compiler deduces `T` to be `double` and compiles corresponding code for:

```
complex<double>::complex(const double& re = double(),
                         const double& im = double());
```

Note that the template parameter has to be unambiguously deducible. Thus, the following initialization doesn't work:

```
std::complex c5{5,3.3};   // ERROR: attempts to int and double as T
```

As usual for templates there are no type conversions used to deduce template parameters.

Class template argument deduction for variadic templates is also supported. For example, for a `std::tuple<>` which is defined as:

```
namespace std {
  template<typename... Types>
  class tuple;
    public:
      constexpr tuple(const Types&...);
      ...
  };
};
```

the declaration:

```
std::tuple t{42, 'x', nullptr};
```

deduces the type of `t` as `std::tuple<int, char, std::nullptr_t>`.

You can also deduce non-type template parameters. For example, we can deduce template parameters for both the element type and the size from a passed initial array as follows:

```
template<typename T, int SZ>
class MyClass {
  public:
    MyClass (T(&)[SZ]) {
      ...
    }
};

MyClass mc("hello"); // deduces T as const char and SZ as 6
```

Here we deduce 6 as `SZ` because the template parameter passed is a string literals with 6 characters.[1]

You can even deduce the type of lambdas used as base classes for overloading or deduce the type of `auto` template parameters.

9.1.1 Copying by Default

If class template argument deduction could be interpreted as initializing a copy, it prefers this interpretation. For example, after initializing a `std::vector` with one element:

```
std::vector v1{42};          // vector<int> with one element
```

using that vector as initializer for another vector is interpreted to create a copy:

```
std::vector v2{v1};          // v2 also is vector<int>
```

instead of assuming that a vector gets initialized having elements being vectors (`vector<vector<int>>`).

Again, this applies to all valid forms of initialization:

```
std::vector v3(v1);          // v3 also is vector<int>
std::vector v4 = {v1};       // v3 also is vector<int>
auto v5 = std::vector{v1};   // v3 also is vector<int>
```

Only if multiple elements are passed so that this cannot be interpreted as creating a copy, the elements of the initializer list define the element type of the new vector:

```
std::vector vv{v, v};        // vv is vector<vector<int>>
```

This raises the question what happens with class template argument deduction when passing variadic templates:

```
template<typename... Args>
auto make_vector(const Args&... elems) {
  return std::vector{elems...};
}

std::vector<int> v{1, 2, 3};
auto x1 = make_vector(v, v); // vector<vector<int>>
```

[1] Note that passing the initial argument by reference is important here, because otherwise by language rules the constructor declares a pointer so that `SZ` can't be deduced.

```
auto x2 = make_vector(v);     // vector<int> or vector<vector<int>> ?
```

Currently, different compilers handle this differently and the issue is under discussion.

9.1.2 Deducing the Type of Lambdas

With class template argument deduction, for the first time we can instantiate class templates with the type of a lambda (to be exact: the *closure type* of a lambda). For example, we could provide a generic class, wrapping and counting call of an arbitrary callback:

tmpl/classarglambda.hpp

```
#include <utility>      // for std::forward()

template<typename CB>
class CountCalls
{
  private:
    CB callback;        // callback to call
    long calls = 0;     // counter for calls
  public:
    CountCalls(CB cb) : callback(cb) {
    }
    template<typename... Args>
    auto operator() (Args&&... args) {
      ++calls;
      return callback(std::forward<Args>(args)...);
    }
    long count() const {
      return calls;
    }
};
```

Here, the constructor, taking the callback to wrap, enables to deduce its type as template parameter `CB`. For example, we can initialize an object passing a lambda as argument:

```
CountCalls sc([](auto x, auto y) {
                  return x > y;
              });
```

which means that the type of the sorting criterion `sc` is deduced as `CountCalls<`*TypeOfTheLambda*`>`. This way, we can for example count the number of calls for a passed sorting criterion:

```
std::sort(v.begin(), v.end(),
          std::ref(sc));
std::cout << "sorted with " << sc.count() << " calls\n";
```

Here, the wrapped lambda is used as sorting criterion, which however has to be passed by reference, because otherwise `std::sort()` only uses the counter of its own copy of the passed counter, because `std::sort()` itself takes the sorting criterion by value.

However, we can pass a wrapped lambda to `std::for_each()`, because this algorithm (in the non-parallel version) returns its own copy of the passed callback to be able to use its resulting state:

```
auto fo = std::for_each(v.begin(), v.end(),
                        CountCalls([](auto i) {
                                     std::cout << "elem: " << i << '\n';
                                   }));
std::cout << "output with " << fo.count() << " calls\n";
```

9.1.3 No Partial Class Template Argument Deduction

Note that, unlike function templates, class template arguments may not be partially deduced (by explicitly specifying only *some* of the template arguments). For example:

```
template<typename T1, typename T2, typename T3 = T2>
class C
{
  public:
    C (T1 x = T1{}, T2 y = T2{}, T3 z = T3{}) {
      ...
    }
    ...
};

// all deduced:
C c1(22, 44.3, "hi");       // OK: T1 is int, T2 is double, T3 is const char*
C c2(22, 44.3);             // OK: T1 is int, T2 and T3 are double
C c3("hi", "guy");          // OK: T1, T2, and T3 are const char*

// only some deduced:
C<string> c4("hi", "my");   // ERROR: only T1 explicitly defined
C<> c5(22, 44.3);           // ERROR: neither T1 not T2 explicitly defined
C<> c6(22, 44.3, 42);       // ERROR: neither T1 nor T2 explicitly defined

// all specified:
C<string,string,int> c7;              // OK: T1,T2 are string, T3 is int
C<int,string> c8(52, "my");           // OK: T1 is int,T2 and T3 are strings
C<string,string> c9("a", "b", "c");   // OK: T1,T2,T3 are strings
```

Note that the third template parameter has a default value. For this reason, it is unnecessary to explicitly specify the last type if the second type is specified.

If you wonder why partial specialization is not supported, here is the example that caused this decision:

```
std::tuple<int> t(42, 43);   // still ERROR
```

`std::tuple` is a variadic template, so you could specific an arbitrary number of arguments. So, in this case it is not clear whether it is an error to specify only one type, or whether this is intentional. It looks at least questionable. Partial specialization can still be added later to standard C++, after taking more time to think about it.

Unfortunately, the lack of the ability to partially specialize means that a common unfortunate coding requirement is not solved. We still can't easily use a lambda to specify the sorting criterion of an associative container or the hash function of an unordered container:

```
std::set<Cust> coll([](const Cust& x, const Cust& y) {   // still ERROR
                       return x.name() > y.name();
                    });
```

We still also have to specify the type of the lambda, so that we for example need the following:[2]

```
auto sortcrit = [](const Cust& x, const Cust& y) {
                    return x.name() > y.name();
                };
std::set<Cust, decltype(sortcrit)> coll(sortcrit);        // OK
```

9.1.4 Class Template Argument Deduction Instead of Convenience Functions

By using class template argument deduction in principle we can get rid of several convenience function templates that only existed to be able to deduce the type of a class from the passed call arguments.

The obvious example is `make_pair()`, which allowed to avoid the need specify the type of the passed arguments. For example, after:

```
std::vector<int> v;
```

we could use:

```
auto p = std::make_pair(v.begin(), v.end());
```

instead of writing.

```
std::pair<typename std::vector<int>::iterator,
          typename std::vector<int>::iterator> p(v.begin(), v.end());
```

Here, `make_pair()` is no longer needed, as we can simply declare now:

```
std::pair p(v.begin(), v.end());
```

However, `std::make_pair()` is also a good example to demonstrate that sometimes the convenience functions did more than just deducing template parameters. In fact, `std::make_pair()`

[2] Specifying the type only doesn't work because then the container tries to create a lambda of the given type, which is not allowed, because the default constructor is only callable by the compiler. With C++20 this will probably be possible.

also decays, which especially means that the type of passed string literals is converted to `const char*`:

```
auto q = std::make_pair("hi", "world"); // pair of pointers
```

In this case, `q` has type `std::pair<const char*, const char*>`.

By using class template argument deduction, things get more complicated. Let's look at the relevant part of a simple class declaration like `std::pair`:

```
template<typename T1, typename T2>
struct Pair1 {
  T1 first;
  T2 second;
  Pair1(const T1& x, const T2& y) : first{x}, second{y} {
  }
};
```

The point is that the elements are passed by reference. And by language rules, when passing arguments of a template type, by reference, the parameter type doesn't *decay*, which is the term for the mechanism to convert a raw array type to the corresponding raw pointer type. So, when calling:

```
Pair1 p1{"hi", "world"};   // deduces pair of arrays of different size, but...
```

`T1` is deduced as `char[3]` and `T2` is deduced as `char[6]`. In principle, such a deduction is valid. However, we use `T1` and `T2` to declare the members `first` and `second`. As a consequence, they are declared as

```
char first[3];
char second[6];
```

and initializing an array from an lvalue of an array is not allowed. It's like trying to compile:

```
const char x[3] = "hi";
const char y[6] = "world";
char first[3] {x};          // ERROR
char second[6] {y};         // ERROR
```

Note that we wouldn't have this problem when declaring the parameter to be passed by value:

```
template<typename T1, typename T2>
struct Pair2 {
  T1 first;
  T2 second;
  Pair2(T1 x, T2 y) : first{x}, second{y} {
  }
};
```

If for this type we'd call:

```
Pair2 p2{"hi", "world"};   // deduces pair of pointers
```

`T1` and `T2` both would be deduced as `const char*`.

Because class `std::pair<>` is declared so that the constructors take the arguments by reference, you might expect now the following initialization not to compile:

```
std::pair p{"hi", "world"};   // seems to deduce pair of arrays of different size, but...
```

But it compiles. The reason is that we use *deduction guides*.

9.2 Deduction Guides

You can define specific *deduction guides* to provide additional class template argument deductions or fix existing deductions defined by constructors. For example, you can define that whenever the types of a `Pair3` are deduced, the type deduction should operate as if the types had been passed by value:

```
template<typename T1, typename T2>
struct Pair3 {
  T1 first;
  T2 second;
  Pair3(const T1& x, const T2& y) : first{x}, second{y} {
  }
};

// deduction guide for the constructor:
template<typename T1, typename T2>
Pair3(T1, T2) -> Pair3<T1, T2>;
```

Here, on the left side of the `->` we declare *what* we want to deduce. In this case, it is the creation of a `Pair3` from two objects of arbitrary types `T1` and `T2` passed by value. On the right side of the `->` we define the resulting deduction. In this example, `Pair3` is instantiated with the two types `T1` and `T2`.

You might argue that this is what the constructor already does. However, the constructor takes the argument by reference, which is not the same. In general, even outside of templates, arguments passed by value *decay*, while arguments passed by reference do not decay. *Decay* means that raw arrays convert to pointers and top-level qualifiers, such as `const` and references, are ignored.

Without the deduction guide, for example, when declaring the following:

```
Pair3 p3{"hi", "world"};
```

the type of parameter `x` and therefore `T1` is `const char[3]` and the type of parameter `y` and therefore `T2` is `const char[6]`.

Due to the deduction guide, the template parameters decay, which means that passed array or string literals decay to the corresponding pointer types. Now, when declaring the following:

```
Pair3 p3{"hi", "world"};
```

the deduction guide is used, which takes the parameters by value so that both types decay to `const char*`. The declaration has the effect as if we'd have declared:

```
Pair3<const char*, const char*> p3{"hi", "world"};
```

Note that still the constructor takes the arguments by reference. The deduction guide only matters for the deduction of the template types. It is irrelevant for the actual constructor call after the types `T1` and `T2` are deduced.

9.2.1 Using Deduction Guides to Force Decay

As the previous example demonstrates, in general, a very useful application of these overloading rules is to ensure that a template parameter `T` *decays* while it is deduced. Consider a typical class template:

```
template<typename T>
struct C {
  C(const T&) {
  }
  ...
};
```

If we here pass a string literal `"hello"`, `T` is deduced as the type of the string literal, which is `const char[6]`:

```
C x{"hello"};     // T deduced as const char[6]
```

The reason is that template parameter deduction does not *decay* to the corresponding pointer type, when arguments are passed by reference.

With a simple deduction guide

```
template<typename T> C(T) -> C<T>;
```

we fix this problem:

```
C x{"hello"};     // T deduced as const char*
```

Now, because the deduction guide takes it argument by value, its type decays, so that `"hello"` deduces `T` to be of type `const char*`.

For this reason, a corresponding deduction guide sounds very reasonable for any class template having a constructor taking an object of its template parameter by reference. The C++ standard library provides corresponding deduction guides for pairs and tuples.

9.2.2 Non-Template Deduction Guides

Deduction guides don't have to be templates and don't have to apply to constructors. For example, given the following structure and deduction guide:

```
template<typename T>
struct S {
  T val;
};

S(const char*) -> S<std::string>;  // map S<> for string literals to S<std::string>
```

the following declarations are possible, where `std::string` is deduced as type of `T` from `const char*` because the passed string literal implicitly converts to it:

```
S s1{"hello"};          // OK, same as: S<std::string> s1{"hello"};
S s2 = {"hello"};       // OK, same as: S<std::string> s2 = {"hello"};
S s3 = S{"hello"};      // OK, both S deduced to be S<std::string>
```

Note that aggregates need list initialization (the deduction works, but the initialization is not allowed):

```
S s4 = "hello";         // ERROR (can't initialize aggregates that way)
```

9.2.3 Deduction Guides versus Constructors

Deduction guides compete with the constructors of a class. Class template argument deduction uses the constructor/guide that has the highest priority according to overload resolution. If a constructor and a deduction guide match equally well, the deduction guide is preferred.

Consider we have the following definition:

```
template<typename T>
struct C1 {
  C1(const T&) {
  }
};
C1(int) -> C1<long>;
```

When passing an `int`, the deduction guide is used, because it is preferred by overload resolution.[3] Thus, `T` is deduced as `long`:

```
C1 x1{42};      // T deduced as long
```

But if we pass a `char`, the constructor is a better match (because no type conversion is necessary), so that we deduce `T` to be `char`:

```
C1 x3{'x'};     // T deduced as char
```

Because taking argument by value matches equally well as taking arguments by references and deduction guides are preferred for equally well matches, it is usually fine to let the deduction guide take the argument by value (which also has the advantage to decay).

9.2.4 Explicit Deduction Guides

A deduction guide can be declared as to be `explicit`. It is then ignored only for the cases, where the `explicit` would disable initializations or conversions. For example, given:

```
template<typename T>
struct S {
  T val;
```

[3] A non-template function is preferred over a template unless other aspects of overload resolution matter more.

```
};

explicit S(const char*) -> S<std::string>;
```

a copy initialization (using the =) of an S object passing the type of deduction guide argument ignores the deduction guide. Here, it means that the initialization becomes invalid:

```
S s1 = {"hello"};           // ERROR (deduction guide ignored and otherwise invalid)
```

Direct initialization or having an explicit deduction on the right-hand side is still possible:

```
S s2{"hello"};              // OK, same as: S<std::string> s1{"hello"};
S s3 = S{"hello"};          // OK
S s4 = {S{"hello"}};        // OK
```

As another example, we could do the following:

```
template<typename T>
struct Ptr
{
  Ptr(T) { std::cout << "Ptr(T)\n"; }
  template<typename U>
  Ptr(U) { std::cout << "Ptr(U)\n"; }
};

template<typename T>
explicit Ptr(T) -> Ptr<T*>;
```

which would have the following effect:

```
Ptr p1{42};    // deduces Ptr<int*> due to deduction guide
Ptr p2 = 42;   // deduces Ptr<int> due to constructor
int i = 42;
Ptr p3{&i};    // deduces Ptr<int**> due to deduction guide
Ptr p4 = &i;   // deduces Ptr<int*> due to constructor
```

9.2.5 Deduction Guides for Aggregates

Deduction guides can be used in generic aggregates to enable class template argument deduction there. For example, for:

```
template<typename T>
struct A {
  T val;
};
```

any trial of class template argument deduction without a deduction guide is an error:

```
A i1{42};          // ERROR
A s1("hi");        // ERROR
A s2{"hi"};        // ERROR
```

```
A s3 = "hi";      // ERROR
A s4 = {"hi"};    // ERROR
```

You have to pass the argument for type `T` explicitly:

```
A<int> i2{42};
A<std::string> s5 = {"hi"};
```

But after a deduction guide such as:

```
A(const char*) -> A<std::string>;
```

you can initialize the aggregate as follows:

```
A s2{"hi"};       // OK
A s4 = {"hi"};    // OK
```

However, as usual for aggregates, you still need curly braces. Otherwise, type `T` is successfully deduced, but the initialization is an error:

```
A s1("hi");       // ERROR: T is string, but no aggregate initialization
A s3 = "hi";      // ERROR: T is string, but no aggregate initialization
```

The deduction guides for `std::array` are another example of deduction guides for aggregates.

9.2.6 Standard Deduction Guides

The C++ standard library introduces a couple of deduction guides with C++17.

Deduction Guides for Pairs and Tuples

As introduced in motivation of deduction guides `std::pair` needs deduction guides to ensure that class template argument deduction uses the decayed type of the passed argument:[4]

```
namespace std {
  template<typename T1, typename T2>
  struct pair {
    ...
    constexpr pair(const T1& x, const T2& y);   // take argument by-reference
    ...
  };

  template<typename T1, typename T2>
    pair(T1, T2) -> pair<T1, T2>;               // deduce argument types by-value
}
```

As a consequence, the declaration

```
std::pair p{"hi", "world"};        // takes const char[3] and const char[6]
```

[4] The original declaration uses `class` instead of `typename` and declared the constructors as conditionally `explicit`.

is equivalent to:

```
std::pair<const char*, const char*> p{"hi", "world"};
```

For the variadic class template `std::tuple`, the same approach is used:

```
namespace std {
  template<typename... Types>
  class tuple {
   public:
    constexpr tuple(const Types&...);          // take arguments by-reference
    template<typename... UTypes> constexpr tuple(UTypes&&...);
    ...
  };

  template<typename... Types>
    tuple(Types...) -> tuple<Types...>;        // deduce argument types by-value
};
```

As a consequence, the declaration:

```
std::tuple t{42, "hello", nullptr};
```

deduces the type of `t` as `std::tuple<int, const char*, std::nullptr_t>`.

Deduction from Iterators

To be able to deduce the type of the elements from iterators that define a range for initialization, containers have a deduction guide such as the following for `std::vector<>`:

```
// let std::vector<> deduce element type from initializing iterators:
namespace std {
  template<typename Iterator>
  vector(Iterator, Iterator)
    -> vector<typename iterator_traits<Iterator>::value_type>;
}
```

This allows, for example:

```
std::set<float> s;
std::vector(s.begin(), s.end());    // OK, deduces std::vector<float>
```

`std::array<>` Deduction

A more interesting example provides class `std::array<>`: To be able to deduce both the element type and the number of elements:

```
std::array a{42,45,77};             // OK, deduces std::array<int,3>
```

the following deduction guide is defined:

```
// let std::array<> deduce their number of elements (must have same type):
```

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

The deduction guide uses the fold expression

```
(is_same_v<T,U> && ...)
```

to ensure that the types of all passed arguments are the same.[5] Thus, the following is not possible:

```
std::array a{42,45,77.7};          // ERROR: types differ
```

(Unordered) Map Deduction

The complexity involved in getting deduction guides that behave correctly can be demonstrated by the trials to define deduction guides for containers that have key/value pairs (`map`, `multimap`, `unordered_map`, `unordered_multimap`).

The elements of these containers have type `std::pair<const` *keytype*`,` *valuetype*`>`. The `const` is necessary, because the location of an element depends on the value of the key, so that the ability to modify the key could create inconsistencies inside the container.

So, the approach in the C++17 standard for a `std::map`:

```
namespace std {
  template<typename Key, typename T,
           typename Compare = less<Key>,
           typename Allocator = allocator<pair<const Key, T>>>
  class map {
    ...
  };
}
```

was to define, for example, for the following constructor:

```
    map(initializer_list<pair<const Key, T>>,
        const Compare& = Compare(),
        const Allocator& = Allocator());
```

the following deduction guide:

```
namespace std {
  template<typename Key, typename T,
           typename Compare = less<Key>,
           typename Allocator = allocator<pair<const Key, T>>>
  map(initializer_list<pair<const Key, T>>,
```

[5] We discussed to allow implicit type conversions, but we decided to be conservative here.

```
      Compare = Compare(),
      Allocator = Allocator())
    -> map<Key, T, Compare, Allocator>;
}
```

As all arguments are passed by value, this deduction guide enables that the type of a passed comparator or allocator decays as discussed. However, we naively used the same arguments types, which meant that the initializer list takes a `const` key type. But as a consequence, the following didn't work as Ville Voutilainen pointed out in https://wg21.link/lwg3025:

```
std::pair elem1{1,2};
std::pair elem2{3,4};
...
std::map m1{elem1, elem2};          // ERROR with original C++17 guides
```

because here the elements are deduced as `std::pair<int,int>`, which does not match the deduction guide requiring a `const` type as first pair type. So, you still had to write the following:

```
std::map<int,int> m1{elem1, elem2};  // OK
```

As a consequence, in the deduction guide the `const` should be removed:

```
namespace std {
  template<typename Key, typename T,
           typename Compare = less<Key>,
           typename Allocator = allocator<pair<const Key, T>>>
  map(initializer_list<pair<Key, T>>,
      Compare = Compare(),
      Allocator = Allocator())
    -> map<Key, T, Compare, Allocator>;
}
```

However, to still support the decay of the comparator and allocator we also have to overload the deduction guide for a pair with `const` key type. Otherwise the constructor would be used so that the behavior for class template argument deduction would slightly differ when pairs with `const` and non-`const` keys are passed.

No Deductions Guides for Smart Pointers

Note that some places in the C++ standard library don't have deductions guides although you might expect them to be available.

You might, for example, expect to have deduction guides for shared and unique pointers, so that instead of:

```
std::shared_ptr<int> sp{new int(7)};
```

you just could write:

```
std::shared_ptr sp{new int(7)};      // not supported
```

This doesn't work automatically, because the corresponding constructor is a template, so that no implicit deduction guide applies:

```
namespace std {
  template<typename T> class shared_ptr {
    public:
      ...
      template<typename Y> explicit shared_ptr(Y* p);
      ...
  };
}
```

`Y` is a different template parameter than `T` so that deducing `Y` from the constructor does not mean that we can deduce type `T`. This is a feature to be able to call something like:

```
std::shared_ptr<Base> sp{new Derived(...)};
```

The corresponding deduction guide would be simple to provide:

```
namespace std{
  template<typename Y> shared_ptr(Y*) -> shared_ptr<Y>;
}
```

However, this would also mean that this guide is taken when allocating arrays:

```
std::shared_ptr sp{new int[10]};      // OOPS: would deduces shared_ptr<int>
```

As so often in C++, we run into the nasty C problem that the type of a pointer to one object and an array of objects have or decay to the same type.

Because this problem seems to be dangerous, the C++ standard committee decided not to support it (yet). You still have to call for single objects:

```
std::shared_ptr<int> sp1{new int};        // OK
auto sp2 = std::make_shared<int>();       // OK
```

and for arrays:

```
std::shared_ptr<std::string> p(new std::string[10],
                               [](std::string* p) {
                                   delete[] p;
                               });
```

or:

```
std::shared_ptr<std::string> p(new std::string[10],
                               std::default_delete<std::string[]>());
```

9.3 Afternotes

Class template argument deduction was first proposed in 2007 by Michael Spertus in https://wg21.link/n2332. The proposal came back in 2013 by Michael Spertus and David Vandevoorde in https://wg21.link/n3602. The finally accepted wording was formulated by Michael

Spertus, Faisal Vali, and Richard Smith in https://wg21.link/p0091r3 with modifications by Michael Spertus, Faisal Vali, and Richard Smith in https://wg21.link/p0512r0, by Jason Merrill in https://wg21.link/p0620r0, and by Michael Spertus and Jason Merrill (as a defect report against C++17) in https://wg21.link/p702r1.

The support for class template argument deduction in the standard library was added by Michael Spertus, Walter E. Brown, and Stephan T. Lavavej in https://wg21.link/p0433r2 and (as a defect report against C++17) in https://wg21.link/p0739r0.

This page is intentionally left blank

Chapter 10

Compile-Time `if`

With the syntax `if constexpr(...)`, the compiler uses a compile-time expression to decide at compile time whether to use the *then* part or the *else* part (if any) of an `if` statement. The other part (if any) gets discarded, so that no code gets generated. This does not mean that the discarded part it is completely ignored, though. It will be checked like code of unused templates.

For example:

tmpl/ifcomptime.hpp

```
#include <string>

template <typename T>
std::string asString(T x)
{
  if constexpr(std::is_same_v<T, std::string>) {
    return x;                   // statement invalid, if no conversion to string
  }
  else if constexpr(std::is_arithmetic_v<T>) {
    return std::to_string(x);   // statement invalid, if x is not numeric
  }
  else {
    return std::string(x);      // statement invalid, if no conversion to string
  }
}
```

Here, we use this feature to decide at compile time whether we just return a passed string, call `std::to_string()` for a passed integral or floating-point value, or try to convert the passed argument to `std::string`. Because the invalid calls are *discarded*, the following code compiles (which would not be the case when using a regular run-time `if`):

tmpl/ifcomptime.cpp

```
#include "ifcomptime.hpp"
#include <iostream>

int main()
{
  std::cout << asString(42) << '\n';
  std::cout << asString(std::string("hello")) << '\n';
  std::cout << asString("hello") << '\n';
}
```

10.1 Motivation for Compile-Time `if`

If we'd use the run-time `if` in the example just introduced:

tmpl/ifruntime.hpp

```
#include <string>

template <typename T>
std::string asString(T x)
{
  if (std::is_same_v<T, std::string>) {
    return x;                   // ERROR, if no conversion to string
  }
  else if (std::is_numeric_v<T>) {
    return std::to_string(x);   // ERROR, if x is not numeric
  }
  else {
    return std::string(x);      // ERROR, if no conversion to string
  }
}
```

the corresponding code would *never* compile. This is a consequence of the rule that function templates usually are either not compiled or compiled as a whole. The check of the `if` condition is a run-time feature. Even if at compile-time it becomes clear that a condition must be `false`, the *then* part must be able to compile. So, when passing a `std::string` or string literal, the compilation fails, because the call of `std::to_string()` for the passed argument is not valid. And when passing a numeric value, the compilation fails, because the third and third return statements would be invalid.

Now and only by using the compile-time `if`, the *then* and *else* parts that can't be used become *discarded statements*:

- When passing a `std::string` value, the *else* part of the first `if` gets discarded.
- When passing a numeric value, the *then* part of the first `if` and the final *else* part get discarded.

- When passing a string literal (i.e., type `const char*`), the *then* parts of the first and second `if` get discarded.

So, each invalid combination can't occur any longer at compile-time and the code compiles successfully.

Note that a discarded statement is not ignored. The effect is that it doesn't get instantiated, when depending on template parameters. The syntax must be correct and calls that don't depend on template parameters must be valid. In fact, the first translation phase (the *definition time*) is performed, which checks for correct syntax and the usage of all names that don't depend on template parameters. All `static_asserts` must also be valid, even in branches that aren't compiled.

For example:

```
template<typename T>
void foo(T t)
{
  if constexpr(std::is_integral_v<T>) {
    if (t > 0) {
      foo(t-1);  // OK
    }
  }
  else {
    undeclared(t);    // error if not declared and not discarded (i.e., T is not integral)
    undeclared();     // error if not declared (even if discarded)
    static_assert(false, "no integral");  // always asserts (even if discarded)
  }
}
```

With a conforming compiler, this example *never* compiles for two reasons:

- Even if `T` is an integral type, the call of

  ```
  undeclared();     // error if not declared (even if discarded)
  ```

  in the discarded *else* part is an error if no such function is declared, because this call doesn't depend on a template parameter.
- The call of

  ```
  static_assert(false, "no integral");  // always asserts (even if discarded)
  ```

  always falls even if it is part of the discarded *else* part, because again this call doesn't depend on a template parameter. A static assertion repeating the compile-time condition would be fine:

  ```
  static_assert(!std::is_integral_v<T>, "no integral");
  ```

Note that some compilers (e.g., Visual C++ 2013 and 2015) do not implement or perform the two-phase translation of templates correctly. They defer most of the first phase (the *definition time*) to

the second phase (the *instantiation time*) so invalid function calls and even some syntax errors might compile.[1]

10.2 Using Compile-Time `if`

In principle, you can use the compile-time `if` like the run-time `if` provided the condition is a compile-time expression. You can also mix compile-time and run-time `if`:

```
if constexpr (std::is_integral_v<std::remove_reference_t<T>>) {
  if (val > 10) {
    if constexpr (std::numeric_limits<char>::is_signed) {
      ...
    }
    else {
      ...
    }
  }
  else {
    ...
  }
}
else {
  ...
}
```

Note that you cannot use `if constexpr` outside function bodies. Thus, you can't use it to replace conditional preprocessor directives.

10.2.1 Caveats for Compile-Time `if`

Even when it is possible to use compile-time `if` there might be some consequences that are not obvious, which are discussed in the following subsections.[2]

Compile-Time `if` Impacts the Return Type

Compile-time `if` might impact the return type of a function. For example, the following code always compiles, but the return type might differ:

```
auto foo()
{
```

[1] Visual C++ is on the way to fix this behavior step-by-step, which, however, requires specific options such as `/permissive-`, because it might break existing code.

[2] Thanks to Graham Haynes, Paul Reilly, and Barry Revzin for bringing all these aspects of compile-time `if` to attention.

```
  if constexpr (sizeof(int) > 4) {
    return 42;
  }
  else {
    return 42u;
  }
}
```

Here, because we use `auto`, the return type of the function depends on the return statements, which depend on the size of `int`:

- If the size is greater than 4, there is only one valid return statement returning `42`, so that the return type is `int`.
- Otherwise, there is only one return statement returning `42u`, so that the return type becomes `unsigned int`.

This way the return type of a function with `if constexpr` might differ even more dramatically. For example, if we skip the *else* part the return type might be `int` or `void`:

```
auto foo()                              // return type might be int or void
{
  if constexpr (sizeof(int) > 4) {
    return 42;
  }
}
```

Note that this code never compiles if the run-time `if` is used here, because then both return statements are taken into account so that the deduction of the return type is ambiguous.

`else` Matters Even if `then` Returns

For run-time `if` statements there is a pattern that does not apply to compile-time `if` statements: If code with return statements in both the *then* and the *else* part compiles, you can always skip the `else` in the run-time `if` statements. That is, instead of

```
if (...) {
  return a;
}
else {
  return b;
}
```

you can always write:

```
if (...) {
  return a;
}
return b;
```

This pattern does not apply to compile-time `if`, because in the second form the return type depends on two return statements instead of one, which can make a difference. For example, modifying the example above results in code that *might or might not* compile:

```
auto foo()
{
  if constexpr (sizeof(int) > 4) {
    return 42;
  }
  return 42u;
}
```

If the condition is true (the size of `int` is greater than 4), the compiler deduces two different return types, which is not valid. Otherwise, we have only one return statement that matters, so that the code compiles.

Short-Circuit Compile-Time Conditions

Consider the following code:

```
template<typename T>
constexpr auto foo(const T& val)
{
  if constexpr (std::is_integral<T>::value) {
    if constexpr (T{} < 10) {
      return val * 2;
    }
  }
  return val;
}
```

Here we have two compile-time conditions to decide whether to return the passed value as it is or doubled.

This compiles for both:

```
constexpr auto x1 = foo(42);     // yields 84
constexpr auto x2 = foo("hi");   // OK, yields "hi"
```

Conditions in run-time `if`s short-circuit (evaluating conditions with `&&` only until the first `false` and conditions with `||` only until the first `true`). Which might result in the expectation that this is also the case for compile-time `if`:

```
template<typename T>
constexpr auto bar(const T& val)
{
  if constexpr (std::is_integral<T>::value && T{} < 10) {
      return val * 2;
  }
  return val;
```

```
}
```

However, the condition for the the compile-time `if` is always instantiated and needs to be valid as a whole, so that passing a type that doesn't support `<10` no longer compiles:

```
constexpr auto x2 = bar("hi");  // compile-time ERROR
```

So, compile-time `if` does *not* short-circuit the instantiations. If the validity of compile-time conditions depend on earlier compile-time conditions, you have to nest them as done in `foo()`. As another example, you have to write:[3]

```
if constexpr (std::is_same_v<MyType, T>) {
  if constexpr (T::i == 42) {
    ...
  }
}
```

instead of just:

```
if constexpr (std::is_same_v<MyType, T> && T::i == 42) {
  ...
}
```

10.2.2 Other Compile-Time `if` Examples

Perfect Return of a Generic Value

One application of compile-time `if` is the perfect forwarding of return values, when they have to get processed before they can be returned. Because `decltype(auto)` can't be deduced for `void` (because `void` is an incomplete type), you have to write something like the following:

tmpl/perfectreturn.hpp

```
#include <functional>   // for std::forward()
#include <type_traits>  // for std::is_same<> and std::invoke_result<>

template<typename Callable, typename... Args>
decltype(auto) call(Callable op, Args&&... args)
{
  if constexpr(std::is_void_v<std::invoke_result_t<Callable, Args...>>) {
    // return type is void:
    op(std::forward<Args>(args)...);
    ...  // do something before we return
    return;
```

[3] For the discussion about this example, see: https://groups.google.com/a/isocpp.org/forum/#!msg/std-proposals/eiBAIoynhrM/Y_iPP6aNBgAJ

```
  }
  else {
    // return type is not void:
    decltype(auto) ret{op(std::forward<Args>(args)...)};
    ... // do something (with ret) before we return
    return ret;
  }
}
```

Compile-Time `if` for Tag Dispatching

A typical application of compile-time `if` is tag dispatching. Before C+17, you had to provide an overload set with a separate function for each type you wanted to handle. Now, with compile-time `if`, you can put all the logic together in one function.

For example, instead of overloading the `std::advance()` algorithm:

```
template<typename Iterator, typename Distance>
void advance(Iterator& pos, Distance n) {
  using cat = std::iterator_traits<Iterator>::iterator_category;
  advanceImpl(pos, n, cat);    // tag dispatch over iterator category
}

template<typename Iterator, typename Distance>
void advanceImpl(Iterator& pos, Distance n,
                 std::random_access_iterator_tag) {
  pos += n;
}

template<typename Iterator, typename Distance>
void advanceImpl(Iterator& pos, Distance n,
                 std::bidirectional_iterator_tag) {
  if (n >= 0) {
    while (n--) {
      ++pos;
    }
  }
  else {
    while (n++) {
      --pos;
    }
  }
}
```

```
template<typename Iterator, typename Distance>
void advanceImpl(Iterator& pos, Distance n, std::input_iterator_tag) {
  while (n--) {
    ++pos;
  }
}
```

we can now implement all behavior in one function:

```
template<typename Iterator, typename Distance>
void advance(Iterator& pos, Distance n) {
  using cat = std::iterator_traits<Iterator>::iterator_category;

  if constexpr (std::is_same_v<cat, std::random_access_iterator_tag>) {
    pos += n;
  }
  else if constexpr (std::is_same_v<cat,
                       std::bidirectional_access_iterator_tag>) {
    if (n >= 0) {
      while (n--) {
        ++pos;
      }
    }
    else {
      while (n++) {
        --pos;
      }
    }
  }
  else {  //input_iterator_tag
    while (n--) {
      ++pos;
    }
  }
}
```

So, to some extent, we have a compile-time `switch` now, where the different cases have to get formulated by `if constexpr` clauses, though. However, note one difference that might matter:[4]

- The set of overloaded functions gives you **best match** semantics.
- The implementation with compile-time `if` gives you **first match** semantics.

Another example of tag dispatching is the use of compile-time `if` for `get<>()` overloads to implement a structure bindings interface.

[4] Thanks to Graham Haynes and Barry Revzin for pointing that out.

A third example is the handling of different types in a generic lambda as in std::variant<> visitors.

10.3 Compile-Time if with Initialization

Note that the compile-time if can also use the new form of if with initialization. For example, if there is a constexpr function foo(), you can use:

```
template<typename T>
void bar(const T x)
{
  if constexpr (auto obj = foo(x); std::is_same_v<decltype(obj), T>) {
    std::cout << "foo(x) yields same type\n";
    ...
  }
  else {
    std::cout << "foo(x) yields different type\n";
    ...
  }
}
```

If there is a constexpr function foo() for a passed type you can use this code to provide different behavior on whether foo(x) yields the same type as x.

To decide on the value returned by foo(x) you can write:

```
constexpr auto c = ...;
if constexpr (constexpr auto obj = foo(c); obj == 0) {
  std::cout << "foo() == 0\n";
  ...
}
```

Note that obj has to get declared as constexpr to use its value in the condition.

10.4 Using Compile-Time if Outside Templates

if constexpr can be used in any function, not only in templates. We only need a compile-time expression that yields something convertible to bool. However, in that case in both the *then* and the *else* parts all statements always have to be valid even if discarded.

For example, the following code will always fail to compile, because the call of undeclared() must be valid even if chars are signed and the *else* part is discarded:

```
#include <limits>

template<typename T>
void foo(T t);
```

```
int main()
{
 if constexpr(std::numeric_limits<char>::is_signed) {
   foo(42);          // OK
 }
 else {
   undeclared(42);   // ALWAYS ERROR if not declared (even if discarded)
 }
}
```

Also the following code can never successfully compile, because one of the static assertion will always fail:

```
if constexpr(std::numeric_limits<char>::is_signed) {
  static_assert(std::numeric_limits<char>::is_signed);
}
else {
  static_assert(!std::numeric_limits<char>::is_signed);
}
```

The (only) benefit of the compile-time `if` outside generic code is that code in the discarded statement, although it must be valid, does not become part of the resulting program, which reduces the size of the resulting executable. For example, in this program:

```
#include <limits>
#include <string>
#include <array>

int main()
{
  if (!std::numeric_limits<char>::is_signed) {
    static std::array<std::string,1000> arr1;
    ...
  }
  else {
    static std::array<std::string,1000> arr2;
    ...
  }
}
```

either `arr1` or `arr2` is part of the final executable but not both.[5]

[5] This effect is also possible without `constexpr`, because compilers can optimize code that is not used away. However, with `constexpr` this is guaranteed behavior.

10.5 Afternotes

Compile-time `if` was initially motivated by Walter Bright, Herb Sutter, and Andrei Alexandrescu in https://wg21.link/n3329 and Ville Voutilainen in https://wg21.link/n4461, by proposing a `static if` language feature. In https://wg21.link/p0128r0 Ville Voutilainen proposed the feature for the first time as `constexpr_if` (where the feature got its name from). The finally accepted wording was formulated by Jens Maurer in https://wg21.link/p0292r2.

Chapter 11

Fold Expressions

Since C++17, there is a feature to compute the result of using a binary operator over *all* the arguments of a parameter pack (with an optional initial value).

For example, the following function returns the sum of all passed arguments:

```
template<typename... T>
auto foldSum (T... args) {
  return (... + args);   // ((arg1 + arg2) + arg3) ...
}
```

Note that the parentheses around the return expression are part of the fold expression and can't be omitted.

Calling the function with

```
foldSum(47, 11, val, -1);
```

instantiates the template to perform:

```
return 47 + 11 + val + -1;
```

Calling it for

```
foldSum(std::string("hello"), "world", "!");
```

instantiates the template for:

```
return std::string("hello") + "world" + "!";
```

Also note that the order of fold expression arguments can differ and matters (and might look a bit counter-intuitive): As written,

```
(... + args)
```

results in

```
((arg1 + arg2) + arg3) ...
```

which means that it repeatedly "post-adds" things. You can also write

```
(args + ...)
```

which repeatedly "pre-adds" things, so that the resulting expression is:

```
(arg1 + (arg2 + arg3)) ...
```

11.1 Motivation for Fold Expressions

Fold expression avoid the need to recursively instantiate templates to perform an operation on all parameters of a parameter pack. Before C++17, you had to implement:

```
template<typename T>
auto foldSumRec (T arg) {
  return arg;
}
template<typename T1, typename... Ts>
auto foldSumRec (T1 arg1, Ts... otherArgs) {
  return arg1 + foldSumRec(otherArgs...);
}
```

Such an implementation is not only cumbersome to write, it also stresses C++ compilers. With

```
template<typename... T>
auto foldSum (T... args) {
  return (... + args);   // arg1 + arg2 + arg3 ...
}
```

the effort becomes significantly less for both the programmer and the compiler.

11.2 Using Fold Expressions

Given a parameter *args* and an operator *op*, C++17 allows us to write

- either a ***unary left fold***

  (**...** ***op*** *args* **)**

  which expands to: ((*arg1* ***op*** *arg2*) ***op*** *arg3*) ***op*** ...
- or a ***unary right fold***

  (*args* ***op*** **...** **)**

  which expands to: *arg1* ***op*** (*arg2* ***op*** ... (*argN-1* ***op*** *argN*))

The parentheses are required. However, the parentheses and the ellipsis (...) don't have to be separated by whitespaces.

The difference between left and right fold matters more often than expected. For example, even when using operator + there might be different effects. When using the left fold expression:

```
template<typename... T>
auto foldSumL(T... args){
  return (... + args);   // ((arg1 + arg2) + arg3) ...
```

```
}
```

the call

```
foldSumL(1, 2, 3)
```

evaluates to:

```
(1 + 2) + 3)
```

This also means that the following example compiles:

```
std::cout << foldSumL(std::string("hello"), "world", "!") << '\n'; // OK
```

Remember that operator + is defined for standard strings provided at least one operand is a `std::string`. Because the left fold is used, the call first evaluates

```
std::string("hello") + "world"
```

which returns a `std::string`, so that adding the string literal `"!"` then also is valid.

However, a call such as

```
std::cout << foldSumL("hello", "world", std::string("!")) << '\n'; // ERROR
```

will not compile because it evaluates to

```
("hello" + "world") + std::string("!")
```

and adding two string literals is not allowed.

However, if we change the implementation to:

```
template<typename... T>
auto foldSumR(T... args){
  return (args + ...);    // (arg1 + (arg2 + arg3)) ...
}
```

the call

```
foldSumR(1, 2, 3)
```

evaluates to:

```
(1 + (2 + 3)
```

which means that the following example no longer compiles:

```
std::cout << foldSumR(std::string("hello"), "world", "!") << '\n';  // ERROR
```

while the following call now compiles:

```
std::cout << foldSumR("hello", "world", std::string("!")) << '\n';  // OK
```

Because in almost all cases evaluation from left to right is the intention, usually, the left fold syntax with the parameter pack at the end should be preferred (unless this doesn't work):

```
(... + args);    // preferred syntax for fold expressions
```

11.2.1 Dealing with Empty Parameter Packs

If a fold expression is used with an empty parameter pack, the following rules apply:

- If operator `&&` is used, the value is `true`.
- If operator `||` is used, the value is `false`.
- If the comma operator is used, the value is `void()`.
- For all other operators the call is ill-formed.

For all other cases (and in general) you can add an initial value: Given a parameter pack *args*, an initial value *value* and an operator *op*, C++17 also allows us to write either

- either a ***binary left fold***

  (*value* ***op*** **. . .** ***op*** *args* **)**

  which expands to: (((*value* ***op*** *arg1*) ***op*** *arg2*) ***op*** *arg3*) ***op*** ...
- or a ***binary right fold***

  (*args* ***op*** **. . .** ***op*** *value* **)**

  which expands to: *arg1* ***op*** (*arg2* ***op*** ... (*argN* ***op*** *value*))

The operator *op* has to be the same on both sides of the ellipsis.

For example, the following definition allows to pass an empty parameter pack when adding values:

```
template<typename... T>
auto foldSum (T... s){
  return (0 + ... + s);    // even works if sizeof...(s)==0
}
```

Conceptionally, it shouldn't matter, whether we add `0` as first or last operand:

```
template<typename... T>
auto foldSum (T... s){
  return (s + ... + 0);    // even works if sizeof...(s)==0
}
```

But as for unary fold expression the different evaluation order matters more often than thought and the binary left fold should be preferred:

```
(val + ... + args);    // preferred syntax for binary fold expressions
```

Also, the first operand might be special, such as in this example:

```
template<typename... T>
void print (const T&... args)
{
  (std::cout << ... << args) << '\n';
}
```

Here, it is important that the first call is the output of the first passed argument to `print()`, which returns the stream to perform the other output calls. Other implementations might not compile or even do something unexpected. For example, with

```
std::cout << (args << ... << '\n');
```

a call like `print(1)` will compile but print the value `1` left shifted by the value of `'\0'`, which usually is 10, so that the resulting output is `1024`.

Note that in this `print()` example no whitespace separates all the elements of the parameter pack from each other. A call such as `print("hello", 42, "world")` will print:

```
hello42world
```

To separate the passed elements by spaces, you need a helper that ensures that the output of any but the first argument is extended by a leading space. This can, for example, be done with a helper function template `spaceBefore()`:

tmpl/addspace.hpp

```
template<typename T>
const T& spaceBefore(const T& arg) {
  std::cout << ' ';
  return arg;
}

template <typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
  std::cout << firstarg;
  (std::cout << ... << spaceBefore(args)) << '\n';
}
```

Here,

```
(std::cout << ... << spaceBefore(args))
```

is a fold expression that expands to:

```
std::cout << spaceBefore(arg1) << spaceBefore(arg2) << ...
```

Thus, for each element in the parameter pack `args` it calls a helper function, printing out a space character before returning the passed argument, writing it to `std::cout`. To ensure that this does not apply to the first argument, we add an additional first parameter not using `spaceBefore()`.

Note that the evaluation of the output of the parameter pack requires that all output on the left is done before `spaceBefore()` is called for the actual element. Thanks to the defined evaluation order of operator << and function calls, this is guaranteed to work since C++17.

We can also use a lambda to define `spaceBefore()` inside `print()`:

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
  std::cout << firstarg;
  auto spaceBefore = [](const auto& arg) {
    std::cout << ' ';
    return arg;
  };
```

```
  (std::cout << ... << spaceBefore(args)) << '\n';
}
```

However, note that lambdas by default return objects by value, which means that this would create an unnecessary copy of the passed argument. The way to avoid that is to explicitly declare the return type of the lambda to be `const auto&` or `decltype(auto)`:

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
  std::cout << firstarg;
  auto spaceBefore = [](const auto& arg) -> const auto& {
    std::cout << ' ';
    return arg;
  };
  (std::cout << ... << spaceBefore(args)) << '\n';
}
```

And C++ would not be C++ if you couldn't combine this all in one statement:

```
template<typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
  std::cout << firstarg;
  (std::cout << ... << [](const auto& arg) -> decltype(auto) {
                         std::cout << ' ';
                         return arg;
                       }(args)) << '\n';
}
```

Nevertheless, a simpler way to implement `print()` is to use a lambda that prints both the space and the argument and pass this to a unary fold:[1]

```
template<typename First, typename... Args>
void print(First first, const Args&... args) {
  std::cout << first;
  auto outWithSpace = [](const auto& arg) {
                        std::cout << ' ' << arg;
                      };
  (... , outWithSpace(args));
  std::cout << '\n';
}
```

By using an additional template parameter declared with `auto` we can make `print()` even more flexible to be parameterized for the separator to be a character, a string, or any other printable type.

[1] Thanks to Barry Revzin for pointing that out.

11.2.2 Supported Operators

You can use all binary operators for fold expressions except `.`, `->`, and `[]`.

Folded Function Calls

Fold expression can also be used for the comma operator, combining multiple expressions into one statement. For example, you can fold the comma operator, which enables to perform function calls of member functions of a variadic number of base classes:

tmpl/foldcalls.cpp

```
#include <iostream>

// template for variadic number of base classes
template<typename... Bases>
class MultiBase : private Bases...
{
  public:
    void print() {
      // call print() of all base classes:
      (... , Bases::print());
    }
};

struct A {
  void print() { std::cout << "A::print()\n"; }
};

struct B {
  void print() { std::cout << "B::print()\n"; }
};

struct C {
  void print() { std::cout << "C::print()\n"; }
};

int main()
{
    MultiBase<A,B,C> mb;
    mb.print();
}
```

Here,

```
template<typename... Bases>
```

```
class MultiBase : private Bases...
{
  ...
};
```

allows us to initialize objects with a variadic number of base classes:

```
MultiBase<A,B,C> mb;
```

And with

```
(... , Bases::print());
```

a fold expression is used to expand this to call `print` for each base class. That is, the statement with the fold expression expands to the following:

```
(A::print() , B::print()) , C::print();
```

However, note that due to the nature of the comma operator it doesn't matter whether we use the left or right fold operator. The functions are always called from left to right. With

```
(Bases::print() , ...);
```

the parentheses only group the calls so that the first `print()` call is combined with the result of the other two `print()` calls as follows:

```
A::print() , (B::print() , C::print());
```

But because the evaluation order of the comma operator always is from left to right still the first call happens before the group of two calls inside the parentheses, in which still the middle call happens before the right call.

Nevertheless, as the left fold expression matches with the resulting evaluation order, again the use of left fold expressions is recommended when using them for multiple function calls.

Combining Hash Functions

One example of using the comma operator is to combine hash values. This can be done as follows:

```
template<typename T>
void hashCombine (std::size_t& seed, const T& val)
{
  seed ^= std::hash<T>()(val) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}

template<typename... Types>
std::size_t combinedHashValue (const Types&... args)
{
  std::size_t seed = 0;              // initial seed
  (... , hashCombine(seed,args));    // chain of hashCombine() calls
  return seed;
}
```

By calling

```
std::size_t combinedHashValue ("Hello", "World", 42,);
```

the statement in the middle expands to:

```
hashCombine(seed,"Hello"), (hashCombine(seed,"World"), hashCombine(seed,42);
```

With this definition we can easily define a new hash function object for a type such as `Customer`:

```
struct CustomerHash
{
  std::size_t operator() (const Customer& c) const {
    return combinedHashValue(c.getFirstname(), c.getLastname(), c.getValue());
  }
};
```

which we can use to put `Customers` in an unordered set:

```
std::unordered_set<Customer, CustomerHash> coll;
```

Folded Path Traversals

You can also use a fold expression to traverse a path in a binary tree with operator `->*`:

tmpl/foldtraverse.cpp

```
// define binary tree structure and traverse helpers:
struct Node {
  int value;
  Node* left;
  Node* right;
  Node(int i=0) : value(i), left(nullptr), right(nullptr) {
  }
  ...
};
auto left = &Node::left;
auto right = &Node::right;

// traverse tree, using fold expression:
template<typename T, typename... TP>
Node* traverse (T np, TP... paths) {
  return (np ->* ... ->* paths);      //np ->* paths1 ->* paths2 ...
}

int main()
{
  // init binary tree structure:
  Node* root = new Node{0};
```

```
  root->left = new Node{1};
  root->left->right = new Node{2};
  ...
  // traverse binary tree:
  Node* node = traverse(root, left, right);
  ...
}
```

Here,

```
(np ->* ... ->* paths)
```

uses a fold expression to traverse the variadic elements of `paths` from `np`. When calling

```
traverse(root, left, right);
```

the call of the fold expression expands to:

```
root -> left -> right
```

11.2.3 Using Fold Expressions for Types

By using type traits we can also use fold expressions to deal with template parameter packs (an arbitrary number of types passed as template parameters). For example, you can use a fold expression to find out whether a list of types is homogeneous:

tmpl/ishomogeneous.hpp

```
#include <type_traits>

// check whether passed types are homogeneous:
template<typename T1, typename... TN>
struct IsHomogeneous {
  static constexpr bool value = (std::is_same<T1,TN>::value && ...);
};

// check whether passed arguments have the same type:
template<typename T1, typename... TN>
constexpr bool isHomogeneous(T1, TN...)
{
  return (std::is_same<T1,TN>::value && ...);
}
```

The type trait `IsHomogeneous<>` can be used, for example, as follows:

```
IsHomogeneous<int, Size, decltype(42)>::value
```

In this case the fold expression that initializes the member `value` expands to:

```
std::is_same<int,MyType>::value && std::is_same<int,decltype(42)>::value
```

The function template `isHomogeneous<>()` can be used, for example, as follows:

```
isHomogeneous(43, -1, "hello", nullptr)
```

In this case the fold expression that initializes the member `value` expands to:

```
std::is_same<int,int>::value && std::is_same<int,const char*>::value
  && std::is_same<int,std::nullptr_t>::value
```

As usual, operator `&&` short-circuits (aborts the evaluation after the first `false`).

The deduction guide for `std::array<>` uses this feature in the standard library.

11.3 Afternotes

Fold expressions were first proposed by Andrew Sutton and Richard Smith in `https://wg21.link/n4191`. The finally accepted wording was formulated by Andrew Sutton and Richard Smith in `https://wg21.link/n4295`. Support for empty sequences was later removed for operators `*`, `+`, `&`, and `|` as proposed by Thibaut Le Jehan in `https://wg21.link/p0036`.

This page is intentionally left blank

Chapter 12

Dealing with Strings as Template Parameters

Over time the different versions of C++ relaxed the rules for what can be used as templates parameters, and with C++17 this happened again. Templates now can be used without the need to have them defined outside the current scope.

12.1 Using Strings in Templates

Non-type template parameters can be only constant integral values (including enumerations), pointers to objects/functions/members, lvalue references to objects or functions, or `std::nullptr_t` (the type of `nullptr`).

For pointers, linkage is required, which means that you can't pass string literals directly. However, since C++17, you can have pointers with internal linkage. For example:

```
template<const char* str>
class Message {
  ...
};

extern const char hello[] = "Hello World!";       // external linkage
const char hello11[] = "Hello World!";            // internal linkage

void foo()
{
  Message<hello>   msg;          // OK (all C++ versions)
  Message<hello11> msg11;        // OK since C++11

  static const char hello17[] = "Hello World!";  // no linkage
```

```
  Message<hello17> msg17;          // OK since C++17
}
```

That is, since C++17, you still need two lines to pass a string literal to a template. But you can have the first line in the same scope as the class instantiation.

This ability also solves an unfortunate constraint: While you could pass a pointer to a class template since C++11:

```
template<int* p> struct A {
};

int num;
A<&num> a;       // OK since C++11
```

You couldn't use a compile-time function that returned the address, which now is supported:

```
int num;
...
constexpr int* pNum() {
  return &num;
}
A<pNum()> b;     // ERROR before C++17, now OK
```

12.2 Afternotes

Allowing constant evaluation for all non-type template arguments was first proposed by Richard Smith in https://wg21.link/n4198. The finally accepted wording was formulated by Richard Smith in https://wg21.link/n4268.

Chapter 13

Placeholder Types like `auto` as Template Parameters

Since C++17 you can use placeholder types (`auto` and `decltype(auto)`) as non-type template parameter types. That means, that we can write generic code for non-type parameters of different types.

13.1 Using `auto` as Template Parameter

Since C++17, you can use `auto` to declare a non-type template parameter. For example:

```
template<auto N> class S {
  ...
};
```

This allows us to instantiate the non-type template parameter `N` for different types:

```
S<42>  s1;  // OK: type of N in S is int
S<'a'> s2;  // OK: type of N in S is char
```

However, you can't use this feature to get instantiations for types that in general are not allowed as template parameters:

```
S<2.5> s3;  // ERROR: template parameter type still cannot be double
```

We can even have a specific type as partial specialization:

```
template<int N> class S<N> {
  ...
};
```

Even class template argument deduction is supported. For example:

```
template<typename T, auto N>
class A {
```

```
  public:
    A(const std::array<T,N>&) {
    }
    A(T(&)[N]) {
    }
    ...
};
```

This class can deduce the type of `T`, the type of `N`, and the value of `N`:

```
A a2{"hello"};  // OK, deduces A<const char, 6> with N being int

std::array<double,10> sa1;
A a1{sa1};      // OK, deduces A<double, 10> with N being std::size_t
```

You can also qualify `auto`, for example, to require the type of the template parameter to be a pointer:

```
template<const auto* P> struct S;
```

And by using variadic templates, you can parameterize templates to use a list of heterogeneous constant template arguments:

```
template<auto... VS> class HeteroValueList {
};
```

or a list of homogeneous constant template arguments:

```
template<auto V1, decltype(V1)... VS> class HomoValueList {
};
```

For example:

```
HeteroValueList<1, 2, 3> vals1;       // OK
HeteroValueList<1, 'a', true> vals2;  // OK
HomoValueList<1, 2, 3> vals3;         // OK
HomoValueList<1, 'a', true> vals4;    // ERROR
```

13.1.1 Parameterizing Templates for Characters and Strings

One application of this feature is to allow passing both a character or a string as template parameter. For example, we can improve the way we output an arbitrary number of arguments with fold expressions as follows:

tmpl/printauto.hpp

```
#include <iostream>

template<auto Sep = ' ', typename First, typename... Args>
void print(const First& first, const Args&... args) {
  std::cout << first;
```

```
  auto outWithSpace = [](const auto& arg) {
                        std::cout << Sep << arg;
                      };
  (... , outWithSpace(args));
  std::cout << '\n';
}
```

Still, we can print the arguments with a space being the default argument for the template parameter Sep:

```
template<auto Sep = ' ', typename First, typename... Args>
void print (const First& firstarg, const Args&... args) {
  ...
}
```

That is, we still can call:

```
std::string s{"world"};
print(7.5, "hello", s);          // prints: 7.5 hello world
```

But by having print() parameterized for the separator Sep, we can now explicitly pass a different character as first template argument:

```
print<'-'>(7.5, "hello", s);     // prints: 7.5-hello-world
```

And due to the use of auto, we can even pass a string literal, which we have to declare as an object having no linkage, though:

```
static const char sep[] = ", ";
print<sep>(7.5, "hello", s);     // prints: 7.5, hello, world
```

Or we can pass a separator of any other type usable as template parameter (which can make more sense than here):

```
print<-11>(7.5, "hello", s);     // prints: 7.5-11hello-11world
```

13.1.2 Defining Metaprogramming Constants

Another application of the auto feature for template parameters is to define compile-time constants more easily.[1] Instead of defining:

```
template<typename T, T v>
struct constant
{
  static constexpr T value = v;
};

using i = constant<int, 42>;
```

[1] Thanks to Bryce Adelstein Lelbach for providing these examples.

```
using c = constant<char, 'x'>;
using b = constant<bool, true>;
```

You can now just do the following:

```
template<auto v>
struct constant
{
  static constexpr auto value = v;
};

using i = constant<42>;
using c = constant<'x'>;
using b = constant<true>;
```

And instead of:

```
template<typename T, T... Elements>
struct sequence {
};

using indexes = sequence<int, 0, 3, 4>;
```

you can now just implement:

```
template<auto... Elements>
struct sequence {
};

using indexes = sequence<0, 3, 4>;
```

You can now even define compile-time objects representing a heterogeneous list of values (something like a condensed tuple):

```
using tuple  = sequence<0, 'h', true>;
```

13.2 Using `auto` as Variable Template Parameter

You can also use `auto` as template parameters with *variable templates*.[2] For example, the following declaration, which might occur in a header file, defines a variable template `arr` parameterized for the type of elements and both the tape and value of the number of elements:

```
template<typename T, auto N> std::array<T,N> arr;
```

In each translation unit all usages of `arr<int,10>` share the same global object, while `arr<long,10>` and `arr<int,10u>` would be different global objects (again both usable in all translation units).

[2] Don't confuse *variable templates*, which are templified variables, with *variadic templates*, which are templates that have an arbitrary number of parameters.

As a full example, consider the following header file:

tmpl/vartmplauto.hpp

```
#ifndef VARTMPLAUTO_HPP
#define VARTMPLAUTO_HPP

#include <array>

template<typename T, auto N> std::array<T,N> arr{};

void printArr();

#endif // VARTMPLAUTO_HPP
```

Here, one translation unit could modify the values of two different instances of this variable template

tmpl/vartmplauto1.cpp

```
#include "vartmplauto.hpp"

int main()
{
  arr<int,5>[0] = 17;
  arr<int,5>[3] = 42;
  arr<int,5u>[1] = 11;
  arr<int,5u>[3] = 33;
  printArr();
}
```

And another translation unit could print these two variables:

tmpl/vartmplauto2.cpp

```
#include "vartmplauto.hpp"
#include <iostream>

void printArr()
{
  std::cout << "arr<int,5>:  ";
  for (const auto& elem : arr<int,5>) {
    std::cout << elem << ' ';
  }
  std::cout << "\narr<int,5u>: ";
  for (const auto& elem : arr<int,5u>) {
```

```
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

The output of the program would be:[3]

```
arr<int,5>:  17 0 0 42 0
arr<int,5u>: 0 11 0 33 0
```

The same way you can declare a constant variable of an arbitrary type deduced from its initial value:

```
template<auto N> constexpr auto val = N;       // OK since C++17
```

and use it later, for example, as follows:

```
auto v1 = val<5>;                              // v1 == 5, v1 is int
auto v2 = val<true>;                           // v2 == true, v2 is bool
auto v3 = val<'a'>;                            // v3 == 'a', v3 is char
```

To clarify what is happening here:

```
std::is_same_v<decltype(val<5>), int>          // yields false
std::is_same_v<decltype(val<5>), const int>    // yields true
std::is_same_v<decltype(v1), int>;             // yields true (because auto decays)
```

13.3 Using `decltype(auto)` as Template Parameter

You can also use the other placeholder type, `decltype(auto)`, introduced with C++14. Note however, that this type has very special rules how the type is deduced. According to `decltype`, if *expressions* instead of names are passed, it deduces the type according to the value category of the expression:

- *type* for a prvalue (e.g., temporaries)
- *type*`&` for an lvalue (e.g., objects with names)
- *type*`&&` for an xvalue (e.g., objects casted to rvalue-references, as with `std::move()`.

That means, you can easily deduce template parameters to become references, which might result in surprising effects.

For example:

tmpl/decltypeauto.cpp

```
#include <iostream>

template<decltype(auto) N>
```

[3] There is a bug in g++ 7, so that these are handles as one object. This bug is fixed with g++ 8.

```
struct S {
  void printN() const {
    std::cout << "N: " << N << '\n';
  }
};

static const int c = 42;
static int v = 42;

int main()
{
  S<c> s1;       // deduces N as const int 42
  S<(c)> s2;     // deduces N as const int& referring to c
  s1.printN();
  s2.printN();

  S<(v)> s3;     // deduces N as int& referring to v
  v = 77;
  s3.printN();   // prints: N: 77
}
```

13.4 Afternotes

Placeholder types for non-type template parameters were first proposed by James Touton and Michael Spertus as part of https://wg21.link/n4469. The finally accepted wording was formulated by James Touton and Michael Spertus in https://wg21.link/p0127r2.

This page is intentionally left blank

Chapter 14

Extended Using Declarations

Using declarations were extended to allow a comma separated list of declarations, to allow them to be used in a pack expansion.

For example, you can program now:

```
class Base {
  public:
    void a();
    void b();
    void c();
};

class Derived : private Base {
  public:
    using Base::a, Base::b, Base::c;
};
```

Before C++17, you needed three different using declarations, instead.

14.1 Using Variadic Using Declarations

Comma separated using declarations provide the ability us to generically derive all operations of the same kind from a variadic list of base classes.

A pretty cool application of this technique is to create a set of lambda overloads. By defining the following:

tmpl/overload.hpp

```
// "inherit" all function call operators of passed base types:
template<typename... Ts>
struct overload : Ts...
```

```
{
  using Ts::operator()...;
};

// base types are deduced from passed arguments:
template<typename... Ts>
overload(Ts...) -> overload<Ts...>;
```

you can overload two lambdas as follows:

```
auto twice = overload {
                [](std::string& s) { s += s; },
                [](auto& v) { v *= 2; }
             };
```

Here we create an object of type `overload`, where we use a deduction guide to deduce the types of the lambdas as base classes of the template type `overload` and use aggregate initialization to initialize the subobjects for the bases classes with the copy constructor of the closure type, each lambda has.

The using declaration then makes both function call operators available for type `overload`. Without the using declaration, the base classes would have two different overloads of the same member function `operator()`, which is ambiguous.[1]

As a result, you can pass a string, which calls the first overload or pass another type, which (provided operator `*=` is valid) uses the second overload:

```
int i = 42;
twice(i);
std::cout << "i: " << i << '\n';   // prints: 84
std::string s = "hi";
twice(s);
std::cout << "s: " << s << '\n';   // prints: hihi
```

An application of this technique are `std::variant` visitors.

14.2 Variadic Using Declarations for Inheriting Constructors

Together with some clarifications on inheriting constructors, the following also is possible now: You can declare a variadic class template `Multi` that derives from a base class for each of its passed types:

tmpl/using2.hpp

[1] Both clang and Visual C++ don't handle the overloading of operators of base classes for different types as ambiguity, so that there the `using` is not necessary. However, according to language rules, this is necessary as for overloaded member functions, where both compilers require it, and should be used to be portable.

```
template<typename T>
class Base {
    T value{};
  public:
    Base() {
      ...
    }
    Base(T v) : value{v} {
      ...
    }
    ...
};

template<typename... Types>
class Multi : private Base<Types>...
{
  public:
    // derive all constructors:
    using Base<Types>::Base...;
    ...
};
```

With the using declaration for all base class constructors, you derive for each type a corresponding constructor.

Now, when declaring `Multi<>` type for values of three different types:

```
using MultiISB = Multi<int,std::string,bool>;
```

you can declare objects using each one of the corresponding constructors:

```
MultiISB m1 = 42;
MultiISB m2 = std::string("hello");
MultiISB m3 = true;
```

By the new language rules, each initialization calls the corresponding constructor for the matching base class and the default constructor for all other base classes. Thus

```
MultiISB m2 = std::string("hello");
```

calls the default constructor for `Base<int>`, the string constructor for `Base<std::string>`, and the default constructor for `Base<bool>`.

In principle, you could also enable all assignment operators in `Multi<>` by specifying:

```
template<typename... Types>
class Multi : private Base<Types>...
{
  ...
  // derive all assignment operators:
```

```
    using Base<Types>::operator=...;
};
```

14.3 Afternotes

Comma-separated using declarations were proposed by Robert Haberlach in https://wg21.link/p0195r0. The finally accepted wording was formulated by Robert Haberlach and Richard Smith in https://wg21.link/p0195r2.

Various core issues requested clarifications on inheriting constructors. The finally accepted wording to fix them was formulated by Richard Smith in https://wg21.link/n4429.

There is a proposal by Vicente J. Botet Escriba to add a generic `overload` function to overload lambdas but also ordinary functions and member functions. However, the paper didn't make it into C++17. See https://wg21.link/p0051r1 for details.

Part III

New Library Components

This part introduces the new library components of C++17.

This page is intentionally left blank

Chapter 15

`std::optional<>`

In programming often we have the case that we *might* return/pass/use an object of a certain type. That is, we could have a value of a certain type or we might not have any value at all. Thus, we need a way to simulate semantics similar to pointers, where we can express to have *no value* by using `nullptr`. The way to handle this is to define an object of a certain type with an additional Boolean member/flag signaling whether a value exists. `std::optional<>` provides such objects in a type-safe way.

Optional objects simply have internal memory for the *contained* objects plus a Boolean flag. Thus, the size usually is one byte larger than the contained object. For some contained types, there might even be no size overhead at all, provided the additional information can be placed inside the contained object. No heap memory is allocated. The objects use the same alignment as the contained type.

However, optional objects are not just structures adding the functionality of a Boolean flag to a value member. For example, if there is no value, no constructor is called for the contained type (thus, you can give objects a default state that don't have one).

As with `std::variant<>` and `std::any` the resulting objects have value semantics. That is, copying is implemented as a *deep copy* creating an independent object with the flag and the contained value if any in its own memory. Copying a `std::optional<>` without a contained value is cheap; Copying a `std::optional<>` with a contained value is as cheap/expensive as copying the contained type/value. Move semantics are supported.

15.1 Using `std::optional<>`

`std::optional<>` model a nullable instance of an arbitrary type. The instance might be a member, an argument, or a return value. You could also argue that a `std::optional<>` is a container for zero or one element.

15.1.1 Optional Return Values

The following program demonstrates the abilities of std::optional<> to be used as return values:

lib/optional.cpp

```
#include <optional>
#include <string>
#include <iostream>

// convert string to int if possible:
std::optional<int> asInt(const std::string& s)
{
  try {
    return std::stoi(s);
  }
  catch (...) {
    return std::nullopt;
  }
}

int main()
{
  for (auto s : {"42", "  077", "hello", "0x33"} ) {
    // convert s to int and use the result if possible:
    std::optional<int> oi = asInt(s);
    if (oi) {
      std::cout << "convert '" << s << "' to int: " << *oi << "\n";
    }
    else {
      std::cout << "can't convert '" << s << "' to int\n";
    }
  }
}
```

In the program asInt() is a function to convert a passed string to an integer. However, this might not succeed. For this reason a std::optional<> is used so that we can return "*no int*" and avoid to define a special int value for it or throw an exception to the caller.

Thus, we either return the result of calling stoi(), which initializes the return value with an int, or we return std::nullopt, signaling that we don't have an int value. We could implement the same behavior as follows:

```
std::optional<int> asInt(const std::string& s)
{
  std::optional<int> ret;   // initially no value
```

```
  try {
    ret = std::stoi(s);
  }
  catch (...) {
  }
  return ret;
}
```

In `main()` we call this function for different strings.

```
for (auto s : {"42", "  077", "hello", "0x33"} ) {
  // convert s to int and use the result if possible:
  std::optional<int> oi = asInt(s);
  ...
}
```

For each returned `std::optional<int> oi` we evaluate, whether we have a value (by evaluating the object as Boolean expression) and access the value by "dereferencing" the optional object:

```
if (oi) {
  std::cout << "convert '" << s << "' to int: " << *oi << "\n";
}
```

Note that for the string `"0x33"` `asInt()` yields `0` because `stoi()` does not parse the string as hexadecimal value.

There are alternative ways to implement the handling of the return value, such as:

```
std::optional<int> oi = asInt(s);
if (oi.has_value()) {
  std::cout << "convert '" << s << "' to int: " << oi.value() << "\n";
}
```

Here, `has_value()` is used to check, whether a value was returns, and with `value()` we access it. `value()` is safer than operator `*`: It throws an exception if no value exists. Operator `*` should only be used when you are sure that the optional contains a value; otherwise your program will have undefined behavior.[1]

Note that we can improve `asInt()` by using the new type `std::string_view`.

15.1.2 Optional Arguments and Data Members

Another example using `std::optional<>` is the optional passing of arguments and/or the optional setting of a data member:

lib/optionalmember.cpp

```
#include <string>
```

[1] Note that you might not see this undefined behavior, because operator `*` yields the value at the memory, which might (still) make sense.

```
#include <optional>
#include <iostream>

class Name
{
  private:
    std::string first;
    std::optional<std::string> middle;
    std::string last;
  public:
    Name (std::string f,
          std::optional<std::string> m,
          std::string l)
     : first{std::move(f)}, middle{std::move(m)}, last{std::move(l)} {
    }
    friend std::ostream& operator << (std::ostream& strm, const Name& n) {
      strm << n.first << ' ';
      if (n.middle) {
        strm << *n.middle << ' ';

      }
      return strm << n.last;
    }
};

int main()
{
    Name n{"Jim", std::nullopt, "Knopf"};
    std::cout << n << '\n';

    Name m{"Donald", "Ervin", "Knuth"};
    std::cout << m << '\n';
}
```

Class `Name` represents a name that consists out of a first name, an optional middle name, and a last name. The member `middle` is defined accordingly and the constructor enables to pass `std::nullopt` when there is no middle name. This is a different state than the middle name being the empty string.

Note that as usual for types with value semantics, the best way to define a constructor initializing the corresponding members is to take the arguments by value and move the parameters to the member:

Note also that `std::optional<>` changes the access to the value of the member `middle`. Using `middle` as a Boolean expression yields whether there is a middle name, which `*middle` has to be used to access the current value (if any).

Another option to access the value is by using the member function value_or(), which enables to specify a fallback value in case no value exists. For example, inside class Name we could also implement:

```
std::cout << middle.value_or("");   // print middle name or nothing
```

15.2 std::optional<> Types and Operations

This section describes the types and operations of std::optional<> in detail.

15.2.1 std::optional<> Types

In the header file <optional> the C++ standard library defines class std::optional<> as follows:

```
namespace std {
  template<typename T> class optional;
}
```

In addition, the following types and objects are defined:

- nullopt of type std::nullopt_t as a "value" for optional objects having no value.
- Exception class std::bad_optional_access, which is derived from std::exception, for value access without having a value.

Optional objects also use the object std::in_place (of type std::in_place_t) defined in <utility> to initialize the value of an optional object with multiple arguments (see below).

15.2.2 std::optional<> Operations

Table std::optional *Operations* lists all operations that are provided for std::optional<>.

Construction

Special constructors enable to pass the arguments directly to the contained type.

- You can create an optional object not having a value. In that case you have to specify the contained type:

  ```
  std::optional<int> o1;
  std::optional<int> o2(std::nullopt);
  ```

  This does not call any constructor for the contained type.
- You can pass a value to initialize the contained type. Due to a deduction guide you don't have to specify the contained type, then:

  ```
  std::optional o3{42};        // deduces optional<int>
  std::optional<std::string> o4{"hello"};
  std::optional o5{"hello"};   // deduces optional<const char*>
  ```

Operation	Effect
constructors	Create an optional object (might call constructor for contained type)
make_optional<>()	Create an optional object (passing value(s) to initialize it)
destructor	Destroys an optional object
=	Assign a new value
emplace()	Assign a new value to the contained type
reset()	Destroys any value (makes the object empty)
has_value()	Returns whether the object has a value
conversion to bool	Returns whether the object has a value
*	Value access (undefined behavior if no value)
->	Access to member of the value (undefined behavior if no value)
value()	Value access (exception if no value)
value_or()	Value access (fallback argument if no value)
swap()	Swaps values between two objects
==, !=, <, <=, >, >=	Compare optional objects
hash<>	Function object type to compute hash values

Table 15.1. std::optional<> *Operations*

- To initialize an optional object with multiple arguments, you have to create the object or add std::in_place as first argument (the contained type can't be deduced):

```
std::optional o6{std::complex{3.0, 4.0}};
std::optional<std::complex<double>> o7{std::in_place, 3.0, 4.0};
```

  Note that the latter form avoids the creation of a temporary object. By using this form, you can even pass an initializer list plus additional arguments:

```
// initialize set with lambda as sorting criterion:
auto sc = [] (int x, int y) {
            return std::abs(x) < std::abs(y);
          };
std::optional<std::set<int,decltype(sc)>> o8{std::in_place,
                                             {4, 8, -7, -2, 0, 5},
                                             sc};
```

- You can copy optional objects (including type conversions).

```
std::optional o5{"hello"};  // deduces optional<const char*>
std::optional<std::string> o9{o5};  // OK
```

Note that there is also a convenience function make_optional<>(), which allows an initialization with single or multiple arguments (without the need for the in_place argument). As usual for make... functions it decays:

```
auto o10 = std::make_optional(3.0);       // optional<double>
```

```
auto o11 = std::make_optional("hello");  // optional<const char*>
auto o12 = std::make_optional<std::complex<double>>(3.0, 4.0);
```

However, note that there is no constructor taking a value and deciding according its value, whether to initialize an optional with a value or `nullopt`. For this, operator `?:` has to be used.[2] For example:

```
std::multimap<std::string, std::string> englishToGerman;
...
auto pos = englishToGerman.find("wisdom");
auto o13 = pos != englishToGerman.end()
            ? std::optional{pos->second}
            : std::nullopt;
```

Here, `o13` is initialized as `std::optional<std::string>` due to class template argument deduction for `std::optional{pos->second}`. For `std::nullopt` class template argument deduction would not work, but operator `?:` converts that also to this type, when deducing the resulting type of the expression.

Accessing the Value

To check, whether an optional object has a value you can use it in a Boolean expression or call `has_value()`:

```
std::optional o{42};

if (o) ...                // true
if (!o) ...               // false
if (o.has_value()) ...    // true
```

To access the value then, a pointer syntax is provided. That is with `operator*` you can directly access its value while `operator->` enables to access members of the value:

```
std::optional o{std::pair{42, "hello"}};

auto p = *o;              // initializes p as pair<int,string>
std::cout << o->first;    // prints 42
```

Note that these operators require that the optional contains a value. Using them without having a value is undefined behavior:

```
std::optional<std::string> o{"hello"};

std::cout << *o;          // OK: prints "hello"
o = std::nullopt;
std::cout << *o;          // undefined behavior
```

Note that in practice the second output will still compile and perform some output such as printing `"hello"` again, because the underlying memory for the value of the optional object was not modified.

[2] Thanks to Roland Bock for pointing that out.

However, you can't and should never rely on that. If you don't know whether an optional object has a value, you have to call the following instead:

```
if (o) std::cout << *o;   // OK (might output nothing)
```

Alternatively, you can use `value()`, which throws a `std::bad_optional_access` exception, if there is no contained value:

```
std::cout << o.value();   // OK (throws if no value)
```

`std::bad_optional_access` is directly derived from `std::exception`.

Finally, you can ask for the value and pass a fallback value, which is used, if the optional object has no value:

```
std::cout << o.value_or("fallback");   // OK (outputs fallback if no value)
```

The fallback argument is passed as rvalue reference so that it costs nothing if the fallback isn't used and it supports move semantics if it is used.

Please note that both `operator*` and `value()` return the contained object by reference. For this reason, you have to be careful, when calling these operation directly for temporary return values. For example:

```
std::optional<std::string> getString();
...
auto a = getString().value();            // OK: copy of contained object
auto b = *getString();                   // ERROR: undefined behavior if std::nullopt
const auto& r1 = getString().value();    // ERROR: reference to deleted contained ob-
ject
auto&& r2 = getString().value();         // ERROR: reference to deleted contained ob-
ject
```

An example might be the following usage of a range-based `for` loop:

```
std::optional<std::vector<int>> getVector();
...
for (int i : getVector().value()) {      // ERROR: iterate over deleted vector
  std::cout << i << '\n';
}
```

Note that iterating over a returned vector of `int` would work. So, do not blindly replace the return type of a function `foo()` by the corresponding optional type, calling `foo().value()` instead.

Comparisons

You can use the usual comparison operators. Operands can be an optional object, an object of the contained type, and `std::nullopt`.

- If both operands are objects with a value, the corresponding operator of the contained type is used.
- If both operands are objects without a value they are considered to be equal (`==` yields `true` and all other comparisons yield `false`).

- If only one operand is an object with a value the operand without a value is considered to be less than the other operand.

For example:

```
std::optional<int> o0;
std::optional<int> o1{42};

  o0 == std::nullopt  // yields true
  o0 == 42            // yields false
  o0 < 42             // yields true
  o0 > 42             // yields false
  o1 == 42            // yields true
  o0 < o1             // yields true
```

This means that for optional objects of `unsigned int` there is a value less than `0` and for optional objects of `bool` there is a value less than `0`:

```
std::optional<unsigned> uo;
  uo < 0              // yields true
std::optional<bool> bo;
  bo < false          // yields true
```

Again, implicit type conversions for the underlying type are supported:

```
std::optional<int> o1{42};
std::optional<double> o2{42.0};

o2 == 42          // yields true
o1 == o2          // yields true
```

Note that optional Boolean or raw pointer values can result in some surprises here.

Changing the Value

Assignment and `emplace()` operations exist corresponding to the initializations:

```
std::optional<std::complex<double>> o;  // has no value
std::optional ox{77}; // optional<int> with value 77

o = 42;                 // value becomes complex(42.0, 0.0)
o = {9.9, 4.4};         // value becomes complex(9.9, 4.4)
o = ox;                 // OK, because int converts to complex<double>
o = std::nullopt;       // o no longer has a value
o.emplace(5.5, 7.7);    // value becomes complex(5.5, 7.7)
```

Assigning `std::nullopt` removes the value, which calls the destructor of the contained type if there was a value before. You can have the same effect by calling `reset()`:

```
o.reset();              // o no longer has a value
```

or assigning empty curly braces:

```
o = {};                  // o no longer has a value
```

Finally, we can also use `operator*` to modify the value, because it yields the value by reference. However, note that this requires that there is a value to modify:

```
std::optional<std::complex<double>> o;
*o = 42;                 // undefined behavior
...
if (o) {
  *o = 88;               // OK: value becomes complex(88.0, 0.0)
  *o = {1.2, 3.4};       // OK: value becomes complex(1.2, 3.4)
}
```

Move Semantics

`std::optional<>` also supports move semantics. If you move the object as a whole, the state gets copied and the contained object (if any) is moved. As a result, a moved-from object still has the same state, but any value becomes unspecified.

But you can also move a value into or out of the contained object. For example:

```
std::optional<std::string> os;
std::string s = "a very very very long string";
os = std::move(s);                  // OK, moves
std::string s2 = *os;               // OK copies
std::string s3 = std::move(*os);    // OK, moves
```

Note that after the last call `os` still has a string value, but as usual for moved-from objects the value is unspecified. Thus, you can use it as long as you don't make any assumption about which value it is. You can even assign a new string value there.

Hashing

The hash value for an optional object is the hash value of the contained non-constant type (if any).

15.3 Special Cases

Specific optional value types can result in special or unexpected behavior.

15.3.1 Optional of Boolean or Raw Pointer Values

Note that using the comparison operator has different semantics than using an optional object as a Boolean value. This can become confusing if the contained type is `bool` or a pointer type: For example:

```
std::optional<bool> ob{false};   // has value, which is false
if (!ob) ...                     // yields false
```

```
if (ob == false) ...      // yields true

std::optional<int*> op{nullptr};
if (!op) ...              // yields false
if (op == nullptr) ...    // yields true
```

15.3.2 Optional of Optional

In principle you can also define an optional of an optional value:

```
std::optional<std::optional<std::string>> oos1;
std::optional<std::optional<std::string>> oos2 = "hello";
std::optional<std::optional<std::string>>
  oos3{std::in_place, std::in_place, "hello"};

std::optional<std::optional<std::complex<double>>>
  ooc{std::in_place, std::in_place, 4.2, 5.3};
```

Also you can assign new values even with implicit conversions:

```
oos1 = "hello";          // OK: assign new value
ooc.emplace(std::in_place, 7.2, 8.3);
```

Due to the two levels of having no value, an optional of optional enables to have "no value" on the outside or on the inside, which can have different semantic meaning:

```
*oos1 = std::nullopt;   // inner optional has no value
oos1 = std::nullopt;    // outer optional has no value
```

But you have to take special care to deal with the optional value:

```
if (!oos1) std::cout << "no value\n";
if (oos1 && !*oos1) std::cout << "no inner value\n";
if (oos1 && *oos1) std::cout << "value: " << **oos1 << '\n';
```

However, because this is semantically more a value with two different states representing to have no value, a `std::variant<>` with two Boolean or `monostate` alternatives might be more appropriate.

15.4 Afternotes

Optional objects were first proposed 2005 by Fernando Cacciola in https://wg21.link/n1878 referring to Boost.Optional as a reference implementation. This class was adopted to become part of the Library Fundamentals TS as proposed by Fernando Cacciola and Andrzej Krzemienski in https://wg21.link/n3793.

The class was adopted with other components for C++17 as proposed by Beman Dawes and Alisdair Meredith in https://wg21.link/p0220r1.

Tony van Eerd significantly improved the semantics for comparison operators with https://wg21.link/n3765 and https://wg21.link/p0307r2. Vicente J. Botet Escriba harmonized the

API with std::variant<> and std::any with https://wg21.link/p0032r3. Jonathan Wakely fixed the behavior for in_place tag types with https://wg21.link/p0504r0.

Chapter 16

`std::variant<>`

Adopted from C, C++ provides support for `unions`, which are objects able to hold *one* of a list of possible types. However, there are some drawbacks with this language feature:

- Objects do not know, which type of value they currently hold.
- For this reason you can't have non-trivial members, such as `std::string` (without specific effort).[1]
- You can't derive from a `union`.

With `std::variant<>` the C++ standard library provides a *closed discriminated union* (which means that there is a specified list of possible types and you can specify which type you mean), where

- the type of the current value is always known,
- that can have members of any specified type, and
- you can derive from.

In fact, a `std::variant<>` holds a value of various *alternatives*, which usually have different types. But two alternatives also can have the same type, which is useful if alternatives with different semantic meaning have the same type (e.g., holding two strings, which represent different database columns, so that you still know, which of the columns the value represents).

Variants simply have internal memory for the maximum size of the underlying types plus some fixed overhead to manage which alternative is used. No heap memory is allocated.[2]

In general, variants can't be empty unless you use a specific alternative to signal emptiness. However, in very rare cases (such as due to exceptions during the assignment of a new value of a different type) the variant can come into a state having no value at all.

---

[1] Since C++11, `unions` in principle can have non-trivial members, but you have to implement special member functions such as the copy-constructor and destructor then, because only by programming logic you know which member is active.

[2] This is different from Boost.Variant, where memory had to be allocated to be able to recover from exceptions during value changes.

As with std::optional<> and std::any the resulting objects have value semantics. Copying happens deeply by creating an independent object with the current value of the current alternative in its own memory. Therefore, copying a std::variant<> is as cheap/expensive as copying the type/value of the current alternative. Move semantics is supported.

16.1 Using std::variant<>

The following example demonstrates the core abilities of std::variant<>:

lib/variant.cpp

```
#include <variant>
#include <iostream>

int main()
{
  std::variant<int, std::string> var{"hi"};  // initialized with string alternative
  std::cout << var.index() << '\n';          // prints 1
  var = 42;                                  // now holds int alternative
  std::cout << var.index() << '\n';          // prints 0
  ...
  try {
    int i = std::get<0>(var);                      // access by index
    std::string s = std::get<std::string>(var);    // access by type (throws exception in this case)
    ...
  }
  catch (const std::bad_variant_access& e) {       // in case a wrong type/index is used
    std::cerr << "EXCEPTION: " << e.what() << '\n';
    ...
  }
}
```

The member function index() can be used to find out, which alternative is currently set (the first alternative has the index 0).

Initializations and assignment always use the best match to find out the new alternative. If the type doesn't fit exactly, there might be surprises.

Note that empty variants, variants with reference members, variants with C-style array members, and variants with incomplete types (such as void) are not allowed.[3]

There is no empty state. That means that for each constructed object at least one constructor has to be called. The default constructor initializes the first type with the default constructor:

[3] These features might be added later, but for C++17 there was not enough experience to support it.

```
std::variant<std::string, int> var;          // => var.index() == 0, value == ""
```

If there is no default constructor defined for the first type, calling the default constructor for the variant is a compile-time error:

```
struct NoDefConstr {
  NoDefConstr(int i) {
    std::cout << "NoDefConstr::NoDefConstr(int) called\n";
  }
};

std::variant<NoDefConstr, int> v1;    // ERROR: can't default construct first type
```

The auxiliary type `std::monostate` provides the ability to deal with this situation and also provides the ability to simulate an empty state.

`std::monostate`

To support variants, where the first type has no default constructor, a special helper type is provided: `std::monostate`. Objects of type `std::monostate` always have the same state. Thus, they always compare equal. Their own purpose is to represent an alternative type so that the variant has *no value of any other type*.

That is, the `struct std::monostate` can serve as a first alternative type to make the variant type default constructible. For example:

```
std::variant<std::monostate, NoDefConstr> v2;  // OK
std::cout << "index: " << v2.index() << '\n';  // prints 0
```

To some extent you can interpret the state to signal emptiness.[4]

There are various ways to check for the monostate, which also demonstrates some of the other operations, you can call for variants:

```
if (v2.index() == 0) {
  std::cout << "has monostate\n";
}
if (!v2.index()) {
  std::cout << "has monostate\n";
}
if (std::holds_alternative<std::monostate>(v2)) {
  std::cout << "has monostate\n";
}
if (std::get_if<0>(&v2)) {
  std::cout << "has monostate\n";
}
```

[4] In principle, `std::monostate` can serve as any alternative not just the first one, but, of course, then this alternative does not help to make the variant default constructible.

```
if (std::get_if<std::monostate>(&v2)) {
  std::cout << "has monostate\n";
}
```

`get_if<>()` uses a *pointer* to a variant and returns a pointer to the current alternative if the current alternative is `T`. Otherwise it returns `nullptr`. This differs from `get<T>()`, which takes a reference to a variant and returns the current alternative by value if the provided type is correct, and throws otherwise.

As usual, you can assign a value of another alternative and even assign the `monostate`, signaling emptiness again:

```
v2 = 42;
std::cout << "index: " << v2.index() << '\n';  // index: 1

v2 = std::monostate{};
std::cout << "index: " << v2.index() << '\n';  // index: 0
```

Deriving from Variants

You can derive from `std::variant`. For example, you can define an aggregate deriving from a `std::variant<>` as follows:

```
class Derived : public std::variant<int, std::string> {
};

Derived d = {{"hello"}};
std::cout << d.index() << '\n';       // prints: 1
std::cout << std::get<1>(d) << '\n'; // prints: hello
d.emplace<0>(77);                     // initializes int, destroys string
std::cout << std::get<0>(d) << '\n'; // prints: 77
```

16.2 `std::variant<>` Types and Operations

This section describes the types and operations of `std::variant<>` in detail.

16.2.1 `std::variant<>` Types

In the header file `<variant>` the C++ standard library defines class `std::variant<>` as follows:

```
namespace std {
  template<typename Types...> class variant;
}
```

That is, `std::variant<>` is a *variadic* class template (a feature introduced with C++11, allowing to deal with an arbitrary number of types).

In addition, the following type and objects are defined:

- Type std::variant_size
- Type std::variant_alternative
- Value std::variant_npos
- Type std::monostate
- Exception class std::bad_variant_access, derived from std::exception.

Variants also use the objects std::in_place_type (of type std::in_place_type_t) and std::in_place_index (of type std::in_place_index_t) defined in utility>.

16.2.2 std::variant<> Operations

Table std::variant *Operations* lists all operations that are provided for std::variant<>.

Operation	Effect
constructors	Create a variant object (might call constructor for underlying type)
destructor	Destroys an variant object
=	Assign a new value
emplace<T>()	Assign a new value to the alternative having type T
emplace<Idx>()	Assign a new value to the alternative with index Idx
valueless_by_exception()	Returns whether the variant has no value due to an exception
index()	Returns the index of the current alternative
swap()	Swaps values between two objects
==, !=, <, <=, >, >=	Compare variant objects
hash<>	Function object type to compute hash values
holds_alternative<T>()	Returns whether there is a value for type T
get<T>()	Returns the value for the alternative with type T or throws
get<Idx>()	Returns the value for the alternative with index Idx or throws
get_if<T>()	Returns a pointer to the value for the alternative with type T or nullptr
get_if<Idx>()	Returns a pointer to the value for the alternative with index Idx or nullptr
visit()	Perform operation for the current alternative

Table 16.1. std::variant<> *Operations*

Construction

By default, the default constructor of a variant calls the default constructor of the first alternative:

```
std::variant<int, int, std::string> v1;  // sets first int to 0, index()==0
```

The alternative is *value initialized*, which means that is it `0`, `false`, or `nullptr`, for fundamental types.

If a value is passed for initialization, the best matching type is used:

```
std::variant<long, int> v2{42};
std::cout << v2.index() << '\n';    // prints 1
```

However, the call is ambiguous if two types match equally well:

```
std::variant<long, long> v3{42};           // ERROR: ambiguous
std::variant<int, float> v4{42.3};         // ERROR: ambiguous
std::variant<int, double> v5{42.3};        // OK
std::variant<int, long double> v6{42.3};   // ERROR: ambiguous

std::variant<std::string, std::string_view> v7{"hello"}; // ERROR: ambiguous
std::variant<std::string, std::string_view, const char*> v8{"hello"};  // OK
std::cout << v8.index() << '\n';    // prints 2
```

To pass more than one value for initialization, you have to use the `in_place_type` or `in_place_index` tags:

```
std::variant<std::complex<double>> v9{3.0, 4.0};      // ERROR
std::variant<std::complex<double>> v10{{3.0, 4.0}};   // ERROR
std::variant<std::complex<double>> v11{std::in_place_type<std::complex<double>>,
                                       3.0, 4.0};
std::variant<std::complex<double>> v12{std::in_place_index<0>, 3.0, 4.0};
```

You can also use the `in_place_index` tags to resolve ambiguities or overrule priorities during the initialization:

```
std::variant<int, int> v13{std::in_place_index<1>, 77};   // init 2nd int
std::variant<int, long> v14{std::in_place_index<1>, 77}; // init long, not int
std::cout << v14.index() << '\n';   // prints 1
```

You can even pass an initializer list followed by additional arguments:

```
// initialize variant with a set with lambda as sorting criterion:
auto sc = [] (int x, int y) {
            return std::abs(x) < std::abs(y);
          };
std::variant<std::vector<int>,
             std::set<int,decltype(sc)>> v15{std::in_place_index<1>,
                                             {4, 8, -7, -2, 0, 5},
                                             sc};
```

You can't use class template argument deduction for `std::variant<>`. and there is no `make_variant<>()` convenience function (unlike for `std::optional<>` and `std::any`). Both makes no sense, because the whole goal of a variant is to deal with multiple alternatives.

Accessing the Value

The usual way to access the value is to call get<>() for the corresponding alternative. You can pass its index or, provided a type is not used more than once, its type. For example:

```
std::variant<int, int, std::string> var; // sets first int to 0, index()==0

auto a = std::get<double>(var);          // compile-time ERROR: no double
auto b = std::get<4>(var);               // compile-time ERROR: no 4th alternative
auto c = std::get<int>(var);             // compile-time ERROR: int twice

try {
  auto s = std::get<std::string>(var);   // throws exception (first int currently set)
  auto i = std::get<0>(var);             // OK, i==0
  auto j = std::get<1>(var);             // throws exception (other int currently set)
}
catch (const std::bad_variant_access& e) {      // in case of an invalid access
  std::cout << "Exception: " << e.what() << '\n';
}
```

There is also an API to access the value with the option to check whether it exists:

```
if (auto ip = std::get_if<1>(&var); ip) {
  std::cout << *ip << '\n';
}
else {
  std::cout << "alternative with index 1 not set\n";
}
```

You must pass a pointer to a variant to get_if<>() and it either returns a pointer to the current value or nullptr. Note that here if with initialization is used, which enables to check against a value just initialized.

Another way to access the values of the different alternatives are variant visitors.

Changing the Value

Assignment and emplace() operations exist corresponding to the initializations:

```
std::variant<int, int, std::string> var; // sets first int to 0, index()==0
var = "hello";                           // sets string, index()==2
var.emplace<1>(42);                      // sets second int, index()==1
```

You can also use get<>() or get_if<>() to assign a new value to the current alternative:

```
std::variant<int, int, std::string> var; // sets first int to 0, index()==0
std::get<0>(var) = 77;                   // OK, because first int already set
std::get<1>(var) = 99;                   // throws exception (other int currently set)

if (auto p = std::get_if<1>(&var); p) {  // if second int set
```

```
  *p = 42;                                  // modify it
}
```

Another way to modify the values of the different alternatives are variant visitors.

Comparisons

For two variants of the same type (i.e., having the same alternatives in the same order) you can use the usual comparison operators. The operators act according to the following rules:

- A variant with a value of an earlier alternative is less than a variant with a value with a later alternative.
- If two variants have the same alternative the corresponding operators for the type of the alternatives are evaluated. Note that all objects of type `std::monostate` are always equal.
- Two variants with the special state `valueless_by_exception()` being true are equal. Otherwise, any variant with `valueless_by_exception()` being true is less than any other variant.

For example:

```
std::variant<std::monostate, int, std::string> v1, v2{"hello"}, v3{42};
std::variant<std::monostate, std::string, int> v4;

    v1 == v4    // COMPILE-TIME ERROR

    v1 == v2    // yields false
    v1 < v2     // yields true
    v1 < v3     // yields true
    v2 < v3     // yields false

v1 = "hello";

    v1 == v2    // yields true

v2 = 41;

    v2 < v3     // yields true
```

Move Semantics

`std::variant<>` also supports move semantics. If you move the object as a whole, the state gets copied and the value of the current alternative is moved. As a result, a moved-from object still has the same alternative, but any value becomes unspecified.

You can also move a value into or out of the contained object.

Hashing

The hash value for a variant object is enabled if and only if each member type can provide a hash value. Note that the hash value is *not* the hash value of the current alternative.

16.2.3 Visitors

They have to unambiguously provide a function call operator for each possible type. Then, the corresponding overload is used to deal with the current alternative.

Using Function Objects as Visitors

For example:

lib/variantvisit.cpp

```
#include <variant>
#include <string>
#include <iostream>

struct MyVisitor
{
  void operator() (int i) const {
    std::cout << "int:    " << i << '\n';
  }
  void operator() (std::string s) const {
    std::cout << "string: " << s << '\n';
  }
  void operator() (long double d) const {
    std::cout << "double: " << d << '\n';
  }
};

int main()
{
  std::variant<int, std::string, double> var(42);
  std::visit(MyVisitor(), var);  // calls operator() for int
  var = "hello";
  std::visit(MyVisitor(), var);  // calls operator() for string
  var = 42.7;
  std::visit(MyVisitor(), var);  // calls operator() for long double
}
```

The call of `visit()` is a compile-time error if not all possible types are supported by an `operator()` or the call is ambiguous. The example here works fine because `long double` is a better match for a `double` value than `int`.

You can also use visitors to modify the value of the current alternative (but not to assign a new alternative). For example:

```
struct Twice
{
  void operator()(double& d) const {
    d *= 2;
  }
  void operator()(int& i) const {
    i *= 2;
  }
  void operator()(std::string& s) const {
    s = s + s;
  }
};

std::visit(Twice(), var);  // calls operator() for matching type
```

Because only the type matters, you can't have different behavior for alternatives that have the same type.

Note that the function call operators should be marked as being `const`, because they are *stateless* (they don't change their behavior, only the passed value).

Using Generic Lambdas as Visitors

The easiest way to use this feature is to use a generic lambda, which is a function object for an arbitrary type:

```
auto printvariant = [](const auto& val) {
                        std::cout << val << '\n';
                    };
...
std::visit(printvariant, var);
```

Here, the generic lambda defines a closure type having the function call operator as member template:

```
class CompilerSpecifyClosureTypeName {
  public:
    template<typename T>
    auto operator() (const T& val) const {
        std::cout << val << '\n';
    }
};
```

Thus, the call of the lambda passed to `std::visit()` compiles if the statement in the generated function call operator is valid (i.e., calling the output operator is valid).

You can also use a lambda to modify the value of the current alternative:

```
// double the value of the current alternative:
std::visit([](auto& val) {
               val = val + val;
           },
           var);
```

Or:

```
// restore to the default value of the current alternative;
std::visit([](auto& val) {
               val = std::remove_reference_t<decltype(val)>{};
           },
           var);
```

You can even still handle the different alternatives differently using the compile-time `if` language feature. For example:

```
auto dblvar = [](auto& val) {
                 if constexpr(std::is_convertible_v<decltype(val),
                                                    std::string>) {
                    val = val + val;
                 }
                 else {
                    val *= 2;
                 }
              };
...
std::visit(dblvar, var);
```

Here, for a `std::string` alternative the call of the generic lambda instantiates its generic function call template to compute:

```
val = val + val;
```

while for other alternatives, such as `int` or `double`, the call of the lambda instantiates its generic function call template to compute:

```
val *= 2;
```

Using Overloaded Lambdas as Visitors

By using an *overloader* for function objects and lambdas, you can also define a set of lambdas where the best match is used as visitor.

Assume, the overloader is `overload` defined as follows:

tmpl/overload.hpp

```
// "inherit" all function call operators of passed base types:
template<typename... Ts>
struct overload : Ts...
{
  using Ts::operator()...;
};

// base types are deduced from passed arguments:
template<typename... Ts>
overload(Ts...) -> overload<Ts...>;
```

You can use `overload` to visit a variant by providing lambdas for each alternative:

```
std::variant<int, std::string> var(42);
...
std::visit(overload{    // calls best matching lambda for current alternative
             [](int i) { std::cout << "int: " << i << '\n'; },
             [](const std::string& s) {
                         std::cout << "string: " << s << '\n'; },
           },
           var);
```

You can also use generic lambdas. Always the best match is used. For example, to modify the current alternative of a variant, you can use the overload to "double" the value for strings and other types:

```
auto twice = overload{
                 [](std::string& s)  { s += s; },
                 [](auto& i) { i *= 2; },
             };
```

With this overload, for string alternatives the current value gets appended while for all other types the value is multiplied by 2, which demonstrates the following application for a variant:

```
std::variant<int, std::string> var(42);
std::visit(twice, var);  // value 42 becomes 84
...
var = "hi";
std::visit(twice, var);  // value "hi" becomes "hihi"
```

16.2.4 Valueless by Exception

When modifying a variant so that it gets a new value and this modification throws an exception, the variant can get into a very special state: The variant already lost its old value, but didn't get its new value. For example:

```
struct S {
  operator int() { throw "EXCEPTION"; }  // any conversion to int throws
```

```
};

std::variant<double,int> var{12.2};       // initialized as double
var.emplace<1>(S{});                      // OOPS: throws while set as int
```

If this happens, then:

- `var.valueless_by_exception()` returns `true`
- `var.index()` returns `std::variant_npos`

which signals that the variant holds no value at all.

The exact guarantees are as follows:

- If `emplace()` throws `valueless_by_exception()` is always set to `true`.
- If `operator=()` throws and the modification would not change the alternative `valueless_by_exception()` and `index()` keeps their old state. The state of the value depends on the exception guarantees of the value type.
- If `operator=()` throws and the new value would set a different alternative, the variant *might* hold no value (`valueless_by_exception()` *might* become `true`). This depends on when exactly the exception gets thrown. If it happens during a type conversion before the actual modification of the value started, the variant will still hold its old value.

Usually, this behavior should be no problem, provided you no longer use the variant you tried to modify. If you still want to use a variant although using it caused an exception, you should better check its state. For example:

```
std::variant<double,int> var{12.2};       // initialized as double
try {
  var.emplace<1>(S{});                    // OOPS: throws while set as int
}
catch (...) {
  if (!var.valueless_by_exception()) {
    ...
  }
}
```

16.3 Special Cases

Specific variants can result in special or unexpected behavior.

16.3.1 Having Both `bool` and `std::string` Alternatives

If a `std::variant<>` has both a `bool` and a `std::string` alternative, assigning string literals can become surprising because a string literal converts better to `bool` than to `std::string`. For example:

```
std::variant<bool, std::string> v;
```

```
v = "hi";    // OOPS: sets the bool alternative
std::cout << "index: " << v.index() << '\n';
std::visit([](const auto& val) {
               std::cout << "value: " << val << '\n';
           },
           v);
```

This code snippet will have the following output:

```
index: 0
value: true
```

Thus, the string literal is interpreted as initializing the variant by the Boolean value `true` (`true` because the pointer is not `0`).

There are a couple of options to "fix" the assignment here:

```
v.emplace<1>("hello");              // explicitly assign to second alternative

v.emplace<std::string>("hello");    // explicitly assign to string alternative

v = std::string{"hello"};           // make sure a string is assigned

using namespace std::literals;      // make sure a string is assigned
v = "hello"s;
```

16.4 Afternotes

Variant objects were first proposed 2005 by Axel Naumann in https://wg21.link/n4218 referring to Boost.Variant as a reference implementation. The finally accepted wording was formulated by Axel Naumann in https://wg21.link/p0088r3.

Tony van Eerd significantly improved the semantics for comparison operators with https://wg21.link/p0393r3. Vicente J. Botet Escriba harmonized the API with `std::optional<>` and `std::any` with https://wg21.link/p0032r3. Jonathan Wakely fixed the behavior for `in_place` tag types with https://wg21.link/p0504r0. The restriction to disallow references, incomplete types, and arrays, and empty variants was formulated by Erich Keane with https://wg21.link/p0510r0.

Chapter 17

`std::any`

In general, C++ is a language with type binding and type safety. Value objects are declared to have a specific type, which defines which operations are possible and how they behave. And the value objects can't change their type.

`std::any` is a value type that is able to change its type, while still having type safety. That is, objects can hold values of any arbitrary type but they know which type the value has they currently hold. There is no need to specify the possible types when declaring an object of this type.

The trick is that objects contain both the *contained* value and the type of the contained value using `typeid`. Because the value can have any size the memory might be allocated on the heap. However, implementations should avoid the use of heap memory for small contained values, such as `int`.

That is, if you assign a string, the object allocates memory for the value and copies the string, while also storing internally that a string was assigned. Later, run-time checks can be done to find out, which type the current value has and to use that value as its type a `any_cast<>` is necessary.

As for `std::optional<>` and `std::variant<>` the resulting objects have value semantics. That is, copying happens deeply by creating an independent object with the current contained value and its type in its own memory. Because heap memory might be involved, copying a `std::any` usually is expensive and you should prefer to pass objects by reference or move values. Move semantics is partially supported.

17.1 Using `std::any`

The following example demonstrates the core abilities of `std::any`:

```
std::any a;                 // a is empty
std::any b = 4.3;           // b has value 4.3 of type double
a = 42;                     // a has value 42 of type int
b = std::string{"hi"};      // b has value "hi" of type std::string

if (a.type() == typeid(std::string)) {
```

```
  std::string s = std::any_cast<std::string>(a);
  useString(s);
}
else if (a.type() == typeid(int)) {
  useInt(std::any_cast<int>(a));
}
```

You can declare a `std::any` to be empty or to be initialized by a value of a specific type. The type of the initial value become the type of the *contained* value.

By using the member function `type()` you can check the type of the contained value against the type ID of any type. If the object is empty, the type ID is `typeid(void)`.

To access the contained value you have to cast it to its type with a `std::any_cast<>`:

```
auto s = std::any_cast<std::string>(a);
```

If the cast fails, because the object is empty or the contained type doesn't fit, a `std::bad_any_cast` is thrown. Thus, without checking or knowing the type, you better implement the following:

```
try {
  auto s = std::any_cast<std::string>(a);
  ...
}
catch (std::bad_any_cast& e) {
  std::cerr << "EXCEPTION: " << e.what() << '\n';
}
```

Note that `std::any_cast<>` creates an object of the passed type. If you pass `std::string` as template argument to `std::any_cast<>`, it creates a temporary string (a prvalue), which is then used to initialize the new object `s`. Without such an initialization, it is usually better to cast to a reference type to avoid creating a temporary object:

```
std::cout << std::any_cast<const std::string&>(a);
```

To be able to modify the value, you need a cast to the corresponding reference type:

```
std::any_cast<std::string&>(a) = "world";
```

You can also call `std::any_cast` for the address of a `std::any` object. In that case, the cast returns a corresponding pointer if the type fits or `nullptr` if not:

```
auto p = std::any_cast<std::string>(&a);
if (p) {
  ...
}
```

To empty an existing `std::any` object you can call:

```
a.reset();    // makes it empty
```

or:

```
a = std::any{};
```

or just:

```
a = {};
```

And you can directly check, whether the object is empty:

```
if (a.has_value()) {
  ...
}
```

Note also that values are stored using their *decayed* type (arrays convert to pointers, and top-level references and `const` are ignored). For string literals this means that the value type is `const char*`. To check against `type()` and use `std::any_cast<>` you have to use exactly this type:

```
std::any a = "hello";   // type() is const char*
if (a.type() == typeid(const char*)) {                  // true
  ...
}
if (a.type() == typeid(std::string)) {                  // false
  ...
}
std::cout << std::any_cast<const char*>(v[1]) << '\n';  // OK
std::cout << std::any_cast<std::string>(v[1]) << '\n';  // EXCEPTION
```

These are more or less all operations. No comparison operators are defined (so, you can't compare or sort objects), no hash function is defined, and no `value()` member functions are defined. And because the type is only known at run time, no generic lambdas can be used to deal with the current value independent from its type. You always need the run-time function `std::any_cast<>` to be able to deal with the current value, which means that you need *some* type specific code to reenter the C++ type system when dealing with values.

However, it is possible, to put `std::any` objects in a container. For example:

```
std::vector<std::any> v;

v.push_back(42);
std::string s = "hello";
v.push_back(s);

for (const auto& a : v) {
  if (a.type() == typeid(std::string)) {
    std::cout << "string: " << std::any_cast<const std::string&>(a) << '\n';
  }
  else if (a.type() == typeid(int)) {
    std::cout << "int: " << std::any_cast<int>(a) << '\n';
  }
}
```

17.2 std::any Types and Operations

This section describes the types and operations of std::any in detail.

17.2.1 Any Types

In the header file <any> the C++ standard library defines class std::any as follows:

```
namespace std {
  class any;
}
```

That is, std::any is no class template at all.

In addition, the following type and objects are defined:

- Exception class std::bad_any_cast, which is derived from std::bad_cast, which is derived from std::exception, if type conversions fail.

Any objects also use the objects std::in_place_type (of type std::in_place_type_t) defined in utility>.

17.2.2 Any Operations

Table std::any *Operations* lists all operations that are provided for std::any.

Operation	Effect
constructors	Create a any object (might call constructor for underlying type)
make_any()	Create a any object (passing value(s) to initialize it)
destructor	Destroys an any object
=	Assign a new value
emplace<T>()	Assign a new value having the type T
reset()	Destroys any value (makes the object empty)
has_value()	Returns whether the object has a value
type()	Returns the current type as std::type_info object
any_cast<T>()	Use current value as value of type T (exception if other type)
swap()	Swaps values between two objects

Table 17.1. std::any *Operations*

Construction

By default, a std::any is initialized by being empty.

```
std::any a1;          // a1 is empty
```

If a value is passed for initialization, its *decayed* type is used as type of the contained value:

```
std::any a2 = 42;        // a2 contains value of type int
std::any a3 = "hello";   // a2 contains value of type const char*
```

To hold a different type than the type of the initial value, you have to use the `in_place_type` tags:

```
std::any a4{std::in_place_type<long>, 42};
std::any a5{std::in_place_type<std::string>, "hello"};
```

Even the type passed to `in_place_type` decays. The following declaration holds a `const char*`:

```
std::any a5b{std::in_place_type<const char[6]>, "hello"};
```

To initialize an optional object by multiple arguments, you have to create the object or add `std::in_place_type` as first argument (the contained type can't be deduced):

```
std::any a6{std::complex{3.0, 4.0}};
std::any a7{std::in_place_type<std::complex<double>>, 3.0, 4.0};
```

You can even pass an initializer list followed by additional arguments:

```
// initialize a std::any with a set with lambda as sorting criterion:
auto sc = [] (int x, int y) {
            return std::abs(x) < std::abs(y);
          };
std::any a8{std::in_place_type<std::set<int,decltype(sc)>>,
            {4, 8, -7, -2, 0, 5},
            sc};
```

Note that there is also a convenience function `make_any<>()`, which can be used for single or multiple arguments (without the need for the `in_place_type` argument). You always have to explicitly specify the initialized type (it is not deduced if only one argument is passed):

```
auto a10 = std::make_any<float>(3.0);
auto a11 = std::make_any<std::string>("hello");
auto a13 = std::make_any<std::complex<double>>(3.0, 4.0);
auto a14 = std::make_any<std::set<int,decltype(sc)>>({4, 8, -7, -2, 0, 5},
                                                     sc);
```

Changing the Value

Corresponding assignment and `emplace()` operations exist. For example:

```
std::any a;

a = 42;        // a contains value of type int
a = "hello";   // a contains value of type const char*

a.emplace{std::in_place_type<std::string>, "hello"};
              // a contains value of type std::string
```

```
a.emplace{std::in_place_type<std::complex<double>>, 4.4, 5.5};
                // a contains value of type std::complex<double>
```

Accessing the Value

To access the contained value you have to cast it to its type with a `std::any_cast<>`. To cast the value to a string you have several options:

```
std::any_cast<std::string>(a)           // yield copy of the value

std::any_cast<std::string&>(a);         // write value by reference

std::any_cast<const std::string&>(a);   // read-access by reference
```

Here, if the cast fails, a `std::bad_any_cast` exception is thrown.

The type fits if the passed type with top-level references removed has the same type ID.

You can pass an address to get the `nullptr` if the cast fails, because the current type doesn't fit:

```
std::any_cast<std::string>(&a)          // write-access via pointer

std::any_cast<const std::string>(&a);   // read-access via pointer
```

Note that here casting to a reference results in a run-time error:

```
std::any_cast<std::string&>(&a);        // RUN-TIME ERROR
```

Move Semantics

`std::any` also supports move semantics. However, note that move semantics is only supported for type that also have copy semantics. That is, **move-only types** are not supported as contained value types.

The best way to deal with move semantics might not be obvious. So, here is how you should do it:

```
std::string s("hello, world!");

std::any a;
a = std::move(s);                               // move s into a

s = std::move(std::any_cast<string&>(a));   // move assign string in a to s
```

As usual for moved-from objects the contained value of `a` is unspecified after the last call. Thus, you can use `a` as a string as long as you don't make any assumption about which value the contained string value has.

Note that:

```
s = std::any_cast<string>(std::move(a));
```

also works, but needs an additional move. However, the following is dangerous (although it is an example inside the C++ standard):

```
std::any_cast<string&>(a) = std::move(s2); // OOPS: a to hold a string
```

This only works if the contained value of a already is a string. If not, the cast throws a `std::bad_any_cast` exception.

17.3 Afternotes

Any objects were first proposed 2006 by Kevlin Henney and Beman Dawes in https://wg21.link/n1939 referring to Boost.Any as a reference implementation. This class was adopted to become part of the Library Fundamentals TS as proposed by Beman Dawes, Kevlin Henney, and Daniel Krügler in https://wg21.link/n3804.

The class was adopted with other components for C++17 as proposed by Beman Dawes and Alisdair Meredith in https://wg21.link/p0220r1.

Vicente J. Botet Escriba harmonized the API with `std::variant<>` and `std::optional<>` with https://wg21.link/p0032r3. Jonathan Wakely fixed the behavior for `in_place` tag types with https://wg21.link/p0504r0.

This page is intentionally left blank

Chapter 18

`std::byte`

Programs hold data in memory. With `std::byte` C++17 introduces a type for it, which does represent the "natural" type of the elements of memory, bytes. The key difference to types like `char` or `int` is that this type cannot (easily) be (ab)used as an integral value or character type. For cases, where numeric computing or character sequence are not the goal, this results in more type safety. The only "computing" operations supported are bit-wise operators.

18.1 Using `std::byte`

The following example demonstrates the core abilities of `std::byte`:

```
#include <cstddef>  // for std::byte

std::byte b1{0x3F};
std::byte b2{0b1111'0000};

std::byte b3[4] {b1, b2, std::byte{1}};  // 4 bytes (last is 0)

if (b1 == b3[0]) {
  b1 <<= 1;
}

std::cout << std::to_integer<int>(b1) << '\n';  // outputs: 126
```

Here, we define two bytes with two different initial values. `b2` is initialized used two features available since C++14:

- The prefix `0b` enables to define binary literals.
- The *digit separator* `'` allows making numeric literals more readable in source code (it can be placed between any two digits of numeric literals).

Note that list initialization (using curly braces) is the only way you can directly initialize a single value of a `std::byte` object. All other forms do not compile:

```
std::byte b1{42};      // OK (as for all enums with fixed underlying type since C++17)
std::byte b2(42);      // ERROR
std::byte b3 = 42;     // ERROR
std::byte b4 = {42}; // ERROR
```

This is a direct consequence of the fact that `std::byte` is implemented as an enumeration type, using the new way scoped enumerations can be initialized with integral values.

There is also no implicit conversion, so that you have to initialize the byte array with an explicitly converted integral literal:

```
std::byte b5[] {1};               // ERROR
std::byte b6[] {std::byte{1}}; // OK
```

Without any initialization, the value of a `std::byte` is undefined for object on the stack:

```
std::byte b;           // undefined value
```

As usual (except for atomics), you can force an initialization with all bits set to zero with list initialization:

```
std::byte b{};         // same as b{0}
```

`std::to_integer<>()` provides the ability to use the byte object as integral value (including `bool` and `char`). Without the conversion, the output operator would not compile. Note that because it is a template you even need the conversion fully qualified with `std::`:

```
std::cout << b1;                          // ERROR
std::cout << to_integer<int>(b1);         // ERROR (ADL doesn't work here)
std::cout << std::to_integer<int>(b1); // OK
```

Such a conversion is also necessary to use a `std::byte` as a Boolean value. For example:

```
if (b2) ...                           // ERROR
if (b2 != std::byte{0}) ...           // OK
if (to_integer<bool>(b2)) ...         // ERROR (ADL doesn't work here)
if (std::to_integer<bool>(b2)) ... // OK
```

Because `std::byte` is define as enumeration type with `unsigned char` as the underlying type, the size of a `std::byte` is always 1:

```
std::cout << sizeof(b);       // always 1
```

The number of bits depends on the number of bits of type `unsigned char`, which you can find out with the standard numeric limits:

```
std::cout << std::numeric_limits<unsigned char>::digits;  // number of bits of a std::byte
```

Most of the time it's 8, but there are platforms where this is not the case.

18.2 std::byte Types and Operations

This section describes the types and operations of std::byte in detail.

18.2.1 std::byte Types

In the header file <cstddef> the C++ standard library defines type std::byte as follows:

```
namespace std {
  enum class byte : unsigned char {
  };
}
```

That is, std::byte is nothing but a scoped enumeration type with some supplementary bit-wise operators defined:

```
namespace std {
  ...
  template<typename IntType>
   constexpr byte  operator<< (byte b, IntType shift) noexcept;
  template<typename IntType>
   constexpr byte& operator<<= (byte& b, IntType shift) noexcept;
  template<typename IntType>
   constexpr byte  operator>> (byte b, IntType shift) noexcept;
  template<typename IntType>
   constexpr byte& operator>>= (byte& b, IntType shift) noexcept;

  constexpr byte&  operator|= (byte& l, byte r) noexcept;
  constexpr byte   operator| (byte l, byte r) noexcept;
  constexpr byte&  operator&= (byte& l, byte r) noexcept;
  constexpr byte   operator& (byte l, byte r) noexcept;
  constexpr byte&  operator^= (byte& l, byte r) noexcept;
  constexpr byte   operator^ (byte l, byte r) noexcept;
  constexpr byte   operator~ (byte b) noexcept;

  template<typename IntType>
    constexpr IntType to_integer (byte b) noexcept;
}
```

18.2.2 std::byte Operations

Table std::byte *Operations* lists all operations that are provided for std::byte.

Operation	Effect
constructors	Create a byte object (value undefined with default constructor)
destructor	Destroys a byte object (nothing to be done)
=	Assign a new value
==, !=, <, <=, >, >=	Compares byte objects
<<, >>, |, &, ^, ~	Binary bit-operations
<<=, >>=, |=, &=, ^=	Modifying bit-operations
to_integer<T>()	Converts byte object to integral type T
sizeof()	Yields 1

Table 18.1. std::byte *Operations*

Conversion to Integral Types

By using to_integer<>() you can convert a std::byte to any fundamental integral type (bool, a character type, or an integer type). This is, for example, necessary to compare a std::byte with a numeric value or to use it in a condition:

```
if (b2) ...                          // ERROR
if (b2 != std::byte{0}) ...          // OK
if (to_integer<bool>(b2)) ...        // ERROR (ADL doesn't work here)
if (std::to_integer<bool>(b2)) ...   // OK
```

Another example use is std::byte I/O

to_integer<>() uses the rules for static casts from an unsigned char to the destination type. For example:

```
std::byte ff{0xFF};
std::cout << std::to_integer<unsigned int>(ff);  // 255
std::cout << std::to_integer<int>(ff);           // also 255 (no negative value)
std::cout << static_cast<int>(std::to_integer<signed char>(ff));  // -1
```

I/O with std::byte

There are no input and output operators defined for std::byte, so that you have to convert them to an integral value:

```
std::byte b;
...
std::cout << std::to_integer<int>(b);              // prints value as decimal value
std::cout << std::hex << std::to_integer<int>(b);  // prints value as hexadecimal value
```

By using a std::bitset<>, you can also output the value as binary value (a sequence of bits):

```
#include <bitset>
#include <limits>
```

```
using ByteBitset = std::bitset<std::numeric_limits<unsigned char>::digits>;
std::cout << ByteBitset{std::to_integer<unsigned char>(b1)};
```

The using declaration defines a bitset type with the numbers of bits as `std::byte` has and then we create and output such an object initialized with the integral type of the byte.

You can also use this to write the binary representation of a `std::byte` into a string:

```
std::string s = ByteBitset{std::to_integer<unsigned char>(b1)}.to_string();
```

Input is possible the similar way: Just read the value as an integral, string or bitset value and convert it. For example, you can write an input operator that reads a byte from a binary representation as follows:

```
std::istream& operator>> (std::istream& strm, std::byte& b)
{
    // read into a bitset:
    std::bitset<std::numeric_limits<unsigned char>::digits> bs;
    strm >> bs;
    // without failure, convert to std::byte:
    if (! std::cin.fail()) {
      b = static_cast<std::byte>(bs.to_ulong());  // OK
    }
    return strm;
}
```

Note that we have to use a `static_cast<>()` to convert to bitset converted to an unsigned long into a `std::byte`. A list initialization would not work because the conversion narrows:[1]

```
b = std::byte{bs.to_ulong()};  // ERROR: narrowing
```

and we have no other way of initialization.

18.3 Afternotes

`std::byte` was first proposed by Neil MacIntosh passing in https://wg21.link/p0298r0. The finally accepted wording was formulated by Neil MacIntosh in https://wg21.link/p0298r3.

[1] With gcc/g++ narrowing initializations compile without the compiler option `-pedantic-errors`.

This page is intentionally left blank

Chapter 19

String Views

With C++17, a special string class was adopted by the C++ standard library, that allows us to deal with character sequences like strings, without allocating memory for them: `std::string_view`. That is, `std::string_view` objects refer to external character sequences without owning them. That is, the object can be considered as a *reference* to a character sequence.

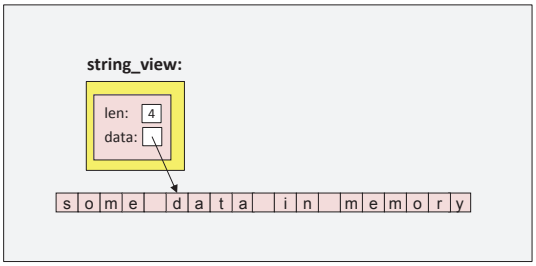

Figure 19.1. String View Objects

Using such a string view is cheap and fast (passing a `string_view` by value is always cheap). However, it is also potentially dangerous, because similar to raw pointers it is up to the programmer to ensure that the referred character sequences is still valid, when using a `string_view`).

19.1 Differences to `std::string`

In contrast to `std::string`, `std::string_view` objects have the following properties:

- The underlying character sequences are read-only. There are no operations that allow the modification of the characters. You can only assign a new value, swap values, and remove characters at the beginning or the end.

- The character sequences are not guaranteed to be null terminated. So, a string view is not a *null terminated byte stream* (*NTBS*).
- The value can be the `nullptr`, which for example is returned by `data()` after initializing a string view with the default constructor.
- There is no allocator support.

Due to the possible `nullptr` value and possible missing null terminator, you should always use `size()` before accessing characters via `operator[]` or `data()` (unless you know better).

19.2 Using String Views

There are two major applications of string views:

1. You might have allocated or mapped data with character sequences or strings and want to use this data without allocating more memory. Typical examples are the use of memory mapped files or dealing with substrings in large texts.
2. You want to improve the performance for functions/operations that receive strings just to directly process them read-only not needing a trailing null terminator.

   A special form of this might be to deal with string literals as objects having an API similar to strings:

   ```
   static constexpr std::string_view hello{"hello world"};
   ```

The first example usually means that usually only string view are passed around, while the programming logic has to ensure that the underlying character sequences remain valid (i.e., the mapped file content is not unmapped). At any time you might use the string view to initialize or assign their value to a `std::string`.

But beware, using string views just like "the better string." This can result to worse performance and to severe run-time errors. So read the following subsections carefully.

19.3 Using String Views Similar to Strings

A first example, using a string_view like a read-only string, is a function that prints a collection of elements with a prefix passed as a string view:

```
#include <string_view>

template<typename T>
void printElems(const T& coll, std::string_view prefix = std::string_view{})
{
  for (const auto& elem : coll) {
    if (prefix.data()) {  // check against nullptr
      std::cout << prefix << ' ';
    }
    std::cout << elem << '\n';
```

```
    }
  }
```

Here, just by declaring that the function will take a `std::string_view`, we might save a call to allocate heap memory compared to a function taking a `std::string`. Details depend on whether short strings are passed and the short string optimization (SSO) is used. For example, if we declare the function as follows:

```
template<typename T>
void printElems(const T& coll, const std::string& prefix = std::string{});
```

and we pass a string literal, the call creates a temporary string which will allocate memory unless the string is short and the short string optimization is used. By using a string view instead, no allocation is needed, because the string view only *refers* to the string literal.

However, note that `data()` has to be checked against the `nullptr` before using any unknown value of a string view.

Another example, using a string_view like a read-only string, is an improved version of the asInt() example of `std::optional<>`, which was declared for a string parameter:

lib/asint.cpp

```
#include <optional>
#include <string_view>
#include <utility>      // for from_char()
#include <iostream>

// convert string to int if possible:
std::optional<int> asInt(std::string_view sv)
{
  int val;
  // read character sequence into the int:
  auto [ptr, ec] = std::from_chars(sv.data(), sv.data()+sv.size(),
                                   &val);
  // if we have an error code, return no value:
  if (ec != std::errc{}) {
    return std::nullopt;
  }
  return val;
}

int main()
{
  for (auto s : {"42", "  077", "hello", "0x33"} ) {
    // convert s to int and use the result if possible:
    std::optional<int> oi = asInt(s);
    if (oi) {
```

```
      std::cout << "convert '" << s << "' to int: " << *oi << "\n";
    }
    else {
      std::cout << "can't convert '" << s << "' to int\n";
    }
  }
}
```

Now, `asInt()` takes a string view by value. However, that has significant consequences. First, it does no longer make sense to use `std::stoi()` to create the integer, because `stoi()` takes a string and creating a string from a string view is a relative expensive operation.

Instead, we pass the range of characters of the string view to the new standard library function `std::from_chars()`. It takes a pair of raw character pointers for the begin and end of the characters to convert. Note that this means that we can skip any special handling of an empty string view, where `data()` is `nullptr` and `size()` is `0`, because the range from `nullptr` until `nullptr+0` is a valid empty range (for any pointer type adding `0` is supported and has no effect).

`std_from_chars()` returns a `std::from_chars_result`, which is a structure with two members, a pointer `ptr` to the first character that was not processed and a `std::errc ec`, for which `std::errc` represents no error. Thus, after initializing `ec` with the `ec` member of the return value (using structured bindings), the following check returns `nullopt` if the conversion failed:

```
if (ec != std::errc{}) {
  return std::nullopt;
}
```

19.3.1 String View Considered Harmful

Usually "smart objects" such as smart pointers are considered to be safer (or at least not more dangerous) than corresponding language features. So, the impression might be that a string view, which is a kind of string reference, is safer or at least as safe as using string references. But unfortunately this is not the case. String views are in fact more dangerous than string references or smart pointers. They behave more like raw character pointers.

Don't Assign Strings to String Views

Consider we declare a function returning a new string:

```
std::string retString();
```

Using the return value is usually pretty safe:

- Assigning it to a string or an object declared with `auto` is safe (but moves, which is usually OK, but doesn't have the best performance):

```
auto std::string s1 = retString();      // safe
```

- Assigning the return value to a string reference is, if possible, pretty safe as long we use the object locally because references extend the lifetime of return values to the end of their lifetime:

```
std::string& s2 = retString();          // Compile-Time ERROR (const missing)

const std::string& s3 = retString();    // s3 extends lifetime of returned string
std::cout << s3 << '\n';                // OK

auto&& s4 = retString();                // s4 extends lifetime of returned string
std::cout << 4 << '\n';                 // OK
```

For a string view this safety is not given. It does *neither* copy *nor* extend the lifetime of a return value:

```
std::string_view sv = retString();    // sv does NOT extend lifetime of returned string
std::cout << sv << '\n';              // RUN-TIME ERROR: returned string destructed
```

Here, the returned string gets destructed at the end of the first statement so that referring to it from the string view `sv` is a fatal run-time error resulting in undefined behavior.

The problem is the same as when calling:

```
const char* p = retString().c_str();
```

or:

```
auto p = retString().c_str();
```

For this reason, you should also be very careful with returning a string view:[1]

```
// very dangerous:
std::string_view substring(const std::string&, std::size_t idx = 0);

// because:
auto sub = substring("very nice", 5);  // returns view to passed temporary string
                                       // but temporary string destructed after the call
std::cout << sub << '\n';              // RUN-TIME ERROR: tmp string already destructed
```

Don't Return String Views to Strings

Especially it is a very dangerous design to let getters of string members return a string view. Thus you should ***not*** implement the following:

```
class Person {
  std::string name;
 public:
```

[1] See https://groups.google.com/a/isocpp.org/forum/#!topic/std-discussion/Gj5gt5E-po8 for a discussion about this example.

```
    Person (std::string n) : name{std::move(n)} {
    }
    std::string_view getName() const {  // don't do this
        return name;
    }
};
```

Because, again, as a result the following would become to be a fatal run-time error causing undefined behavior:

```
Person createPerson();

auto n = createPerson().getName();    // OOPS: deletes temporary string
std::cout << "name: " << n << '\n';   // FATAL RUN-TIME ERROR
```

Again this is a problem you would not have if `getName()` would return a string by value or by reference.

Function Templates should use Return Type `auto`

Note that it is easy to accidentally assign returned strings to string views. Consider, for example, the definition of two functions that both alone look pretty useful:

```
// define + for string views returning string:
std::string operator+ (std::string_view sv1, std::string_view sv2) {
    return std::string(sv1) + std::string(sv2);
}

// generic concatenation:
template<typename T>
T concat (const T& x, const T& y) {
    return x + y;
}
```

However, using them together again might easily result in fatal run-time error:

```
std::string_view hi = "hi";
auto xy = concat(hi, hi);  // xy is std::string_view
std::cout << xy << '\n';   // FATAL RUN-TIME ERROR: referred string destructed
```

Code like that can easily accidentally be written. The real problem here is the return type of `concat()`. If declaring its return type to be deduced by the compiler, the example above initializes `xy` as std::string:

```
// improved generic concatenation:
template<typename T>
auto concat (const T& x, const T& y) {
    return x + y;
}
```

Also, it is counter-productive to use string views in a chain of calls, where in the chain or at then end of it string are needed. For example, if you define class `Person` with the following constructor:

```
class Person {
  std::string name;
 public:
  Person (std::string_view n) : name{n} {
  }
  ...
};
```

Passing a string literals or string you still need is fine:

```
Person p1{"Jim"};          // no performance overhead
std::string s = "Joe";
Person p2{s};              // no performance overhead
```

But moving in a string becomes unnecessary expensive, because the passed string is first implicitly converted to a string view, which is then used to create a new string allocating memory again:

```
Person p3{std::move(s)};   // performance overhead:  move broken
```

Don't deal with `std::string_view` here. Taking the parameter by value and moving it to the member is still the best solution. Thus, the constructor and getter should look as follows:

```
class Person {
  std::string name;
 public:
  Person (std::string n) : name{std::move(n)} {
  }
  std::string getName() const {
      return name;
  }
};
```

Summary of Safe Usage of String Views

To summarize, **use `std::string_view` with care**, which mean that you should also change the general style you program:

- Don't use string views in API's that pass the argument to a string.
  - Don't initialize string members from string view parameters.
  - No string at the end of a string view chain.
- Don't return a string view.
  - Unless it is just a forwarded input argument or you signal the danger by, for example, naming the function accordingly.
- For this reason, **function templates** should never return the type T of a passed generic argument.
  - Return `auto` instead.

- Never use a returned value to initialize a string view.
- For this reason, **don't assign** the return value of a function template returning a generic type to `auto`.
  - This means, the AAA (Almost Always Auto) pattern is broken with string view.

If these rules are too complicated or hard to follow, don't use `std::string_view` at all (unless you know what you do).

19.4 String View Types and Operations

This section describes the types and operations of string views in detail.

19.4.1 Concrete String View Types

In the header file `<string_view>` the C++ standard library provides a couple of specializations of class `basic_string_view<>`:

- Class `std::string_view` is the predefined specialization of that template for characters of type `char`:

```
namespace std {
    using string_view = basic_string_view<char>;
}
```

- For strings that use wider character sets, such as Unicode or some Asian character sets, three other types are predefined:

```
namespace std {
    using u16string_view = basic_string_view<char16_t>;
    using u32string_view = basic_string_view<char32_t>;
    using wstring_view   = basic_string_view<wchar_t>;
}
```

In the following sections, no distinction is made between these types of string views. The usage and the problems are the same because all string view classes have the same interface. So, "string view" means any string view type: `string_view`, `u16string_view`, `u32string_view`, and `wstring_view`. The examples in this book usually use type `string_view` because the European and Anglo-American environments are the common environments for software development.

19.4.2 String View Operations

Table *String View Operations* lists all operations that are provided for string views.

Except for `remove_prefix()` and `remove_suffix()`, all operations of string views are also provided for `std::string`s. However, the guarantees might slightly differ because for string views the value returned by `data()` might be `nullptr` and the missing guarantee end the sequence by a null terminator.

Operation	Effect
constructors	Create or copy a string view
destructor	Destroys a string view
`=`	Assign a new value
`swap()`	Swaps values between two strings view
`==`, `!=`, `<`, `<=`, `>`, `>=`, `compare()`	Compare string views
`empty()`	Returns whether the string view is empty
`size()`, `length()`	Return the number of characters
`max_size()`	Returns the maximum possible number of characters
`[]`, `at()`	Access a character
`front()`, `back()`	Access the first or last character
`<<`	Writes the value to a stream
`copy()`	Copies or writes the contents to a character array
`data()`	Returns the value as `nullptr` or constant character array (note: no terminating null character)
find functions	Search for a certain substring or character
`begin()`, `end()`	Provide normal iterator support
`cbegin()`, `cend()`	Provide constant iterator support
`rbegin()`, `rend()`	Provide reverse iterator support
`crbegin()`, `crend()`	Provide constant reverse iterator support
`substr()`	Returns a certain substring
`remove_prefix()`	Remove leading characters
`remove_suffix()`	Remove trailing characters
`hash<>`	Function object type to compute hash values

Table 19.1. String View Operations

Construction

You can create a string view with the default constructor, as a copy, from a raw character array (null terminated or with specified length), from a `std::string`, or as a literal with the suffix `sv`. However, note the following:

- String views created with the default constructor have `nullptr` as `data()`. Thus, there is no valid call of `operator[]`.

```
std::string_view sv;
auto p = sv.data();    // yields nullptr
std::cout << sv[0];    // ERROR: no valid character
```

- When initializing a string view by a null terminated byte stream, the resulting size is the number of characters without `'\0'` and using the index of the terminating null character is not valid:

```
std::string_view sv{"hello"};
std::cout << sv;           // OK
```

```
std::cout << sv.size();    // 5
std::cout << sv.at(5);     // throws std::out_of_range exception
std::cout << sv[5];        // undefined behavior, but HERE it usually works
std::cout << sv.data();    // undefined behavior, but HERE it usually works
```

The last two calls are formally undefined behavior. Thus, they are not guaranteed to work, although in this case you can assume to have null terminator right after the last character.

You can initialize a string view having the null terminator as part of its value by passing the number of characters including the null terminator:

```
std::string_view sv{"hello", 6};  // NOTE: 6 to include '\0'
std::cout << sv.size();    // 6
std::cout << sv.at(5);     // OK, prints the value of '\0'
std::cout << sv[5];        // OK, prints the value of '\0'
std::cout << sv.data();    // OK
```

- To create a string view from a `std::string` an implicit conversion operator is provided in class `std::string`. Again, having the null terminator right after the last character, which is usually guaranteed for a string, is not guaranteed to exist for the string view:

```
std::string s = "hello";
std::cout << s.size();     // 5
std::cout << s.at(5);      // OK, prints the value of '\0'
std::cout << s[5];         // OK, prints the value of '\0'

std::string_view sv{s};
std::cout << sv.size();    // 5
std::cout << sv.at(5);     // throws std::out_of_range exception
std::cout << sv[5];        // undefined behavior, but HERE it usually works
std::cout << sv.data();    // undefined behavior, but HERE it usually works
```

- As the literal operator is defined for the suffix `sv`, you can also create a string view as follows:

```
using namespace std::literals;
auto s = "hello"sv;
```

The key point here is that in general you should not expect the null terminating character and **always** use `size()` before accessing the characters (unless you know specific things about the value).

As a workaround you can make `'\0'` part of the string view, but you should not use a string view as a null terminates string without the null terminator being part of it, even if the null terminator is right behind.[2]

[2] Unfortunately, people are already starting to propose new C++ standards based on this strange state (having a string view without null terminator in front of a null terminator, which is not part of the string view itself). See https://wg21.link/P0555r0 for an example.

Hashing

The C++ standard library guarantees that hash values for strings and strings views are equal.

Modifying a String View

There are only a few operations provided to modify a string view:

- You can assign a new value or swap the values of two string views:

```
std::string_view sv1 = "hey";
std::string_view sv2 = "world";
sv1.swap(sv2);
sv2 = sv1;
```

- You can skip leading or trailing character (i.e., move the beginning to a character behind the first character or move the end to a character before the last character).

```
std::string_view sv = "I like my kindergarten";
sv.remove_prefix(2);
sv.remove_suffix(8);
std::cout << sv;    // prints: like my kind
```

Note that there is no support for `operator+`. Thus:

```
std::string_view sv1 = "hello";
std::string_view sv2 = "world";
auto s1 = sv1 + sv2;                // ERROR
```

One of the operands has to be a string:

```
auto s2 = std::string(sv1) + sv2;  // OK
```

Note that there is no implicit conversion to a string, because this is an expensive operation because it might allocate memory. For this reason, only the explicit conversion is possible.[3]

19.4.3 String View Support by Other Types

In principle, each place where a string can be passed also passing a string view could make sense, expect when the receiver needs that value to be null terminated (e.g., by passing the value to a C function for strings).

However, so far, we've only added support for the most important places:

- Strings can use or be combined with string views wherever useful. You can create a string from it (the constructor is explicit), assign, append, insert, replace, compare, or find a substring by passing a string view also.

  There is also in implicit conversion from a string to a string view.

[3] In principle, we could standardize an operation to concatenate string views yielding a new string, but so far this is not provided.

- You can pass a string view to `std::quoted`, which prints its value quoted. For example:

```
using namespace std::literals;

auto s = R"(some\value)"sv;      // raw string view
std::cout << std::quoted(s);     // output: "some\\value"
```

- You can initialize, extend or compare filesystem paths with string views.

However, there is, for example, no support for string views in the regex component of the C++ standard library, yet.

19.5 Using String Views in API's

String views are cheap and each `std::string` can be used as a string view. So, it seems `std::string_view` is the better type to deal with string parameters. Well details matter...

First, using a `std::string_view` only makes sense if the function using the parameter has the following constraints:

- It doesn't expect a null terminator at the end. This, for example, is not the case when passing the argument to a C functions as a single `const char*`).
- It respects the lifetime of the passed argument. Usually this means that the receiving function uses the passed value only until it ends.
- The calling function should not deal with the owner of the underlying characters (such as deleting it, changing its value, or freeing its memory).
- It can deal with the `nullptr` as value.

Note that ambiguity errors are possible, if you overloads functions for both `std::string` and `std::string_view`:

```
void foo(const std::string&);
void foo(std::string_view);

foo("hello");   // ERROR: ambiguous
```

19.5.1 Using String Views to Initialize Strings

It looks like a simple and useful application of a string view would be to declare it as parameter type when initializing a string. **But beware!**

Consider the "good old way" to initialize a string member:

```
class Person {
  std::string name;
 public:
  Person (const std::string& n) : name(n) {
  }
  ...
```

```
};
```

This constructor has its drawbacks. Initializing a person with a string literal creates one unnecessary copy, which might cause an unnecessary request for heap memory. For example:

```
Person p("Aprettylong NonSSO Name");
```

first calls the `std::string` constructor to create the temporary parameter `n`, because a reference of type `std::string` is requested. If the string is long or no short string optimization is enabled[4] this means that for the string value heap memory is allocated. Even with move semantics the temporary string is then copied to initialize the member `name`, which means that again memory is allocated. You can avoid this overhead only by adding more constructor overloads or introducing a template constructor, which might cause other problems.

If instead we are using a string view, the performance is better:

```
class Person {
  std::string name;
 public:
  Person (std::string_view n) : name(n) {
  }
  ...
};
```

Now a temporary string view `n` gets created, which does not allocate memory at all, because the string view only refers to the characters of the string literal. Only the initialization of `name` allocates once the memory for the member `name`.

However, there is a problem: If you pass a temporary string or a string marked with `std::move()` the string is converted to type a string view (which is cheap) and then the string view is used to allocate the memory for the new string (which is expensive). In other words: The use of string view disables move semantics unless you provide an additional overload for it.

There is still a clear recommendation for how to initialize objects with string members: Take the string by value and move:

```
class Person {
  std::string name;
 public:
  Person (std::string n) : name(std::move(n)) {
  }
  ...
};
```

We anyway have to create a string. So, creating it as soon as possible allows us to benefit from all possible optimizations the moment we pass the argument. And when we have it, we only move, which is a cheap operation.

If we initialize the string by a helper function returning a temporary string:

[4] With the often implemented short string optimization (SSO) strings only allocate memory if they have more than say 15 characters.

```
std::string newName()
{
  ...
  return std::string{...};
}

Person p{newName()};
```

the mandatory copy elision will defer the materialization of a new string until the value is passed to the constructor. There we have a string named `n` so that we have an object with a location (a *glvalue*). The value of this object is then moved to initialize the member `name`.

This example again demonstrates:

- String views are not a better interface for taking strings.
- In fact, string views should only be used in call chains, where they never have to be used as strings.

19.5.2 Using String Views instead of Strings

There are other replacement of strings by string views possible. But again, beware.

For example, instead of the following code:

```
// convert time point (with prefix) to string:
std::string toString (const std::string& prefix,
                      const std::chrono::system_clock::time_point& tp)
{
  // convert to calendar time:
  auto rawtime = std::chrono::system_clock::to_time_t(tp);
  std::string ts = std::ctime(&rawtime);  // NOTE: not thread safe

  ts.resize(ts.size()-1);  // skip trailing newline

  return prefix + ts;
}
```

you could implement the following:

```
std::string toString (std::string_view prefix,
                      const std::chrono::system_clock::time_point& tp)
{
  auto rawtime = std::chrono::system_clock::to_time_t(tp);
  std::string_view ts = std::ctime(&rawtime);  // NOTE: not thread safe

  ts.remove_suffix(1);      // skip trailing newline

  return std::string(prefix) + ts;  // unfortunately no operator + yet
}
```

Beside the optimization to get the passed string value for the prefix as a `std::string_view` by value, we can also use a string view here internally. **But only** because the C-string returned by `ctime()` is valid for a while (it is valid until the next call of `ctime()` or `asctime()`). Note that we can remove the trailing newline from the string, but that we can't concatenate both string views by simply calling `operator+`. Instead, we have to convert one of the operands to a `std::string` (which unfortunately unnecessarily might allocate additional memory).

19.6 Afternotes

The first string class with reference semantics was proposed by Jeffrey Yasskin in https://wg21.link/n3334 (using the name `string_ref`). This class was adopted as part of the Library Fundamentals TS as proposed by Jeffrey Yasskin in https://wg21.link/n3921.

The class was adopted with other components for C++17 as proposed by Beman Dawes and Alisdair Meredith in https://wg21.link/p0220r1. Some modifications for better integration were added by Marshall Clow in https://wg21.link/p0254r2 and in https://wg21.link/p0403r1 and by Nicolai Josuttis in https://wg21.link/p0392r0.

Additional fixes by Daniel Krügler are in https://wg21.link/lwg2946 (which will probably come as a defect against C++17).

This page is intentionally left blank

Chapter 20

The Filesystem Library

With C++17 the Boost.filesystem library was finally adopted as a C++ standard library. By doing this, the library was adjusted to new language features, made more consistent with other parts of the library, cleaned-up, and extended to provided some missing pieces (such as operations to compute a relative path between filesystem paths).

20.1 Basic Examples

Let's start with some basic examples:

20.1.1 Print Attributes of a Passed Filesystem Path

The following program allows us to use a passed string as a filesystem path to print some aspects of the patch according to its file type:

filesystem/checkpath.cpp

```
#include <iostream>
#include <filesystem>

int main(int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <path> \n";
    return EXIT_FAILURE;
  }

  std::filesystem::path p{argv[1]};  // p represents a filesystem path (might not exist)
  if (!exists(p)) {                  // does path p actually exist?
    std::cout << "path " << p << " does not exist\n";
```

```
  }
  else {
    if (is_regular_file(p)) {          // is path p a regular file?
      std::cout << p << " exists with " << file_size(p) << " bytes\n";
    }
    else if (is_directory(p)) {        // is path p a directory?
      std::cout << p << " is a directory containing:\n";
      for (auto& e : std::filesystem::directory_iterator{p}) {
        std::cout << "    " << e.path() << '\n';
      }
    }
    else {
      std::cout << p << " exists, but is no regular file or directory\n";
    }
  }
}
```

We first check, whether the passed filesystem path represents an existing file:

```
std::filesystem::path p{argv[1]};   // p represents a filesystem path (might not exist)
if (!exists(p)) {                   // does path p actually exist?
  ...
}
```

If yes, we perform the following checks:

- If it is a regular file, we print its size:

  ```
  if (is_regular_file(p)) {          // is path p a regular file?
    std::cout << p << " exists with " << file_size(p) << " bytes\n";
  }
  ```

  Calling this program as follows:

  ```
  checkpath checkpath.cpp
  ```

  will output something like:

  ```
  "checkpath.cpp" exists with 907 bytes
  ```

  Note that the output operator for paths automatically writes the path name quoted (within double quotes and backslashes are escaped by another backslash, which is an issue for Windows paths).

- If it is a directory, we iterate over the files in the directory and print the paths:

  ```
  if (is_directory(p)) {        // is path p a directory?
    std::cout << p << " is a directory containing:\n";
    for (auto& e : std::filesystem::directory_iterator(p)) {
      std::cout << "    " << e.path() << '\n';
    }
  }
  ```

Here we use a `directory_iterator`, which provides `begin()` and `end()` in a way that we can iterate over `directory_entry` elements using a range-based for loop. In this case we use the `directory_entry` member function `path()`, which yields the filesystem path of the entry.

Calling this program as follows:

```
checkpath .
```

will output something like:

```
"." is a directory containing:
    "./checkpath.cpp"
    "./checkpath.exe"
```

Path Handling under Windows

The fact that paths by default are written quoted is an issue under Windows, because the usual directory separator backslash is then always escaped and written twice. Thus, calling this program under Windows as follows:

```
checkpath C:\
```

will output something like:

```
"C:\\" is a directory containing:
    ...
    "C:\\Users"
    "C:\\Windows"
```

Writing paths quoted ensures that the written filenames can be read into a program so that you get back the original filenames. However, for standard output, this is usually not acceptable.

For this reason, a portable version also running well under Windows should avoid writing paths quoted to standard output by using the member function `string()`:

filesystem/checkpath2.cpp

```
#include <iostream>
#include <filesystem>

int main(int argc, char* argv[])
{
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <path> \n";
    return EXIT_FAILURE;
  }

  std::filesystem::path p{argv[1]};  // p represents a filesystem path (might not exist)
  if (!exists(p)) {                  // does path p actually exist?
    std::cout << "path " << p.string() << " does not exist\n";
```

```
  }
  else {
    if (is_regular_file(p)) {        // is path p a regular file?
      std::cout << p.string() << " exists with "
                << file_size(p) << " bytes\n";
    }
    else if (is_directory(p)) {      // is path p a directory?
      std::cout << '"' << p.string() << "\" is a directory containing:\n";
      for (auto& e : std::filesystem::directory_iterator{p}) {
        std::cout << "    " << e.path().string() << '\n';
      }
    }
    else {
      std::cout << p.string()
                << " exists, but is no regular file or directory\n";
    }
  }
}
```

Now, calling this program under Windows as follows:

```
checkpath C:\
```

will output something like:

```
"C:\" is a directory containing:
    ...
    "C:\Users"
    "C:\Windows"
```

Other conversions are provided to use the generic string format or convert the string to the native encoding.

20.1.2 Create Different Types of Files

The following program tries to create different kind of files in the subdirectory `tmp`:

filesystem/createfiles.cpp

```
#include <iostream>
#include <fstream>
#include <filesystem>

int main ()
{
  namespace fs = std::filesystem;
```

```
  try {
    // create tmp/test:
    fs::path tmpDir{"tmp"};
    fs::path testPath{tmpDir / "test"};

    // create directories if they don't exist yet:
    if (!create_directories(testPath)) {
      std::cout << "test directory " << testPath << " already exists\n";
    }

    // create data file tmp/test/data.txt:
    testPath /= "data.txt";
    std::ofstream dataFile{testPath};
    if (!dataFile) {
      std::cerr << "OOPS, can't open " << testPath << '\n';
    }
    dataFile << "The answer is 42\n";

    // create local symbolic link to tmp/test:
    fs::path symlink{"testdir"};
    if (!is_symlink(symlink)) {
      create_directory_symlink(testPath, symlink);
    }

    // recursively list all files (also following symlinks)
    std::cout << "all files:\n";
    auto iterOpts{fs::directory_options::follow_directory_symlink};
    for (const auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
      std::cout << "  " << e.path().lexically_normal().string() << '\n';
    }
  }
  catch (fs::filesystem_error& e) {
    std::cerr << "exception: " << e.what() << '\n';
    std::cerr << "           path1: " << e.path1() << '\n';
  }
}
```

Let's also go through this program step by step.

Namespace `fs`

First, we do something very common: Define `fs` as shortcut for namespace `std::filesystem`:

```
namespace fs = std::filesystem;
```

Using this namespace we initialize two paths, a basic directory for temporary files and subdirectory `test` inside using operator `/`:

```
fs::path tmpDir{"tmp"};
fs::path testPath{tmpDir / "test"};
```

Creating Directories

Then we try to create the subdirectory:

```
if (!create_directories(testPath)) {
  std::cout << "test directory " << testPath << " already exists\n";
}
```

By using `create_directories()` we create all missing directories of the whole path. The function returns `true` if any directory was created. Thus, it is not an error to have this call even if the directory already exists (any other problem is an error and would raise an exception).

Note also that it is not an error, if a file with this name but a different type (e.g., a regular file) already exists. Because also the situation in a multi-user/multi-process operating system can change at any time, it usually is the best approach to try to create the directory and handle exceptions accordingly.

However, sometimes the trial to create a file might work but in the wrong way. For example, if you want to create a file at a specific location and there already is a symbolic link, the file gets created or is overwritten at a maybe unexpected location. In that case, you should check for the existence of a file, which is a bit more complicated than you might think at first.

Creating Regular Files

Then we create a new file `/tmp/test/data.txt` with some contents:

```
testPath /= "data.txt";
std::ofstream dataFile(testPath);
if (!dataFile) {
  std::cerr << "OOPS, can't open " << testPath << '\n';
}
dataFile << "The answer is 42\n";
```

Here we use operator `/=` to extend a path, which we can the pass as argument to the constructor of a file stream. As you can see, the creation of a regular file still only can be done with the existing I/O streams library. However, a new overload for the constructors is provided to be able to directly pass a filesystem path.

Creating Symbolic Links

The next statement tries to create a symbolic link `testdir` in our current directory referring to directory `tmp/test`:

```
fs::path symlink{"testdir"};
if (!is_symlink(symlink)) {
  create_directory_symlink(testPath, symlink);
}
```

In most cases calling

```
create_symlink(testPath, symlink);
```

also works, but some operating systems have special handling for symbolic links to directories to that `create_directory_symlink()` is more portable in this case. Note that the first argument defines the path to the location relatively to the created link. Thus, if you want to create a symbolic link from `"dir/grey.txt"` to `"dir/gray.txt"`, you have to program:

```
std::filesystem::create_symlink("gray.txt", "dir/grey.txt");
```

Recursive Directory Iteration

Finally, we recursively list our current directory:

```
auto iterOpts = fs::directory_options::follow_directory_symlink;
for (auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
  std::cout << "    " << e.path().lexically_normal().string() << '\n';
}
```

Because we use a recursive directory iterator and pass the option to also follow symbolic links, `follow_directory_symlink`, we should get an output like the following on POSIX-based systems:

```
all files:
  ...
  testdir
  testdir/data.txt
  tmp
  tmp/test
  tmp/test/data.txt
```

and an output like the following on Windows systems:

```
all files:
  ...
  testdir
  testdir\data.txt
  tmp
  tmp\test
  tmp\test\data.txt
```

Note that the path of the directory entries would contain the directory the iterator was initialized with as a prefix. Thus, printing just the paths inside the loop:

```
auto iterOpts = fs::directory_options::follow_directory_symlink;
for (auto& e : fs::recursive_directory_iterator(".", iterOpts)) {
  std::cout << "    " << e.path() << '\n';
}
```

would output under POSIX-based systems:

```
all files:
  ...
  "./testdir"
  "./testdir/data.txt"
  "./tmp"
  "./tmp/test"
  "./tmp/test/data.txt"
```

And on Windows the output would be:

```
all files:
  ...
  ".\\testdir"
  ".\\testdir\\data.txt"
  ".\\tmp"
  ".\\tmp\\test"
  ".\\tmp\\test\\data.txt"
```

By calling `lexically_normal()` we yield the normalized path, which does remove the leading dot for the current directory. And as written before, by calling `string()` we avoid that each path is written quoted, which is OK for POSIX-based systems (just having the name in double quotes), but looks very surprising on Windows systems (because each backslash is escaped by another backslash).

Error Handling

Depending on the platform and permissions a couple of things can go wrong here. For those not covered by return values, we catch the corresponding exception and print the general message and the first path in it:

```
try {
  ...
}
catch (fs::filesystem_error& e) {
  std::cerr << "exception: " << e.what() << '\n';
  std::cerr << "          path1: " << e.path1() << '\n';
}
```

For example, if we can't create the directory, a message such as this might get printed:

```
exception: filesystem error: cannot create directory: [tmp/test]
           path1: "tmp/test"
```

20.1.3 Switch Over Filesystem Types

The following code demonstrates how implement different behavior on different filesystem types for a given file name:

filesystem/switchinit.cpp

```
#include <string>
#include <iostream>
#include <filesystem>
namespace std {
    using namespace std::experimental;
}

void checkFilename(const std::string& name)
{
  using namespace std::filesystem;

  switch (path p(name); status(p).type()) {

   case file_type::not_found:
     std::cout << p << " not found\n";
     break;

   case file_type::directory:
     std::cout << p << ":\n";
     for (const auto& entry : std::filesystem::directory_iterator(p)) {
        std::cout << "- " << entry.path() << '\n';
     }
     break;

   default:
     std::cout << p << " exists\n";
     break;
  }
}

int main()
{
  for (auto name : {".", "switchinit.cpp", "nofile"} ) {
    checkFilename(name);
```

```
    std::cout << '\n';
  }
}
```

Here we use a switch with initialization to get access to the file type:

```
switch (path p(name); status(p).type()) {
  ...
}
```

The expression `status(p).type()` creates a `file_status`, for which `type()` creates a `file_type`. This is intentionally provided in multiple steps so that we don't have to pay the price of operating system calls if we are not interested in status information.

20.1.4 Dealing with Filesystems Using Parallel Algorithms

See dirsize.cpp for another example using parallel algorithms to accumulate the size of all regular files in a directory tree.

20.2 Principles and Terminology

Before discussing the details of the filesystem library we have to introduce a couple of design principles and terminology. This is necessary because the standard covers different operating systems and maps them to a common API.

20.2.1 General Portability Disclaimer

The C++ standard does not only standardize what all possible operating systems have in common for their file systems. In many cases, it follows POSIX standards and the C++ standard requests to follow POSIX as close as possible. As long as it is reasonable the behavior should still be there but with some limitations. If no reasonable behavior is possible, an implementation shall report an error. Possible examples for such errors are:

- Characters are used for filenames that are not supported
- File system elements are created that are not supported (e.g., symbolic links)

Still the differences of specific file systems may matter:

- Case sensitivity:
  `"hello.txt"` and `"Hello.txt"` and `"hello.TXT"` might refer to the same (Windows) or three different files (POSIX-based).
- Absolute versus relative paths:
  On some systems `"/bin"` is an absolute path (POSIX-based), while on others it is not (Windows).

20.2.2 Namespace

The filesystem library has its own sub-namespace `filesystem` inside `std`. It is a pretty common convention to introduce the shortcut `fs` for it:

```
namespace fs = std::filesystem;
```

This, for example, enables to use `fs::current_path()` instead of `std::filesystem::current_path()`.

Further code examples of this chapter will often use `fs` as the corresponding shortcut.

Note that not qualifying filesystem calls sometimes results in unintended behavior.

20.2.3 Paths

The key element of the filesystem library is a `path`. It is a name that represents the (potential) location of a file within a filesystem. It consists of an optional root name, an optional root directory, and a sequence of filenames separated by directory separators. The path can be relative (so that the file location depends on the current working directory) or absolute.

Different formats are possible:

- A generic format, which is portable
- A native format, which is specific to the underlying file system

On POSIX-based operating systems there is not difference between the generic and the native format. On Windows the generic format `/tmp/test.txt` is a valid native format besides `\tmp\test.txt`, which is also supported (thus, `/tmp/test.txt` and `\tmp\test.txt` are two native versions of the same path). On OpenVMS the corresponding native format might be `[tmp]test.txt`.

Special filenames exist:

- `"."` represents the current directory
- `".."` represents the parent directory

The generic path format is as follows:

[*rootname*] [*rootdir*] [*relativepath*]

where:

- The optional root name is implementation specific (e.g., it can be `//host` on POSIX systems and `C:` on Windows systems)
- The optional root directory is a directory separator
- The relative path is a sequence of file names separated by directory separators

By definition, a directory separators consists of one or multiple `'/'` or implementation-specific preferred directory separator.

Examples for portable generic paths are:

```
//host1/bin/hello.txt
.
tmp/
/a/b//../c
```

Note that the last path refers to the same location as `/a/c` and is absolute on POSIX systems but relative on Windows systems (because the drive/partition is missing).

On the other hand a path such as `C:/bin` is an absolute path on Windows systems (the root directory `"bin"` on the `"C"` drive/partition) but a relative path on POSIX (the subdirectory `"bin"` in the directory `"C:"`).

On Windows systems the backslash is the implementation specific directory separator so that the paths above can there *also* be written by using the backslash as the preferred directory separator:

```
\\host1\bin\hello.txt
.
tmp\
\a\b\..\c
```

The filesystem library provides function to convert paths between the native and generic format.

A `path` might be empty. This mean that there is no path defined. This is *not* necessarily the same as `"."`. What it means depends on the context.

20.2.4 Normalization

A path might be or can get normalized. In a normalized path:

- Filenames are separated only by a single preferred directory separator.
- The filename `"."` is not used unless the whole path is nothing but `"."` (representing the current directory).
- The filename does not contain `".."` filenames (we don't go down and then up again) unless they are at the beginning of a relative path.
- The path only ends with a directory separator if the trailing filename is a directory with a name other than `"."` or `".."`.

Note that normalization still means that a filename ending with a directory separator is different from a filename not ending with a separator. The reason is that on some operating systems the behavior differs when it is known that the path is a directory (e.g., with a trailing separator symbolic links might get resolved).

Table *Effect of Path Normalization* lists some examples for normalization on POSIX and Windows systems. Note again that on POSIX system `C:bar` and `C:` are just filenames and have no special meaning to specify a partition as on Windows.

Note that the path `C:\bar\..` remains the same when being normalized on a POSIX-based system. The reason is that there the backslash is no directory separator so that the whole path is just *one* filename having a colon, two backslashes, and two dots as part of its name.

The filesystem provides function for both lexical normalization (not taking the filesystem into account) and filesystem-dependent normalization.

Path	POSIX normalized	Windows normalized
`foo/.///bar/../`	`foo/`	`foo\`
`//host/../foo.txt`	`//host/foo.txt`	`\\host\foo.txt`
`./f/../.f/`	`.f/`	`.f\`
`C:bar/../`	`.`	`C:`
`C:/bar/..`	`C:/`	`C:\`
`C:\bar\..`	`C:\bar\..`	`C:\`
`/./../data.txt`	`/data.txt`	`\data.txt`
`././`	`.`	`.`

Table 20.1. Effect of Path Normalization

20.2.5 Member versus Free-Standing Functions

The filesystem library provides several functions, which can be both member and free-standing functions. The general approach is:

- **Member functions** are **cheap**. The reason is that they are pure lexical operations that do not take the actual filesystem into account, so that no operating systems calls are necessary.

  For example:

  ```
  mypath.is_absolute()              // check whether path is absolute or relative
  ```

- **Free-standing functions** are **expensive**, because they usually take the actual filesystem into account, so that no operating systems calls are necessary.

  For example:

  ```
  equivalent(path1, path2);         // true if both paths refer to the same file
  ```

Sometimes, the filesystem library even provides the same functionality operating both lexically and by taking the actual filesystem into account:

```
std::filesystem::path fromP, toP;
...
toP.lexically_relative(fromP);    // yield lexical path from fromP to toP
relative(toP, fromP);             // yield actual path from fromP to toP
```

Thanks to *argument dependent lookup* (ADL) usually you don't have to specify the full namespace `std::filesystem`, when calling free-standing filesystem functions and an argument has a filesystem specific type. Only when implicit conversions from other types are used, you have to qualify the call. For example:

```
create_directory(std::filesystem::path{"tmpdir"});  // OK
remove(std::filesystem::path{"tmpdir"});            // OK
std::filesystem::create_directory("tmpdir");        // OK
std::filesystem::remove("tmpdir");                  // OK
create_directory("tmpdir");           // ERROR
remove("tmpdir");                     // OOPS: calls C function remove()
```

Note that the last call usually compiles, but finds the C function `remove()` which also removes a specified file, but does not remove empty directories under Windows.

20.2.6 Error Handling

Filesystems are a source of errors. You have to take into account that necessary files might not exists, file operations are not allowed, or operations violate resource limits. In addition, while program runs other processes might create, modify, or remove files so that even checks in ahead are no guarantee for no errors.

In many situations it is not useful to handle all possible errors after each and every filesystem call. Nevertheless, sometimes it is important to locally react on a failed filesystem operation. Therefore, the filesystem library uses a mixed approach when dealing with the filesystem:

- By default, filesystem errors are handled as exceptions.
- But you can handle specific errors locally if you have or want to.

This is realized by filesystem operations usually having two overloads for each operation:

1. By default (without an additional error handling argument) the operations throw a `filesystem_error` exceptions on errors.
2. By passing an additional out parameter, you can instead get an error code on error.

Note that in the latter case, you might still have special return values, signaling a specific error that is not handled as an exception.

Using `filesystem_error` Exceptions

For example, you can try to create a directory as follows:

```
if (!create_directory(p)) {                 // exception on error (unless path exists)
  std::cout << p << " already exists\n";    // path exists
}
```

Here, no error code argument is passed so that errors usually raise an exception. Note however that the special case that the path already exists (whether or not it is a directory doesn't matter), is handled by returning `false`. Thus, an exception is raised due to other problems such as missing rights to create the directory, an invalid path `p`, or a violations of file system resources (such as exceeding path length limits).

Code like this should directly or indirectly be enclosed in a `try-catch` clause, which handles an exception of type `std::filesystem::filesystem_error`:

```
try {
  ...
  if (!create_directory(p)) {               // exception on error (unless path exists)
    std::cout << p << " already exists\n"; // path exists
  }
  ...
}
catch (const std::filesystem::filesystem_error& e) {  // derived from std::exception
```

```
    std::cout << "EXCEPTION: " << e.what() << '\n';
    std::cout << "     path: " << e.path1() << '\n';
  }
```

As you can see, filesystem exceptions provide the usual standard exception API to yield an implementation-specific error message with `what()`. However, it also provides `path1()` if a path is involved and even `path2()` if a second path is involved.

Using `error_code` Arguments

The other way to call the function to create a directory is as follows:

```
  std::error_code ec;
  create_directory(p, ec);                    // set error code on error
  if (ec) {                                   // if error code set (due to error)
    std::cout << "ERROR: " << ec.message() << "\n";
  }
```

Afterwards, we can also check against specific error codes:

```
  if (ec == std::errc::read_only_file_system) {     // if specific error code set
    std::cout << "ERROR: " << p << " is read-only\n";
  }
```

Note that in this case we still can check the return value of `create_directory()`:

```
  std::error_code ec;
  if (!create_directory(p, ec)) {                          // set error code on error
    std::cout << "can't create directory " << p << "\n"; // any error occurred
    std::cout << "error: " << ec.message() << "\n";
  }
```

However, not all filesystem operations provide this ability (because they return some value in the normal case).

Type `error_code` was introduced with C++11 including a list of portable error conditions such as `std::errc::read_only_filesystem`. On POSIX system these map to `errno` values.

20.2.7 File Types

Different operating systems support different file types. The standard filesystem library takes this into account. In principle, there is an enumeration type `file_type`, which is standardized to have the following values:

```
  namespace std::filesystem {
    enum class file_type {
      regular, directory, symlink,
      block, character, fifo, socket,
      ...
      none, not_found, unknown,
```

```
    };
}
```

Table `file_type` *Values* lists the meaning of these values.

Value	Meaning
`regular`	Regular file
`directory`	Directory file
`symlink`	Symbolic link file
`character`	Character special file
`block`	Block special file
`fifo`	FIFO or pipe file
`socket`	Socket file
...	additional implementation-defined file type
`none`	The type of the file is not known (yet)
`unknown`	The file exists but the type could not be determined
`not_found`	Pseudo-type indicating the file was not found

Table 20.2. File Type Values

Platforms might provide additional file type values, which however is not portable. For example:

- Windows provides the file type value `junction`, which is used for *NTFS junctions* (also called *soft links*) of the NTFS file system. They are used as links to directories located on different local volumes on the same computer.

Beside regular files and directories the most common other type is a symbolic link, which is a type for file that refer to another filesystem location. At that location there might be a file or there might not. Note that some operating systems and/or file systems (e.g., the FAT file system) don't support symbolic links at all. Some operating systems support them only for regular files. Note that on Windows you need special permissions to create symbolic links, which you for example, can do with the `mklink` command.

Character-special files, block-special files, FIFOs, and sockets come from the UNIX filesystem. Currently, all four types are not used with Visual C++.[1]

As you can see, special values exist for cases when the file doesn't exists or its file type is not known or detectable.

In the remainder of this chapter I use two general categories representing a couple of file types:

- *Other files*: Files with any file type other than regular file, directory, and symbolic link. The library function `is_other()` matches this term.
- *Special files*: Files with any of the following file types: Character-special files, block-special files, FIFOs, and sockets.

The *special* file types plus the implementation-defined file types together form the *other* file types.

[1] Windows pipes behave differently and are not categorized as `fifo`.

20.3 Path Operations

To deal with filesystems there are plenty of operations you can call. A key type to deal with the filesystem is type `std::filesystem::path`, which can be used as an absolute or relative path of a file that might or might not exist (yet).

You can create path, inspect them, modify them, and compare them. Because these operations usually do not take the filesystem into account (care for existing files, symbolic links, etc.), they are cheap to call. As a consequence, they are usually member functions (if they are no constructors or operators).

20.3.1 Path Creation

Table *Path Creation* lists the ways to create a new path object.

Call	Effect
`path(`*string*`)`	creates path from a string
`path(`*beg*`,`*end*`)`	creates path from a range
`u8path(`*u8string*`)`	creates path from a UTF-8 string
`current_path()`	yields the path of the current working directory
`temp_directory_path()`	yields the path for temporary files

Table 20.3. Path Creation

Note that both `current_path()` and `temp_directory_path()` are more expensive operations because they are based on operating system calls. By passing an argument `current_path()` can also be used to modify the current working directory.

With `u8path()` you can create portable paths using all UTF-8 characters. For example:

```
std::filesystem::path{u8path(u8"K\u00F6ln");   // "Köln" (Cologne native)
...

// create directory from returned UTF-8 string:
std::string utf8String = readUTF8String(...);
create_directory(std::filesystem::u8path(utf8String));
```

20.3.2 Path Inspection

Table *Path Inspection* lists the functions you can call to inspect a path `p`. Note that these operations do not take the filesystem into account and are therefore member functions of a path.

Each path is either absolute or relative. It is relative if it has no root directory (a root name is possible; e.g., `C:hello.txt` is a relative path under Windows).

The `has_...()` functions check whether the corresponding functions without `has_` yield an empty path.

Call	Effect
`p.empty()`	yields whether a path is empty
`p.is_absolute()`	yields whether a path is empty
`p.is_relative()`	yields whether a path is empty
`p.has_filename()`	yields whether a path neither a directory nor a root name
`p.has_stem()`	same as `has_filename()` (as any filename has a stem)
`p.has_extension()`	yields whether a path has an extension
`p.has_root_name()`	yields whether a path has a root name
`p.has_root_directory()`	yields whether a path has a root directory
`p.has_root_path()`	yields whether a path has a root name or a root directory
`p.has_parent_path()`	yields whether a path has a parent path
`p.has_relative_path()`	yields whether a path does not only consist of root elements
`p.filename()`	yields the filename (or the empty path)
`p.stem()`	yields the filename without extension (or the empty path)
`p.extension()`	yields the extension (or the empty path)
`p.root_name()`	yields the root name (or the empty path)
`p.root_directory()`	yields the root directory (or the empty path)
`p.root_path()`	yields the root elements (or the empty path)
`p.parent_path()`	yields the parent path (or the empty path)
`p.relative_path()`	yields the path without root elements (or the empty path)
`p.begin()`	begin of a path iteration
`p.end()`	end of a path iteration

Table 20.4. Path Inspection

Note the following:

- There is always a parent path if a root element or a directory separator is part of the path. If the path consists only of root elements (i.e., the relative path is empty), `parent_path()` yields the same path. That is, example, the parent path of `"/"` is `"/"`. Only the parent path of a pure filename such as `"hello.txt` is empty.
- If a path has a filename it also always has a stem.[2]
- The empty path is a relative path (yielding `false` or an empty path for all other operations beside `is_empty()` and `is_relative()`).

The result of these operations might depend on the operation system. For example, the path `C:/hello.txt`

- on Unix systems
  - is relative
  - has no root elements (neither a root name nor a root directory), because `C:` is a filename.

[2] This has changed with C++17 because before a filename could consist of a pure extension.

  - has the parent path `C:`
  - has the relative path `C:/hello.txt`
- on Windows systems
  - is absolute
  - has the root name `C:` and the root directory `/`
  - has no parent path
  - has the relative path `hello.txt`

Path Iteration

You can iterate over a path, which yields the elements of the path: the root name if any, the root directory if any, and all the filenames. If the path ends with a directory separator, the last element is an empty filename.[3]

The iterator is a bidirectional iterator so that you can use `--`. The values the iterators refer to are of type `path` again. However, two iterators iterating over the same path might ***not*** refer to the same `path` object even if they refer to the same element.

For example:

```
void printPath(const std::filesystem::path& p)
{
   std::cout << "path elements of " << p.string << ":\n";
   for (auto pos = p.begin(); pos != p.end(); ++pos) {
      std::filesystem::path elem = *pos;
      std::cout << "  " << elem;
   }
   std::cout << '\n';
}
```

If this function is called as follows:

```
printPath("../sub/file.txt");
printPath("/usr/tmp/test/dir/");
printPath("C:\\usr\\tmp\\test\\dir\\");
```

the output on a POSIX-based system will be:

```
path elements of "../sub/file.txt":
  ".."  "sub"  "file.txt"
path elements of "/usr/tmp/test/dir/":
  "/"  "usr"  "tmp"  "test"  "dir"  ""
path elements of "C:\\usr\\tmp\\test\\dir\\":
  "C:\\usr\\tmp\\test\\dir\\"
```

[3] Before C++17, the filesystem library implementations used `.` to signal a trailing directory separator. This has changed to be able to distinguish a path ending with a separator from a path ending with a dot after the separator.

Note that the last path is just one filename, because neither `C:` is a valid root name nor is the backslash a valid directory separator under POSIX-based systems.

The output on a Windows system will be:

```
path elements of "../sub/file.txt":
  ".."  "sub"  "file.txt"
path elements of "/usr/tmp/test/dir/":
  "/"  "usr"  "tmp"  "test"  "dir"  ""
path elements of "C:\\usr\\tmp\\test\\dir\\":
  "C:"  "\\"  "usr"  "tmp"  "test"  "dir"  ""
```

To check whether a path `p` ends with a directory separator you can implement:

```
if (!p.empty() && (--p.end())->empty()) {
   std::cout << p << " has a trailing separator\n";
}
```

20.3.3 Path I/O and Conversions

Table *Path I/O and Conversions* lists the operations to read or write and to yield a converted path. These function do not take the actual filesystem into account. If you have to deal with paths where symbolic links matter, you might want to use the filesystem-dependent path conversions.

Call	Effect
`strm << p`	write the value of a path as quoted string
`strm >> p`	reads the value of a path as quoted string
`p.string()`	yields the path as a `std::string`
`p.wstring()`	yields the path as a `std::wstring`
`p.u8string()`	yields the path as a UTF-8 string of type `std::u8string`
`p.u16string()`	yields the path as a UTF-16 string of type `std::u16string`
`p.u32string()`	yields the path as a UTF-32 string of type `std::u32string`
`p.string<...>()`	yields the path as a `std::basic_string<...>`
`p.lexically_normal()`	yields `p` as normalized path
`p.lexically_relative(p2)`	yields the path from `p2` to `p` (empty path if none)
`p.lexically_proximate(p2)`	yields the path from `p2` to `p` (`p` if none)

Table 20.5. Path I/O and Conversions

The `lexically_...()` functions return a new path, while the other conversion functions yield a corresponding string type. None of these functions modifies the path they are called for.

For example, the following code:

```
std::filesystem::path p{"/dir/./sub//sub1/../sub2"};
std::cout <<  "path:                " << p << '\n';
std::cout <<  "string():            " << p.string() << '\n';
```

```
std::wcout << "wstring():          " << p.wstring() << '\n';
std::cout <<  "lexically_normal(): " << p.lexically_normal() << '\n';
```

has the same output for the first three rows:

```
path:               "/dir/./sub//sub1/../sub2"
string():           /dir/./sub//sub1/../sub2
wstring():          /dir/./sub//sub1/../sub2
```

but the output for the last row depends on the directory separator. On POSIX-based systems it is:

```
lexically_normal(): "/dir/sub/sub2"
```

while on Windows it is:

```
lexically_normal(): "\\dir\\sub\\sub2"
```

Path I/O

First, note that the I/O operators write and read paths as quoted strings. You have to convert them to a string to write them without quotes:

```
std::filesystem::path file{"test.txt"}
std::cout << file << '\n';            // writes: "test.txt"
std::cout << file.string() << '\n';   // writes: test.txt
```

On Windows this has even worse effects. The following code:

```
std::filesystem::path tmp{"C:\\Windows\\Temp"};
std::cout << tmp << '\n';
std::cout << tmp.string() << '\n';
std::cout << '"' << tmp.string() << "\"\n";
```

has the following output:

```
"C:\\Windows\\Temp"
C:\Windows\Temp
"C:\Windows\Temp"
```

Note that reading filenames supports both forms (quoted with a leading " and non-quoted). Thus, all printed forms will be read correctly back using the standard input operator for paths:

```
std::filesystem::path tmp;
std::cin >> tmp;  // reads quoted and non-quoted paths correctly
```

Normalization

Normalization might have more surprising outcomes when you deal with portable code. For example:

```
std::filesystem::path p2{"//dir\\subdir/subsubdir\\/./\\"};
std::cout <<  "p2:                 " << p2 << '\n';
std::cout <<  "lexically_normal(): " << p2.lexically_normal() << '\n';
```

has the following probably expected output on Windows systems:

```
p2:                  "//host\\dir/sub\\/./\\"
lexically_normal(): "\\\\host\\dir\\sub\\"
```

However, on POSIX-based systems the output becomes:

```
p2:                  "//host\\dir/sub\\/./\\"
lexically_normal(): "/host\\dir/sub\\/\\"
```

The reason is that for POSIX-based system the backslash is no directory separator but also not a valid character for a rootname, so that we have an absolute path with the three filenames `host\dir`, `sub\`, and `\`. On POSIX-based systems there is no way to detect the backslash to be a possible directory separator (neither `generic_string()` nor `make_preferred()` will help in this case). So, for portable code, you should always use the generic path format when dealing with paths.

Nevertheless, it is a good approach to use `lexically_normal()` to remove the leading dot when iterating over the current directory.

Relative Path

Both `lexically_relative()` and `lexically_proximate()` can be called to compute the relative path between to paths. The only difference is the behavior if there is no path, which only can happen if one path is relative and the other is absolute or the root names differ. In that case:

- `p.lexically_relative(p2)` yields the empty path if there is no relative path from `p2` to `p`.
- `p.lexically_proximate(p2)` yields `p` if there is no relative path from `p2` to `p`.

As both operations operate lexically, the actual filesystem (with possible symbolic links) and the `current_path()` are not taken into account. If both paths are equal, the relative path is `.`. For example:

```
fs::path{"/a/d"}.lexically_relative("/a/b/c")     //"../../d"
fs::path{"/a/b/c"}.lexically_relative("/a/d")     //"../b/c"
fs::path{"/a/b"}.lexically_relative("/a/b")       //"."
fs::path{"/a/b"}.lexically_relative("/a/b/")      //"."
fs::path{"/a/b"}.lexically_relative("/a/b\\")     //"."
fs::path{"/a/b"}.lexically_relative("/a/d/../c")  //"../b
fs::path{"a/d/../b"}.lexically_relative("a/c")    //"../d/../b"
fs::path{"a//d/..//b"}.lexically_relative("a/c")  //"../d/../b"
```

On Windows systems, we have:

```
fs::path{"C:/a/b"}.lexically_relative("c:/c/d") ;   //""
fs::path{"C:/a/b"}.lexically_relative("D:/c/d") ;   //""
fs::path{"C:/a/b"}.lexically_proximate("D:/c/d") ;  //"C:/a/b"
```

Conversions to Strings

With `u8string()` you can use the path as UTF-8 string, which is nowadays the common format for stored data. For example:

```
// store paths as UTF-8 string:
std::vector<std::string> utf8paths;  // std::u8string with C++20
for (const auto& entry : fs::directory_iterator(p)) {
   utf8paths.push_back(entry.path().u8string());
}
```

Note that the return value of `u8string()` will probably change from `std::string` to `std::u8string` with C++20 (the new UTF-8 string type as proposed together with `char8_t` for UTF-8 characters in https://wg21.link/p0482).[4]

The member template `string<>()` can be used to convert to a special string type, such as a string type that operates case insensitively:

```
struct ignoreCaseTraits : public std::char_traits<char> {
  // case-insensitively compare two characters:
  static bool eq(const char& c1, const char& c2) {
    return std::toupper(c1) == std::toupper(c2);
  }
  static bool lt(const char& c1, const char& c2) {
    return std::toupper(c1) < std::toupper(c2);
  }
  // compare up to n characters of s1 and s2:
  static int compare(const char* s1, const char* s2, std::size_t n);
  // search character c in s:
  static const char* find(const char* s, std::size_t n, const char& c);
};

// define a special type for such strings:
using icstring = std::basic_string<char, ignoreCaseTraits>;

std::filesystem::path p{"/dir\\subdir/subsubdir\\/./\\"};
icstring s2 = p.string<char,ignoreCaseTraits>();
```

Note also that you should *not* use a function `c_str()`, which is also provided, because it converts to the *native* string format, which might be a `wchar_t` so that you, for example, have to use `std::wcout` instead of `std::cout` to write it to a stream.

[4] Thanks to Tom Honermann for pointing this out and the proposed change (it is really important that C++ gets real UTF-8 support).

20.3.4 Conversions Between Native and Generic Format

Table *Conversions Between Native and Generic Format* lists the operations to convert between the generic path format and the implementations-specific format of the actual platform.

Call	Effect
`p.generic_string()`	yields the path as a generic `std::string`
`p.generic_wstring()`	yields the path as a generic `std::wstring`
`p.generic_u8string()`	yields the path as a generic `std::u8string`
`p.generic_u16string()`	yields the path as a generic `std::u16string`
`p.generic_u32string()`	yields the path as a generic `std::u32string`
`p.generic_string<...>()`	yields the path as a generic `std::basic_string<...>`
`p.native()`	yields the path in the native format of type `path::string_type`
conversionToNativeString	implicit conversion to the native string type
`p.c_str()`	yields the path as a character sequence in the native string format
`p.make_preferred()`	replaces directory separators in `p` by native format and yields the modified `p`

Table 20.6. Conversions Between Native and Generic Format

These functions should have no effect under POSIX-based systems, where there is no difference between the native and the generic path format. Calling these functions on other platforms might matter:

- The `generic...()` path functions yields the path converted to the corresponding string format having the generic format,
- `native()` yields the path converted to the native string encoding, which is defined by the type `std::filesystem::path::string_type`. Under Windows this type is type `std::wstring`, so that you have to use `std::wcout` instead of `std::cout` to directly write it to the standard output stream. New overloads allow us to pass the native string to new overloads of file streams.
- `c_str()` does the same but yields the result as a null terminated character sequence. Note that using this function also is not portable, because printing the sequence with `std::cout` does not result in the correct output on Windows. You have to use `std::wcout` there.
- `make_preferred()` replaces any directory separator except for the root name by the native directory separator. Note that this is the only function that modifies the path it is called for. Thus, strictly speaking belongs to the next section of modifying path function, but because it deals with the conversions for the native format it is also listed here.

For example, under Windows the following code:

```
std::filesystem::path p{"/dir\\subdir/subsubdir\\/./\\"};
std::cout <<  "p:                  " << p << '\n';
std::cout <<  "string():           " << p.string() << '\n';
std::wcout << "wstring():          " << p.wstring() << '\n';
std::cout <<  "lexically_normal(): " << p.lexically_normal() << '\n';
std::cout <<  "generic_string():   " << p.generic_string() << '\n';
```

```
std::wcout << "generic_wstring():  " << p.generic_wstring() << '\n';
// because it's Windows and the native string type is wstring:
std::wcout << "native():           " << p.native() << '\n';  // Windows!
std::wcout << "c_str():            " << p.c_str() << '\n';
std::cout << "make_preferred():   " << p.make_preferred() << '\n';
std::cout << "p:                  " << p << '\n';
```

has the following output:

```
p:                  "/dir\\subdir/subsubdir\\/./\\"
string():           /dir\subdir/subsubdir\/./\
wstring():          /dir\subdir/subsubdir\/./\
lexically_normal(): "\\dir\\subdir\\subsubdir\\"
generic_string():   /dir/subdir/subsubdir//.//
generic_wstring():  /dir/subdir/subsubdir//.//
native():           /dir\subdir/subsubdir\/./\
c_str():            /dir\subdir/subsubdir\/./\
make_preferred():   "\\dir\\subdir\\subsubdir\\\\.\\\\"
p:                  "\\dir\\subdir\\subsubdir\\\\.\\\\"
```

Note again:

- The native string type is not portable. On Windows it is a `wstring`, on POSIX-based systems it is a `string`, so that you would have to use `cout` instead of `wcout` to print the result of `native()` and `c_str()`. Using `wcout` is only portable for the return value of `wstring()` and `generic_wstring()`.
- Only the call of `make_preferred()` modifies the path it is called for. All other calls leave `p` unaffected.

20.3.5 Path Modifications

Table *Path Modifications* lists the operations that allow us to modify paths directly.

While `+=` and `concat()` just append new characters to a path, `/`, `/=`, and `append()` a sub path separated with the current directory separator:

```
std::filesystem::path p{"myfile"};
p += ".git";                // p: myfile.git
p /= ".git";                // p: myfile.git/.git
p.concat("1");              // p: myfile.git/git1
p.append("1");              // p: myfile.git/git1/1
std::cout << p << '\n';
std::cout << p / p << '\n';
```

On POSIX based systems the output is:

```
"myfile.git/.git1/1"
"myfile.git/.git1/1/myfile.git/.git1/1"
```

Call	Effect
`p = p2`	assign a new path
`p = sv`	assign a string (view) as a new path
`p.assign(p2)`	assign a new path
`p.assign(sv)`	assign a string (view) as a new path
`p.assign(beg, end)`	assign elements of the range from `beg` to `end` to the path
`p1 / p2`	yields the path concatenating `p2` as sub-path of `p1`
`p /= sub`	appends `sub` as sub-path to path `p`
`p.append(sub)`	appends `sub` as sub-path to path `p`
`p.append(beg, end)`	appends elements of the range from `beg` to `end` as sub-paths to path `p`
`p += str`	appends the characters of `str` to path `p`
`p.concat(sub)`	appends the characters of `str` to path `p`
`p.concat(beg, end)`	appends elements of the range from `beg` to `end` to path `p`
`p.remove_filename()`	remove a trailing filename from the path
`p.replace_filename(repl)`	replace the trailing filename (if any)
`p.replace_extension()`	remove any trailing filename extension
`p.replace_extension(repl)`	replace the trailing filename extension (if any)
`p.clear()`	make the path empty
`p.swap(p2)`	swap the values of two paths
`swap(p1, p2)`	swap the values of two paths
`p.make_preferred()`	replaces directory separators in `p` by native format and yields the modified `p`

Table 20.7. Path Modifications

On Windows system the output is:

```
"myfile.git\\.git1\\1"
"myfile.git\\.git1\\1\\myfile.git\\.git1\\1"
```

Note that appending an absolute sub-path means replacing an existing path. For example, after:

```
namespace fs = std::filesystem;
auto p1 = fs::path("/usr") / "tmp";      // path is /usr/tmp or /usr\tmp
auto p2 = fs::path("/usr/") / "tmp";     // path is /usr/tmp
auto p3 = fs::path("/usr") / "/tmp";     // path is /tmp
auto p4 = fs::path("/usr/") / "/tmp";    // path is /tmp
```

we have 4 paths referring to two different files:

- `p1` and `p2` are equal and refer to the file `/usr/tmp` (note that on Windows they are equal with `p1` being `/usr\tmp`)
- `p3` and `p4` are equal and refer to the file `/tmp` because an absolute path was appended.

For root elements it also matters whether a new element is assigned. For example, under Windows we have:

```
auto p1 = fs::path("usr") / "C:/tmp";    // path is C:/tmp
auto p2 = fs::path("usr") / "C:";        // path is C:
auto p3 = fs::path("C:") / "";           // path is C:
auto p4 = fs::path("C:usr") / "/tmp";    // path is C:/tmp
auto p5 = fs::path("C:usr") / "C:tmp";   // path is C:usr\tmp
auto p6 = fs::path("C:usr") / "c:tmp";   // path is c:tmp
auto p7 = fs::path("C:usr") / "D:tmp";   // path is D:tmp
```

The function `make_preferred()` converts the directory separators inside a path to the native format. For example:

```
std::filesystem::path p{"//server/dir//subdir///file.txt"};
p.make_preferred();
std::cout << p << '\n';
```

writes on POSIX-based platforms:

```
"//server/dir/subdir/file.txt"
```

On Windows, the output is as follows:

```
"\\\\server\\dir\\\\subdir\\\\\\file.txt"
```

Note that the leading root name is not modified because it has to consist of two slashes or backslashes. Note also that this function can't convert backslashes to a slash on a POSIX-based system, because the backslash is not recognized as a directory separator.

`replace_extension()` replaces, adds, or removes an extension:

- If the file has an extension it is replaced
- If the file has no extension, the new extension is added.
- If you skip the new extension or the new extension is empty, any existing extension is removed.

It doesn't matter whether you place a leading dot in the replacement. The function ensures that there is exactly one dot between the stem and the extension of the resulting filename. For example:

```
fs::path{"file.txt"}.replace_extension("tmp")   //file.tmp
fs::path{"file.txt"}.replace_extension(".tmp")  //file.tmp
fs::path{"file.txt"}.replace_extension("")      //file
fs::path{"file.txt"}.replace_extension()        //file
fs::path{"dir"}.replace_extension("tmp")        //dir.tmp
fs::path{".git"}.replace_extension("tmp")       //.git.tmp
```

Note that filenames that are "pure extensions" (such as `.git`) don't count as extensions.[5]

[5] This has changed with C++17. Before C++17, the result of the last statement would have been `.tmp`.

20.3.6 Path Comparisons

Table *Path Comparisons* lists the operations you can use to compare two different paths.

Call	Effect
`p1 == p2`	yields whether two paths are equal
`p1 != p2`	yields whether two paths are not equal
`p1 < p2`	yields whether a paths is less than another
`p1 <= p2`	yields whether a path is less or equal than another
`p1 >= p2`	yields whether a path is greater or equal than another
`p1 > p2`	yields whether a path is greater than another
`p.compare(p2)`	yields whether `p2` is less, equal, or greater than `p`
`p.compare(sv)`	yields whether `p2` is less, equal, or greater than the string (view) `sv` converted to a path
`equivalent(p1, p2)`	expensive path comparison taking the filesystem into account

Table 20.8. Path Comparisons

Note that most of the comparisons don't take the filesystem into account, which means that they operate only lexically, which is cheap but may result in surprising return values:

- Using `==`, `!=` and `compare()` the following paths are all different:

```
tmp1/f
./tmp1/f
tmp1/./f
tmp1/tmp11/../f
```

- Only different formats of specifying a directory separator are detected. Thus, the following paths are all equal (provided the backslash is a valid directory separator):

```
tmp1/f
/tmp1//f
/tmp1\f
tmp1/\/f
```

Only if you call `lexically_normal()` for each path, all of the paths above are equal (provided the backslash is a valid directory separator). For example:

```
std::filesystem::path p1{"tmp1/f"};
std::filesystem::path p2{"./tmp1/f"};

p1 == p2                                                  //true
p1.compare(p2)                                            //not 0
p1.lexically_normal() == p2.lexically_normal()            //true
p1.lexically_normal().compare(p2.lexically_normal())      //0
```

If you want to take the filesystem into account so that symbolic links are correctly handled, you can use `equivalent()`. Note, however, that this function requires that both paths represent existing files. Thus, a generic way to compare paths as accurate as possible (but not having the best performance) is as follows:

```
bool pathsAreEqual(const std::filesystem::path& p1,
                   const std::filesystem::path& p2)
{
  return exists(p1) && exists(p2) ? equivalent(p1, p2)
         : p1.lexically_normal() == p2.lexically_normal();
}
```

20.3.7 Other Path Operations

Table *Other Path Operations* lists the remaining path operations not listed yet.

Call	Effect
`p.hash_value()`	yields the hash value for a path

Table 20.9. Other Path Operations

Note that only equal paths have the same hash_value. That is the following paths yield different hash values:

```
tmp1/f
./tmp1/f
tmp1/./f
tmp1/tmp11/../f
```

For this reason, you might want to normalize paths before you put them in a hash table.

20.4 Filesystem Operations

This section covers the more expensive filesystem operations that take the current filesystem into account.

Because these operations usually take the filesystem into account (care for existing files, symbolic links, etc.), they significant more expensive than pure path operations As a consequence, they are usually free-standing functions.

20.4.1 File Attributes

There are a couple of attributes you can get about a file behind a given path. First, table *Operations for File Types* lists the functions you can call to inspect whether the file specified by a path `p` exists

and its overall type (if any). Note that these operations do take the filesystem into account and are therefore free-standing functions.

Call	Effect
`exists(p)`	yields whether there is a file to open
`is_symlink(p)`	yields whether the file `p` exists and is a symbolic link
`is_regular_file(p)`	yields whether the file `p` exists and is a regular file
`is_directory(p)`	yields whether the file `p` exists and is a directory
`is_other(p)`	yields whether the file exists `p` and is neither regular nor a directory nor a symbolic link
`is_block_file(p)`	yields whether the file `p` exists and is a block special file
`is_character_file(p)`	yields whether the file `p` exists and is a character special file
`is_fifo(p)`	yields whether the file `p` exists and is FIFO or pipe file
`is_socket(p)`	yields whether the file `p` exists and is a socket

Table 20.10. Operations for File Types

The function for the filesystem type match with the corresponding `file_type` values. However, note that these function (except `is_symlink()`) follow symbolic links. That is, for a symbolic link to a directory both `is_symlink()` and `is_directory()` yield `true`.

Note also that for all checks for files that are special files (no regular file, no directory, no symbolic link) `is_other()` also yields `true` according to the definition of other file types.

For implementation-specific file types there is no specific convenience function so that for them only `is_other()` is `true` (and `is_symlink()` if we have a symbolic link to such a file). You can use the file status API to check against these specific types.

To not follow symbolic links, use `symlink_status()` and call these functions for the returned `file_status` as discussed next for `exists()`.

Check for Existence of a File

`exists()` answers the question, whether there is effectively a file to open. Thus, as just discussed, it follows symbolic links. So, it yields `false` if there is a symbolic link to a non-existing file.

As a consequence, code like this does not work as expected:

```
// if not done yet, create a symbolic link to file:
if (!exists(p)) {                    // OOPS: checks if the file p refers to doesn't exist
  std::filesystem::create_symlink(file, p);
}
```

If `p` already exists as a symbolic link to a non-existing file, it will try to create the symbolic link at the location where already the symbolic link exists and raise a corresponding exception.

Because the situation in a multi-user/multi-process filesystem can change at any time, it usually is the best approach to try to perform an operation and handle the error if it fails. Thus, we can simply

call the operation and handle a corresponding exception or handle an error code passed as additional argument.

However, sometime you need the check for the existence of a file (before performing a filesystem operation). For example, if you want to create a file at a specific location and there already is a symbolic link, the file gets created or is overwritten at a maybe unexpected location. In that case, you should check for the existence of a file as follows:[6]

```
if (!exists(symlink_status(p))) {   // OK: checks if p doesn't exist yet (as symbolic link)
  ...
}
```

Here we use `symlink_status()`, which yields the status *not* following symbolic links, to check for the existence of any file at the location of `p`.

Other File Attributes

Table *Operations for File Attributes* lists a couple of free-standing functions to check for additional file attributes.

Call	Effect
`is_empty(p)`	yields whether the file is empty
`file_size(p)`	yields the size of a file
`hard_link_count(p)`	yields the number of hard links
`last_write_time(p)`	yields the timepoint of the last write to a file

Table 20.11. Operations for File Attributes

Note that there is a difference whether a path is empty and whether the file specified by a path is empty:

```
p.empty()      // true if path p is empty (cheap operation)
is_empty(p)    // true if file at path p is empty (filesystem operation)
```

`file_size(p)` returns the size of file `p` in bytes if it exists as regular file (as if the member `st_size` of the POSIX function `stat()`. For all other files the result is implementation-defined and not portable.

`hard_link_count(p)` returns the number of times a file exists in a file system. Usually this number is `1`, but on some file systems the same file can exist at different locations in the file system (i.e., has different paths). This is different from a symbolic link where a file refers to another file. Here we have a file with different path to access it directly. Only if the last hard link is removed, the file itself is removed.

[6] Thanks to Billy O'Neal for pointing this out.

Dealing with the Last Modification

`last_write_time(p)` returns the timepoint of the last modification or write access of the file. The return type is a special `time_point` type of the standard chrono library for timepoints:

```
namespace std::filesystem {
  using file_time_type = chrono::time_point<trivialClock>;
}
```

The clock type *trivialClock* is an implementation specific clock type that reflects the resolution and range of file time values. For example, you can use it as follows:

```
void printFileTime(const std::filesystem::path& p)
{
  auto filetime = last_write_time(p);
  auto diff = std::filesystem::file_time_type::clock::now() - filetime;
  std::cout << p << " is "
            << std::chrono::duration_cast<std::chrono::seconds>(diff).count()
            << " Seconds old.\n";
}
```

which might output:

```
"fileattr.cpp" is 4 Seconds old.
```

Instead of

```
std::filesystem::file_time_type::clock::now()
```

in this example, you could also write:

```
decltype(filetime)::clock::now()
```

Note that the clock used by filesystem timepoint is not guaranteed to be the standard `system_clock`. For this reason, there is no standardized support to convert the filesystem timepoint into type `time_t` to use it as absolute time in strings or output. There is a workaround, though. The following function "roughly" converts a timepoint of any clock to a `time_t` object:

```
template<typename TimePoint>
std::time_t toTimeT(TimePoint tp)
{
  using system_clock = std::chrono::system_clock;
  return system_clock::to_time_t(system_clock::now()
                                 + (tp - decltype(tp)::clock::now()));
}
```

The trick is to compute the time of the filesystem timepoint as duration relative to now and then add this difference to the current time of the system clock. This function is not exact because both clocks might have different resolutions and we call `now()` twice at slightly different times. However, in general, this works pretty well.

For example, for a path `p` we can call:

```
auto ftime = last_write_time(p);
```

```
std::time_t t = toTimeT(ftime);
// convert to calendar time (including skipping trailing newline):
std::string ts = ctime(&t);
ts.resize(ts.size()-1);
std::cout << "last access of " << p << ": " << ts << '\n';
```

which might print:

```
last access of "fileattr.exe": Sun Jun 24 10:41:12 2018
```

To format a string the way we want we can call:

```
std::time_t t = toTimeT(ftime);
char mbstr[100];
if (std::strftime(mbstr, sizeof(mbstr), "last access: %B %d, %Y at %H:%M\n",
                  std::localtime(&t))) {
  std::cout << mbstr;
}
```

which might output:

```
last access: June 24, 2018 at 10:41
```

A useful helper to convert any filesystem timepoint to a string would be:

filesystem/ftimeAsString.hpp

```
#include <string>
#include <chrono>
#include <filesystem>

std::string asString(const std::filesystem::file_time_type& ft)
{
  using system_clock = std::chrono::system_clock;
  auto t = system_clock::to_time_t(system_clock::now()
              + (ft - std::filesystem::file_time_type::clock::now()));
  // convert to calendar time (including skipping trailing newline):
  std::string ts = ctime(&t);
  ts.resize(ts.size()-1);
  return ts;
}
```

Note that `ctime()` and `strftime()` are not thread-safe and must not be called concurrently.

See Modify Existing Files for the corresponding API to modify the last write access.

20.4.2 File Status

To avoid filesystem access, there is a special type `file_status` that can be used to hold and modify file type and permissions are cached. this status can be set

- when asking for the file status of a specific path as listed in table *Operations for File Status*
- when iterating over a directory

Call	Effect
`status(p)`	yields the `file_status` of the file `p` (following symbolic links)
`symlink_status(p)`	yields the `file_status` of `p` (not following symbolic links) `p`

Table 20.12. Operations for File Status

The difference is that if the path `p` resolves in a symbolic link `status()` follows the link and prints the attributes of the file there (the status might be that there is no file), while `symlink_status(p)` prints the status of the symbolic link itself.

Table `file_status` *Operations* lists the possible calls for a `file_status` object `fs`.

Call	Effect
`exists(fs)`	yields whether a file exists
`is_regular_file(fs)`	yields whether the file exists and is a regular file
`is_directory(fs)`	yields whether the file exists and is a directory
`is_symlink(fs)`	yields whether the file exists and is a symbolic link
`is_other(fs)`	yields whether the file exists and is neither regular nor a directory nor a symbolic link
`is_character_file(fs)`	yields whether the file exists and is a character special file
`is_block_file(fs)`	yields whether the file exists and is a block special file
`is_fifo(fs)`	yields whether the file exists and is FIFO or pipe file
`is_socket(fs)`	yields whether the file exists and is a socket
`fs.type()`	yields the `file_type` of the file
`fs.permissions()`	yields the permissions of the file

Table 20.13. `file_status` *Operations*

One benefit of the status operations is that you can save multiple operating system calls for the same file. For example, instead of

```
if (!is_directory(path)) {
  if (is_character_file(path) || is_block_file(path)) {
    ...
  }
```

```
  ...
}
```

you better implement:

```
auto pathStatus{status(path)};
if (!is_directory(pathStatus)) {
  if (is_character_file(pathStatus) || is_block_file(pathStatus)) {
    ...
  }
  ...
}
```

The other key benefit is that by using `symlink_status()` you can check for the status of a path *without* following any symbolic link. This, for example, helps to check whether any file exists at a specific path.

Because these file status don't use the operating system, no overloads to return en error code are provided.

The `exists()` and `is_...()` functions for path arguments are shortcuts for calling and checking the `type()` for a file status. For example,

```
is_regular_file(mypath)
```

is a shortcut for

```
is_regular_file(status(mypath))
```

which is a shortcut for

```
status(mypath).type() == file_type::regular
```

20.4.3 Permissions

The model to deal with file permissions is adopted from the UNIX/POSIX world. There are bits to signal read, write, and/or execute/search access for owners of the file, members of the same group, or all others. In addition, there are special bits for "set user ID on execution," "set group ID on execution," and the sticky bit (or another system-dependent meaning).

Table *Permission Bits* lists the values of the bitmask type `std::filesystem::perms`, which represent one or multiple permission bits.

You can ask for the current permissions and as a result check the bits of the returned `perms` object. To combine flags, you have to use the bit operators. For example:

```
// if writable:
if ((fileStatus.permissions()
       & (fs::perms::owner_write | fs::perms::group_write
           | fs::perms::others_write))
    != fs::perms::none) {
  ...
}
```

Enum	Octal	POSIX	Meaning
none	0		No permissions set
owner_read	0400	S_IRUSR	Read permission for the owner
owner_write	0200	S_IWUSR	Write permission for the owner
owner_exec	0100	S_IXUSR	Execute/search permission for the owner
owner_all	0700	S_IRWXU	All permissions for the owner
group_read	040	S_IRGRP	Read permission for the group
group_write	020	S_IWGRP	Write permission for the group
group_exec	010	S_IXGRP	Execute/search permission for the group
group_all	070	S_IRWXG	All permission for the group
others_read	04	S_IROTH	Read permission for all others
others_write	02	S_IWOTH	Write permission for all others
others_exec	01	S_IXOTH	Execute/search permission for all others
others_all	07	S_IRWXO	All permissions for all others
all	0777		All permissions for 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		Mask for all possible bits
unknown	0xFFFF		Permissions not known

Table 20.14. Permission Bits

A shorter (but maybe less readable) way to initialize a bitmask would be to directly use the corresponding octal value and relaxed enum initialization:

```
// if writable:
if ((fileStatus.permissions() & fs::perms{0222}) != fs::perms::none) {
  ...
}
```

Note that you have to put the `&` expressions in parentheses before comparing the outcome with a specific value. Note also that you can't skip the comparison because there is no implicit conversion to `bool` for bitmask types.

As another example, to convert the permissions of a file to a string with the notation of the UNIX `ls -l` command, you can use the following helper function:

filesystem/permAsString.hpp

```
#include <string>
#include <chrono>
#include <filesystem>

std::string asString(const std::filesystem::perms& pm)
```

```
{
  using perms = std::filesystem::perms;
  std::string s;
  s.resize(9);
  s[0] = (pm & perms::owner_read)   != perms::none ? 'r' : '-';
  s[1] = (pm & perms::owner_write)  != perms::none ? 'w' : '-';
  s[2] = (pm & perms::owner_exec)   != perms::none ? 'x' : '-';
  s[3] = (pm & perms::group_read)   != perms::none ? 'r' : '-';
  s[4] = (pm & perms::group_write)  != perms::none ? 'w' : '-';
  s[5] = (pm & perms::group_exec)   != perms::none ? 'x' : '-';
  s[6] = (pm & perms::others_read)  != perms::none ? 'r' : '-';
  s[7] = (pm & perms::others_write) != perms::none ? 'w' : '-';
  s[8] = (pm & perms::others_exec)  != perms::none ? 'x' : '-';
  return s;
}
```

This allows you to print the permissions of a file as part of an standard ostream command:

```
std::cout << "permissions: " << asString(status(mypath).permissions()) << '\n';
```

A possible output for a file with all permissions for the owner and read/execute permissions for all others would be:

```
permissions: rwxr-xr-x
```

Note, however, that the Windows ACL (Access Control List) approach does not really fit in this scheme. For this reason, when using Visual C++, writable files ***always*** have all read, write, and execute bits set (even if they are *not* executable files) and files with the read-only flag always have all read and executable bits set. This also impacts the API when modifying permissions portably.

20.4.4 Filesystem Modifications

You can also modify the filesystem either by creating and deleting files or by modifying existing files.

Create and Delete Files

Table *Creating and Deleting Files* lists the operations for a path `p` to create and delete files.

There is no function to create a regular file. This is covered by the I/O Stream standard library. For example, the following statement create a new empty file (if it doesn't exist yet):

```
std::ofstream{"log.txt"};
```

The functions to create a directory return whether a new directory was created. Having a file there is not an error and yields `false` even if the file is not a directory.

The `copy...()` functions don't work with special file types. By default they:

- Report an error if existing files are overwritten
- Don't operate recursively

Call	Effect
`create_directory(p)`	create a directory
`create_directory(p, attrPath)`	create a directory with attributes of `attrPath`
`create_directories(p)`	create a directory and all directories above that don't exists yet
`create_hard_link(old, new)`	create another filesystem entry `to` for the existing file `from`
`create_symlink(to, new)`	create a symbolic link from `new` to `to`
`create_directory_symlink(to, new)`	create a symbolic link from `new` to the directory `to`
`copy(from, to)`	copy a file of any type
`copy(from, to, options)`	copy a file of any type with `options`
`copy_file(from, to)`	copy a file (but not directory or symbolic link)
`copy_file(from, to, options)`	copy a file with `options`
`copy_symlink(from, to)`	copy a symbolic link (`to` refers to where `from` refers)
`remove(p)`	remove a file or empty directory
`remove_all(p)`	remove `p` and recursively all files in its subtree (if any)

Table 20.15. Creating and Deleting Files

- Follow symbolic links

This default can be overwritten by the parameter `options`, which has the bitmask type, `copy_options` defined namespace `std::filesystem`. Table *Copy Options*. lists the possible values.

copy_options	Effect
`none`	Default (value `0`)
`skip_existing`	Skip overwriting existing files
`overwrite_existing`	Overwrite existing files
`update_existing`	Overwrite existing files if the new files are newer
`recursive`	Recursively copy sub-directories and their contents
`copy_symlinks`	Copy symbolic links as symbolic links
`skip_symlinks`	Ignore symbolic links
`directories_only`	Copy directories only
`create_hard_links`	Create additional hard links instead of copies of files
`create_symlinks`	Create symbolic links instead of copies of files (the source path must be an absolute path unless the destination path is in the current directory)

Table 20.16. Copy Options

When creating a symbolic link to a directory you should prefer `create_directory_symlink()`. over `create_symlink()`, because some operating systems require to specify explicitly that the target is a directory. Note that the first argument has to be the path to where the symbolic link refers from the view of the symbolic link. Thus, to create a symbolic link from `sub/slink` to `sub/file.txt` you have to call:

```
std::filesystem::create_symlink("file.txt",  "sub/slink");
```

The statement

```
std::filesystem::create_symlink("sub/file.txt",  "sub/slink");
```

would create a symbolic link from `sub/slink` to `sub/sub/file.txt`.

The functions to remove files have the following behavior:

- `remove()` removes a file or an empty directory. It returns `false` if there was no file/directory or the it could not be removed and no exception is thrown.
- `remove_all()` removes a file or recursively a directory. It returns as `uintmax_t` value how many files were removed. It returns `0` if there was no file and `uintmax_t(-1)` if an error occurred and no exception is thrown.

In both cases symbolic links are removed rather than any file they refer to.

Note that you should always qualify `remove()` correctly when passing a string as argument, because otherwise the C function `remove()` is called.

Modify Existing Files

Table *File Modifications* list the operations to modify existing files.

Call	Effect
`rename(old, new)`	rename and/or move a file
`last_write_time(p, newtime)`	change the timepoint of the last write access
`permissions(p, prms)`	replace the permissions of a file by `prms`
`permissions(p, prms, mode)`	modify the permissions of a file according to `mode`
`resize_file(p, newSize)`	change the size of a regular file

Table 20.17. File Modifications

`rename()` can deal with any type of file including directories and symbolic links. For symbolic links the link is renamed, not where it refers to. Note that `rename()` needs the full new path including filename to move it to a different directory:

```
// move "tmp/sub/x" to "tmp/x":
std::filesystem::rename("tmp/sub/x", "top");     // ERROR
std::filesystem::rename("tmp/sub/x", "top/x");   // OK
```

`last_write_time()` uses the timepoint format as described in Dealing with the Last Modification. For example:

```
// touch file p (update last file access):
```

```
last_write_time(p, std::filesystem::file_time_type::clock::now());
```

`permissions()` uses the permission API format as described in Permissions. The optional `mode` is of the bitmask type `std::filesystem::perm_options` and allows on one hand to choose between `replace`, `add`, and `remove` and on the other hand with `nofollow` to modify permissions of the symbolic links instead of the files they refer to.

For example:

```
// remove write access for group and any access for others:
permissions(mypath,
            std::filesystem::perms::group_write
             | std::filesystem::perms::others_all,
            std::filesystem::perm_options::remove);
```

Note again that Windows due to its ACL permission concept only supports two modes:

- read, write, and execute/search for all (`rwxrwxrwx`)
- read, execute/search for all (`r-xr-xrwx`)

To switch portably between these two modes, you have to enable or disable all three write flags together (removing one after the other does not work):

```
// portable value to enable/disable write access:
auto allWrite = std::filesystem::perms::owner_write
                  | std::filesystem::perms::group_write
                  | std::filesystem::perms::others_write;
// portably remove write access:
permissions(file, allWrite, std::filesystem::perm_options::remove);
```

A shorter (but maybe less readable) way to initialize `allWrite` (using relaxed enum initialization) would be as follows:

```
std::filesystem::perms allWrite{0222};
```

`resize_file()` can be used to reduce or extend the size of a regular file: For example:

```
// make file empty:
resize_file(file, 0);
```

20.4.5 Symbolic Links and Filesystem-Dependent Path Conversions

Table *Filesystem Path Conversions* lists the operations to deal with the path of files taking the filesystem into account. This is especially important if you have to deal with symbolic links. See pure path conversions for cheap path conversions not taking the filesystem into account.

Note that these calls handle it differently whether the file(s) must exists, whether they normalize, and whether they follow symbolic links. Table *Filesystem Path Conversion Attributes* gives an overview of what the functions require and perform.

The following function demonstrates the usage and effect of most of these operations (when dealing with symbolic links):

filesystem/symlink.hpp

Call	Effect
`read_symlink(symlink)`	yields the file an existing symbolic link refers to
`absolute(p)`	yields existing `p` as absolute path (not following symbolic links)
`canonical(p)`	yields existing `p` as absolute path (following symbolic links)
`weakly_canonical(p)`	yields `p` as absolute path (following symbolic links)
`relative(p)`	yields relative (or empty) path from current directory to `p`
`relative(p, base)`	yields relative (or empty) path from `base` to `p`
`proximate(p)`	yields relative (or absolute) path from current directory to `p`
`proximate(p, base)`	yields relative (or absolute) path from `base` to `p`

Table 20.18. Filesystem Path Conversions

Call	Must Exist	Normalizes	Follows Symbolic Links
`read_symlink()`	yes	yes	once
`absolute()`	no	yes	no
`canonical()`	yes	yes	all
`weakly_canonical()`	no	yes	all
`relative()`	no	yes	all
`proximate()`	no	yes	all

Table 20.19. Filesystem Path Conversion Attributes

```
#include <filesystem>
#include <iostream>

void testSymLink(std::filesystem::path top)
{
  top = absolute(top);        // use absolute paths as we change current path
  create_directory(top);      // make sure top exists
  current_path(top);          // so that we can change the directory to it
  std::cout << std::filesystem::current_path() << '\n'; // print path of top

  // define our sub-directories (without creating them):
  std::filesystem::path px{top / "a/x"};
  std::filesystem::path py{top / "a/y"};
  std::filesystem::path ps{top / "a/s"};

  // print some relative paths (for non-existing files):
  std::cout << px.relative_path() << '\n';          // relative path from top
  std::cout << px.lexically_relative(py) << '\n'; // to px from py: "../x"
  std::cout << relative(px, py) << '\n';           // to px from py: "../x"
```

```
  std::cout << relative(px) << '\n';              // to px from curr. path: "a/x"

  std::cout << px.lexically_relative(ps) << '\n'; // to px from ps: "../x"
  std::cout << relative(px, ps) << '\n';          // to px from ps: "../x"

  // now create all sub-directories and the symbolic link:
  create_directories(px);
  create_directories(py);
  if (!is_symlink(ps)) {
    create_directory_symlink(top, ps);
  }
  std::cout << "ps: " << ps << '\n'
            << " -> " << read_symlink(ps) << '\n';

  // and see the difference between lexically and filesystem relative:
  std::cout << px.lexically_relative(ps) << '\n'; // to px from ps: "../x"
  std::cout << relative(px, ps) << '\n';          // to px from ps: "a/x"
}
```

Note that we first convert a possible relative path to an absolute path because otherwise changing the current path affects the location of the path variables. `relative_path()` and `lexically_relative()` are cheap path member functions not taking the actual filesystem into account. Thus, they ignore symbolic links. The free-standing function `relative()` takes the filesystem into account. As long as we don't have files yet, it acts like `lexically_relative()`. But after creating the symbolic link `ps` (`top/a/s`), it follows the symbolic links and gives a different result.

On POSIX systems calling the function from `"/tmp"` with the argument `"top"` the output is as follows:

```
"/tmp/sub"
"tmp/sub/a/x"
"../x"
"../x"
"a/x"
"../x"
"../x"
ps: "/tmp/sub/a/s" -> "/tmp/sub"
"../x"
"a/x"
```

On Windows systems calling the function from `"C:/temp"` with the argument `"top"` the output is as follows:

```
"C:\\temp\\top"
"temp\\top\\a/x"
"..\\x"
```

```
"..\\x"
"a\\x"
"..\\x"
"..\\x"
ps: "C:\\temp\\top\\a/s" -> "C:\\temp\\top"
"..\\x"
"a\\x"
```

Note again that you need administrator rights to create symbolic links on Windows.

20.4.6 Other Filesystem Operations

Table *Other Operations* lists other filesystem operations not mentioned yet.

Call	Effect
`equivalent(p1, p2)`	yields whether `p1` and `p2` refer to the same file
`space(p)`	yields information about the disk space available at path `p`
`current_path(p)`	sets the path of the current working directory to `p`

Table 20.20. Other Operations

The `equivalent()` function is discussed in the section about path comparisons.

The return value of `space()` is the following structure:

```
namespace std::filesystem {
  struct space_info {
    uintmax_t capacity;
    uintmax_t free;
    uintmax_t available;
  };
}
```

Thus, using structured bindings you can print the available disk space of root as follows:

```
auto [cap, _, avail] = std::filesystem::space("/");
std::cout << std::fixed << std::precision(2)
          << avail/1.0e6 << " of " << cap/1.0e6 << " MB available\n\n";
```

The output might be for example:

```
43019.82 of 150365.79 MB available
```

`current_path()` called for a path argument modifies the current working directory of the program. The following way you can switch to another working directory and restore the old one, when leaving the scope:

```
auto current{std::filesystem::current_path()};
try {
```

```
    std::filesystem::current_path(subdir);
    ...
  }
  catch (...) {
    std::filesystem::current_path(current);
    throw;
  }
  std::filesystem::current_path(subdir);
```

20.5 Iterating Over Directories

One key application of the filesystem library is to iterate over directories or all files of a filesystem (sub)tree.

The most convenient way to do it is to use a range-based `for` loop. You can iterate over all file in a directory:

```
for (const auto& e : std::filesystem::directory_iterator(dir)) {
  std::cout << e.path() << '\n';
}
```

or iterate recursively over all files in a filesystem (sub)tree:

```
for (const auto& e : std::filesystem::recursive_directory_iterator(dir)) {
  std::cout << e.path() << '\n';
}
```

The passed argument `dir` can be a `path` or anything implicitly convertible to a path (especially all forms of strings);

Note that `e.path()` yields the filename including the directory the iteration was started from. Thus, if we iterate over `"."` a filename `file.txt` becomes `./file.txt` or `.\file.txt`.

In addition, this path is written quoted to a stream, so that the output for this filename becomes `"./file.txt"` or `".\\file.txt"`. Thus, as discussed before in an initial example, the following loop is more portable:

```
for (const auto& e : std::filesystem::directory_iterator(dir)) {
  std::cout << e.path().lexically_normal().string() << '\n';
}
```

To iterate over the current directory you should pass `"."` as current directory instead of `""`. Passing an empty path works on Windows but is not portable.

Directory Iterators are Ranges

It might looks surprising that you can pass an iterator to a range-based `for` loop, because you usually need a range.

The trick is that both `directory_iterator` and `recursive_directory_iterator` are classes for which global overloads of `begin()` and `end()` are provided:

- `begin()` yields the iterator itself,
- `end()` yields the end iterator, which you can also create with the default constructor

For this reason, you can also iterate as follows:

```
std::filesystem::directory_iterator di{p};
for (auto pos = begin(di); pos != end(di); ++pos) {
  std::cout << pos->path() << '\n';
}
```

Or as follows:

```
for (std::filesystem::directory_iterator pos{p};
     pos != std::filesystem::directory_iterator{};
     ++pos) {
  std::cout << pos->path() << '\n';
}
```

Directory Iterator Options

When iterating over directories you can pass options of the bitmask type `directory_options`, defined in namespace `std::filesystem`. They are listed in table *Directory Iterator Options*.

`directory_options`	**Effect**
`none`	Default (value 0)
`follow_directory_symlink`	Follow symbolic links (rather than skipping them)
`skip_permission_denied`	Skip directories where permission is denied

Table 20.21. Directory Iterator Options

Thus, the default is not to follow symbolic links and to skip directories you are not allowed to iterate over. With `skip_permission_denied` iterating over a denied directory, results in an exception.

createfiles.cpp shows an application of `follow_directory_symlink`.

20.5.1 Directory Entries

The elements directory iterators iterate over are of type `std::filesystem::directory_entry`. Thus, if a directory iterator is valid `operator*()` yields that type. Which means that the correct types of the range-based `for` loop are as follows:

```
for (const std::filesystem::directory_entry& e
       : std::filesystem::directory_iterator(p)) {
  std::cout << e.path() << '\n';
}
```

Directory entries contain both a path object and additional attributes such as hard link count, file status, file size, last write time, whether it is a symbolic link, and where it refers to if it is.

Note that the iterators are input iterators. The reason is that iterating over a directory might result into different results as at any time directory entries might change. This has to be taken into account when using directory iterators in parallel algorithms.

Table *Directory Entry Operations* lists the operations you can call for a directory entry `e`. They are more or less the operations you can call to query file attributes, get the file status check permissions, and compare the paths.

Call	Effect
`e.path()`	yields the filesystem `path` for the current entry
`e.exists()`	yields whether the file exists
`e.is_regular_file()`	yields whether the file exists and is a regular file
`e.is_directory()`	yields whether the file exists and is a directory
`e.is_symlink()`	yields whether the file exists and is a symbolic link
`e.is_other()`	yields whether the file exists and is neither regular nor a directory nor a symbolic link
`e.is_block_file()`	yields whether the file exists and is a block special file
`e.is_character_file()`	yields whether the file exists and is a character special file
`e.is_fifo()`	yields whether the file exists and is FIFO or pipe file
`e.is_socket()`	yields whether the file exists and is a socket
`e.file_size()`	yields the size of a file
`e.hard_link_count()`	yields the number of hard links
`e.last_write_time()`	yields the timepoint of the last write to a file
`e.status()`	yields the status of the file `p`
`e.symlink_status()`	yields the file status (following symbolic links) `p`
`e1 == e2`	yields whether the two entry paths are equal
`e1 != e2`	yields whether the two entry paths are not equal
`e1 < e2`	yields whether an entry paths is less than another
`e1 <= e2`	yields whether an entry path is less or equal than another
`e1 >= e2`	yields whether an entry path is greater or equal than another
`e1 > e2`	yields whether an entry path is greater than another
`e.assign(p)`	replaces the path of `e` by `p` and updates all entry attributes
`e.replace_filename(p)`	replaces the filename of the current path of `e` by `p` and updates all entry attributes
`e.refresh()`	updates all cached attributes for this entry

Table 20.22. Directory Entry Operations

`assign()` and `replace_filename()` call the corresponding modifying path operations but do not modify the files in the underlying filesystem.

Directory Entry Caching

Implementations are encouraged to *cache* such additional file attributes to avoid additional filesystem access when using the entries. However, implementations are not required to cache the data, which means that these usually cheap operations might become more expensive.[7]

Because all the values are usually cached, these calls are usually cheap and therefore member functions:

```
for (const auto& e : std::filesystem::directory_iterator{"."})
{
  auto t = e.last_write_time();      // usually cheap
  ...
}
```

Whether cached or not, in a mult-user or multi-process operating system all these iterations might yield data about files that is no longer valid. File contents and therefore sizes might change, file might be removed or replaced (thus, even file types might change), and permissions might get modified.

In that case you can request to refresh the data a directory entry holds:

```
for (const auto& e : std::filesystem::directory_iterator{"."})
{
  ...          // data data becomes old
  e.refresh();    // refresh cache data for the file
  if (e.exists()) {
    auto t = e.last_write_time();
    ...
  }
}
```

Alternatively, you might always ask for the current situation:

```
for (const auto& e : std::filesystem::directory_iterator{"."})
{
  ...          // data data becomes old
  if (exists(e.path())) {
    auto t = last_write_time(e.path());
    ...
  }
}
```

[7] In fact, the beta implementation of the C++17 filesystem library in g++ v9 only caches the file type, not the file size (this might change until the library is released).

20.6 Afternotes

The filesystem library was developed under the lead of Beman Dawes for many years as a Boost library. In 2014 for the first time it became a formal beta standard, the *File System Technical Specification* (see https://wg21.link/n4100).

With https://wg21.link/p0218r0 the *File System Technical Specification* was adopted to the standard library as proposed by Beman Dawes. Support to compute relative paths was added by Beman Dawes, Nicolai Josuttis, and Jamie Allsop in https://wg21.link/p0219r1. A couple of minor fixes were added proposed by Beman Dawes in https://wg21.link/p0317r1, by Nicolai Josuttis in https://wg21.link/p0392r0, by Jason Liu and Hubert Tong in https://wg21.link/p0430r2, and especially by the members of the filesystem small group (Beman Dawes, S. Davis Herring, Nicolai Josuttis, Jason Liu, Billy O'Neal, P.J. Plauger, and Jonathan Wakely) in https://wg21.link/p0492r2.

Part IV

Library Extensions and Modifications

This part introduces extensions and modifications to existing library components with C++17.

This page is intentionally left blank

Chapter 21

Type Traits Extensions

Regarding type traits (standard type functions), C++17 extends the general abilities to use them and introduces some new type traits.

21.1 Type Traits Suffix _v

Since C++17 you can use the suffix `_v` for all type traits yielding a value (as you can use the suffix `_t` for all type traits yielding a type). For example, for any type `T` instead of

```
std::is_const<T>::value
```

you can now write:

```
std::is_const_v<T>          // since C++17
```

This applies to all type traits. The approach is that for each standard type trait a corresponding variable template is defined. For example:

```
namespace std {
  template<typename T>
  constexpr bool is_const_v = is_const<T>::value;
}
```

Usually this helps to formulate Boolean conditions which you can use at run time:

```
if (std::is_signed_v<char>) {
  ...
}
```

But as type traits are evaluated at compile time, you can use the result at compile time in a compile-time `if`:

```
if constexpr (std::is_signed_v<char>) {
  ...
}
```

or when instantiating templates:

```
// primary template for class C<T>
template<typename T, bool = std::is_pointer_v<T>>
class C {
  ...
};

// partial specialization for pointer types:
template<typename T>
class C<T, true> {
  ...
};
```

Here, class C, for example, provides a special implementation for pointer types.

But the suffix _v also can be used if type traits yield a non-Boolean value, such as std::extent<>, which yields the size of the dimension of a raw array:

```
int a[5][7];
std::cout << std::extent_v<decltype(a)> << '\n';     // prints 5
std::cout << std::extent_v<decltype(a),1> << '\n';   // prints 7
```

21.2 New Type Traits

C++17 introduces a couple of new type traits.

In addition, is_literal_type<> and result_of<> are deprecated since C++17.

DETAILED DESCRIPTION UNDER CONSTRUCTION

is_aggregate<>

std::is_aggregate<*T*> evaluates whether *T* is an aggregate type:

```
template<typename T>
struct D : std::string, std::complex<T> {
   std::string data;
};
D<float> s{{"hello"}, {4.5,6.7}, "world"};          // OK since C++17
std::cout << std::is_aggregate<decltype(s)>::value; // outputs: 1 (true)
```

21.3 std::bool_constant<>

If traits yield Boolean values, they use now the alias template bool_constant<>:

```
namespace std {
    template<bool B>
```

Trait	Effect
is_aggregate<*T*>	Is aggregate type
has_unique_object_representations<*T*>	Any two object with same value have same representation in memory
is_invocable<*T*,*Args*...>	Can be used as callable for *Args*...
is_nothrow_invocable<*T*,*Args*...>	Can be used as callable for *Args*... without throwing
is_invocable_r<*RT*,*T*,*Args*...>	Can be used as callable for *Args*... returning *RT*
is_nothrow_invocable_r<*RT*,*T*,*Args*...>	Can be used as callable for *Args*... returning *RT* without throwing
invoke_result<*T*,*Args*...>	Result type if used as callable for *Args*...
is_swappable<*T*>	Can call swap() for this type
is_nothrow_swappable< *T*>	Can call swap() for this type and that operation can't throw
is_swappable_with<*T*,*T2*>	Can call swap() for these two types with specific value category
is_nothrow_swappable_with<*T*,*T2*>	Can call swap() for these two types with specific value category and that operation can't throw
conjunction<*B*...>	Logical *and* for Boolean traits *B*...
disjunction<*B*... >	Logical *or* for Boolean traits *B*...
negation<*B* >	Logical *not* for Boolean trait *B*

Table 21.1. New Type Traits Since C++17

```
    using bool_constant = integral_constant<bool, B>;  // since C++17
    using true_type = bool_constant<true>;
    using false_type = bool_constant<false>;
}
```

Before C++17, std::true_type and std::false_type were directly defined as alias definitions for std::integral_constant<bool,true> and std::integral_constant<bool,false>, respectively.

Still, Boolean traits usually inherit from std::true_type if a specific property applies and from std::false_type if not. For example:

```
// primary template: in general T is not a void type
template<typename T>
struct IsVoid : std::false_type {
};
// specialization for type void:
template<>
struct IsVoid<void> : std::true_type {
};
```

But now you can define your own type trait by deriving from `bool_constant<>` if you are able to formulate the corresponding compile-time expression as Boolean condition. For example:

```
template<typename T>
struct IsLargerThanInt
 : std::bool_constant<(sizeof(T) > sizeof(int))> {
}
```

so that you can use such a trait to compile depending on whether a type is larger than an `int`:

```
template<typename T>
void foo(T x)
{
  if constexpr(IsLargerThanInt<T>::value) {
    ...
  }
}
```

By adding the corresponding variable template for suffix `_v` as inline variable:

```
template<typename T>
inline static constexpr auto IsLargerThanInt_v = IsLargerThanInt<T>::value;
```

you can also shorten the usage of the trait as follows:

```
template<typename T>
void foo(T x)
{
  if constexpr(IsLargerThanInt_v<T>) {
    ...
  }
}
```

As another example, we can define a trait that checks whether the move constructor for a type `T` guarantees not to throw roughly as follows:

```
template<typename T>
struct IsNothrowMoveConstructibleT
 : std::bool_constant<noexcept(T(std::declval<T>()))> {
};
```

21.4 `std::void_t<>`

A little, but incredible useful helper to define type traits was standardized in C++17: `std::void_t<>`. It is simply defined as follows:

```
namespace std {
  template<typename...> using void_t = void;
}
```

That is, it yields void for any variadic list of template parameters. This is helpful, where we only want to deal with types solely in an argument list.

The major application is the ability to check for conditions when defining new type traits. The following example demonstrates the application of this helper:

```
#include <utility>      // for declval<>
#include <type_traits>  // for true_type, false_type, and void_t

// primary template:
template<typename, typename = std::void_t<>>
struct HasVarious : std::false_type {
};

// partial specialization (may be SFINAEd away):
template<typename T>
struct HasVarious<T, std::void_t<decltype(std::declval<T>().begin()),
                                 typename T::difference_type,
                                 typename T::iterator>>
 : std::true_type {
};
```

Here, we define a new type trait HasVariousT<>, which checks for three things:

- Does the type have a member function begin()?
- Does the type have a type member difference_type?
- Does the type have a type member iterator?

The partial specialization is only used, when all the corresponding expressions are valid for a type T. Then it is more specific than the primary template and as we derive from std::true_type, a check against the value of this trait yields true:

```
if constexpr (HasVarious<T>::value) {
  ...
}
```

If any of the expressions results in invalid code (i.e., T has no begin(), or no type member difference_type, or no type member iterator), the partial specialization is *SFINAE'd away*, which means that it is ignored due to the rule that *substitution failure is not an error*. Then, only the primary template is available, which derives from std::false_type, so that a check against the value of this trait yields false.

The same way, you can use std::void_t to easily define other traits checking for one or multiple conditions, where the existence/ability of a member or operation matters.

21.5 Afternotes

Variable templates for standard type traits were first proposed 2014 by Stephan T. Lavavej in https://wg21.link/n3854. They finally were adopted as part of the Library Fundamentals TS as proposed by Alisdair Meredith in https://wg21.link/p0006r0.

The type trait `std::is_aggregate<>` was introduced as a US national body comment for the standardization of C++17 (see https://wg21.link/lwg2911).

`std::bool_constant<>` was first proposed by Zhihao Yuan in https://wg21.link/n4334. They finally were adopted as proposed by Zhihao Yuan in https://wg21.link/n4389.

`std::void_t_<>` was adopted as proposed by Walter E. Brown in https://wg21.link/n3911.

UNDER CONSTRUCTION

Chapter 22

Parallel STL Algorithms

To benefit from modern multi-core architectures, the C++17 standard library introduces the ability to let STL standard algorithms run using multiple threads to deal with different elements in parallel.

Many algorithms were extended by a new first argument to specify, whether and how to run the algorithm in parallel threads (the old way without this argument is, of course, still supported). In addition, some supplementary algorithms were introduced that specifically support parallel processing.

22.1 Using Parallel Algorithms

Let's start with two example programs, demonstrating the ability to let existing algorithm run in parallel and use new parallel algorithms.

22.1.1 A Standard Algorithms in Parallel

Here is a first very simple example of running a standard algorithm in parallel:

lib/parallelcountif.cpp

```
#include <vector>
#include <iostream>
#include <algorithm>
#include <execution>  // for the execution policy

int main()
{
  std::vector<int> coll{0, 8, 15, 2, 3, 7, 42};
  ...
  // count elements with even value:
  auto num = std::count_if(std::execution::par,          // execution policy
                           coll.cbegin(), coll.cend(),   // range
```

```
                          [](int elem){                 // criterion
                            return elem % 2==0;
                          });
  std::cout << "number of elements with even value: " << num << '\n';
}
```

As you can see, using the parallel algorithms is in principle pretty easy:

- Include header `<execution>`
- Call the algorithms the way you would usually call algorithms with an additional first argument, which often will be simply `std::execution::par`.

In this case we use the standard algorithm `count_if()`, to count how many elements of the passed range fulfill a specific predicate. However due to the additional first parameter `std::execution::par` we request the algorithm to run in parallel mode.

```
#include <algorithm>
#include <execution>
...
auto num = std::count_if(std::execution::par,          // execution policy
                         coll.cbegin(), coll.cend(),   // range
                         [](int elem){                 // criterion
                           return elem % 2==0;
                         });
```

As usual, `coll` might here be any range. Note however that all parallel algorithms require the iterators to be at least forward iterators (we iterate through the same elements in different threads, which makes no sense if the iterators would not iterate over the same values, then).

The way the algorithms run in parallel is implementation specific. And of course, using multiple threads might not always be faster, because starting and dealing with multiple threads also takes its time. In fact, for a simple algorithm with a fast predicate as in this example, running in parallel probably doesn't pay off. There should happen something with each element that takes significant time and is independent from the processing of the other elements.

22.1.2 Using New Algorithms

Some supplementary algorithms were introduced to handle the parallel processing of standard algorithms available since C++98.

For example, `reduce()` was introduced as a parallel form of `accumulate()`, which "accumulates" all elements (you can define, which operation performs the "accumulation"). The problem with `accumulate()` for parallel processing is that it guarantees a specific order of calling the operation to "accumulate" the elements, which makes parallelization useless. `reduce()` also accumulates, but no longer gives a specific guarantee for the order of the operation. In many cases this fine (it doesn't matter if we compute `(`*e1*`+`*e2*`)+e3` or *e1*`+(`*e2*`)+e3`). However, for non-associative or non-commutative operations such as adding floating-point numbers results become non-deterministic.

Another new supplementary algorithm, which will be used a lot is, `transform_reduce()`. It "accumulates" in parallel the result of applying a transformation to each element. Here, parallel execution can pay off pretty fast if for example you have to call a non-trivial transformation for each element and compute or combine the results. In fact, I expect this algorithm to be used a lot.

Here is a first very simple example of running a standard algorithm in parallel:

lib/dirsize.cpp

```
#include <vector>
#include <iostream>
#include <numeric>      // for transform_reduce()
#include <execution>    // for the execution policy
#include <filesystem>   // filesystem library

int main(int argc, char* argv[])
{
  // root directory is passed as command line argument:
  if (argc < 2) {
    std::cout << "Usage: " << argv[0] << " <path> \n";
    return EXIT_FAILURE;
  }
  std::filesystem::path root{argv[1]};

  // init list of all file paths in passed file tree:
  std::vector<std::filesystem::path> paths;
  try {
    std::filesystem::recursive_directory_iterator dirpos{root};
    std::copy(begin(dirpos), end(dirpos),
              std::back_inserter(paths));
  }
  catch (const std::exception& e) {
    std::cerr << "EXCEPTION: " << e.what() << std::endl;
    return EXIT_FAILURE;
  }

  // accumulate size of all regular files:
  auto sz = std::transform_reduce(
                  std::execution::par,                  // parallel execution
                  paths.cbegin(), paths.cend(),         // range
                  std::uintmax_t{0},                    // initial value
                  std::plus<>(),                        // accumulate ...
                  [](const std::filesystem::path& p) { // file size if regular file
                    return is_regular_file(p) ? file_size(p)
                                              : std::uintmax_t{0};
```

```
                  });
  std::cout << "size of all " << paths.size()
            << " regular files: " << sz << '\n';
}
```

First, we recursively collect all filesystem paths in the directory given as command-line argument:

```
std::filesystem::path root{argv[1]};

std::vector<std::filesystem::path> paths;
std::filesystem::recursive_directory_iterator dirpos{root};
std::copy(begin(dirpos), end(dirpos),
          std::back_inserter(paths));
```

Note that because we might pass an invalid path, possible (filesystem) exceptions are caught.

Then, we iterate over a collection of the filesystem paths to accumulate their sizes if they are regular files:

```
auto sz = std::transform_reduce(
                  std::execution::par,                  // parallel execution
                  paths.cbegin(), paths.cend(),         // range
                  std::uintmax_t{0},                    // initial value
                  std::plus<>(),                        // accumulate ...
                  [](const std::filesystem::path& p) { // file size if regular file
                    return is_regular_file(p) ? file_size(p)
                                              : std::uintmax_t{0};
                  });
```

The new standard algorithm `transform_reduce()` operates as follows:

- The last argument is applied to each element. Here, passed lambda is called for each path element and queries its size if it is a regular file.
- The second but last argument is the operation that combines all the sizes. Because we want to accumulate the sizes we use the standard function object `std::plus<>`.
- The third but last argument is the initial value for the operation that combines all the sizes. Thus, if the lists of path is empty, we start with 0. We use the same type the the return value of `file_size()`, `std::uintmax_t`.

Note that asking for the size of a file is a pretty expensive operation, because it requires an operating system call. For this reason it pretty fast pays off to use an algorithms that calls this transformation (from path to size) in parallel with multiple threads in any order and computes the sum. First measurements demonstrate a clear win (up to doubling the speed of the program).

Note also that you can't pass the paths the directory iterator iterates over directly to the parallel algorithm, because directory iterators are input iterators while the parallel algorithms require forward iterators.

Finally note that `transform_reduce()` is defined in header `<numeric>` instead of `<algorithm>` (just like `accumulate()` it counts as numeric algorithm.

22.2 Parallel Algorithms in Detail

UNDER CONSTRUCTION

22.3 Afternotes

UNDER CONSTRUCTION

This page is intentionally left blank

Chapter 23

Container Extensions

There are a couple of minor or small changes to the standard containers of the C++ standard library, which are described in this chapter.

23.1 Container-Support of Incomplete Types

Since C++17 `std::vector`, `std::list`, and `std__forward_list` are required to support incomplete types.

The main motivation for this is described in an Article by Matt Austern, called "*The Standard Librarian: Containers of Incomplete Types*" (see http://drdobbs.com/184403814): You can now have a type, which recursively has a member of a container of its type. For example:

```
struct Node
{
  std::string value;
  std::vector<Node> children;   // OK since C++17 (Node is an incomplete type here)
};
```

This also applies to classes with private members and a public API. Here is a complete example:

lib/incomplete.hpp

```
#ifndef NODE_HPP
#define NODE_HPP

#include <vector>
#include <iostream>
#include <string>

class Node
{
```

```
  private:
    std::string value;
    std::vector<Node> children;  // OK since C++17 (Node is an incomplete type here)
  public:
    // create Node with value:
    Node(const std::string& s) : value{s}, children{} {
    }

    // add child node:
    void add(const Node& n) {
      children.push_back(n);
    }

    // access child node:
    Node& operator[](std::size_t idx) {
      return children.at(idx);
    }

    // print node tree recursively:
    void print(int indent = 0) const {
      std::cout << std::string(indent, ' ') << value << '\n';
      for (const auto& n : children) {
        n.print(indent+2);
      }
    }
    ...
};

#endif // NODE_HPP
```

You could use this class, for example, as follows:

lib/incomplete.cpp

```
#include "incomplete.hpp"
#include <iostream>

int main()
{
  // create node tree:
  Node root{"top"};
  root.add(Node{"elem1"});
  root.add(Node{"elem2"});
  root[0].add(Node{"elem1.1"});
```

```
  // print node tree:
  root.print();
}
```

The program has the following output:

```
top
  elem1
    elem1.1
  elem2
```

23.2 Node Handles

By introducing the ability to splice node out of an associative or unordered container, you can easily:

- Modify keys or (unordered) maps or values or (unordered) sets,
- Use move semantics in (unordered) sets and maps, and
- Move elements between (unordered) sets and maps.

For example, after defining and initializing a map as follows:

```
std::map<int, std::string> m{{1,"mango"},
                             {2,"papaya"},
                             {3,"guava"}};
```

you can modify the element with the key 2 as follows:

```
auto nh = m.extract(2);  // nh has type decltype(m)::node_type
nh.key() = 4;
m.insert(std::move(nh));
```

The code takes the node for the element with the key 2 out of the containers, modifies the key, and moves it back as described in Figure 23.1.

Note that no memory (de)allocation is used and that pointers and references to the element remain valid. However, however using the pointers and references while the element is held in the node handle results in undefined behavior.

The type of a node handle is *container*`::node_type`. It provides members

- member `value()` for all (unordered) set types
- members `key()` and `mapped()` for all (unordered) map types

You can also use node handles to move elements from a container to another. The containers can even differ in the following way:

- one supports duplicates while another doesn't (e.g., you can move elements from a multimap to a map)
- comparison functions and hash functions might differ

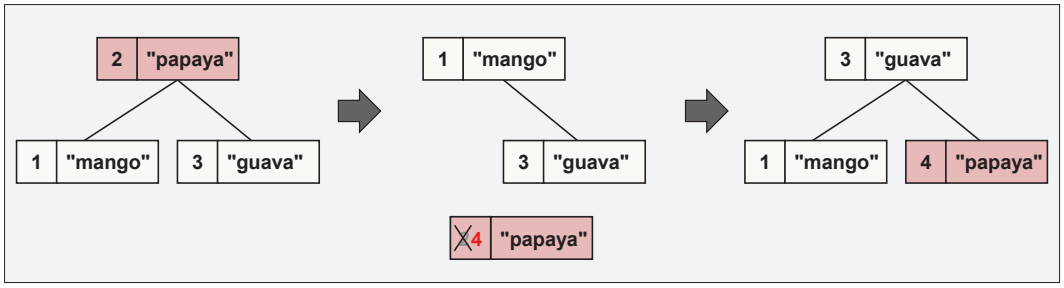

Figure 23.1. Modifying a Key Using Node Handles

For example:

```
std::multimap<double, std::string> src {{1.1,"one"},
                                        {2.2,"two"},
                                        {3.3,"three"}};
std::map<double, std::string> dst {{3.3,"old data"}};

// move some elements from multimap src to map dst:
dst.insert(src.extract(src.find(1.1)));  // splice using an iterator
dst.insert(src.extract(2.2));            // splice using the key
```

Note that the `insert()` function return a structure with three elements (in the following order):

- An iterator **`position`** of the existing element if inserting was not possible.
- A `bool` **`inserted`** to signal whether the insertion was successful.
- The `mode_type` **`node`** with the node handle if the insertion was not possible.

That is, the key information is the second member `inserted`. Using Chapter 1structured bindings you might use the return value as follows:

```
auto [pos,done,node] = dst.insert(src.extract(3.3));
if (!done) {
  std::cout << "insert() of node handle failed:"
            << " tried to insert key '" << node.key()
            << "' with value '" << node.mapped()
            << "' but key exists with value '" << pos->second << "'\n";
}
```

23.3 Afternotes

Container support for incomplete types was first handles were were discussed by Matt Austern in http://drdobbs.com/184403814 and first proposed by Zhihao Yuan in https://wg21.link/

n3890. The finally accepted wording was formulated by Zhihao Yuan in https://wg21.link/n4510.

Node handles were were first proposed indirectly by Alan Talbot requesting splice operations as library issue https://wg21.link/lwg839 and by Alisdair Meredith requesting move support for node elements as library issue https://wg21.link/lwg1041. The finally accepted wording was formulated by Alan Talbot, Jonathan Wakely, Howard Hinnant, and James Dennett in https://wg21.link/p0083r3. The API was slightly clarified finally by Howard E. Hinnant in https://wg21.link/p0508r0.

This page is intentionally left blank

Chapter 24

Multi-Threading and Concurrency

A couple of minor extensions and improvements were introduced in the area of multi-threading and concurrency.

24.1 Supplementary Mutexes and Locks

24.1.1 `std::scoped_lock`

C++11 introduced a simple `std::lock_guard` to have a simple RAII-style way to lock a mutex:

- The constructor locks
- The destructor unlocks (which might be caused by an exception)

Unfortunately, this was not standardized as a variadic template to be able to lock multiple mutexes with a single declaration.

`std::scoped_lock<>` closes this gap. It allows us to lock one or multiple mutexes. For example:

```
#include <mutex>
...
std::vector<std::string> allIssues;
std::mutex allIssuesMx;
std::vector<std::string> openIssues;
std::mutex openIssuesMx;

// lock both issue lists:
{
  std::scoped_lock lg(allIssuesMx, openIssuesMx)
  ...   // manipulate both allIssues and openIssues
```

```
}
```

Note that due to class template argument deduction you don't have to specify the types of the mutexes when declaring `lg`.

This example usage is equivalent to the following code, which is possible to be called since C++11:

```
// lock both issue lists:
{
  std::lock(allIssuesMx, openIssuesMx);       // lock with deadlock avoidance
  std::lock_guard<std::mutex> lg1(allIssuesMx, std::adopt_lock);
  std::lock_guard<std::mutex> lg2(openIssuesMx, std::adopt_lock);
  ...   // manipulate both allIssues and openIssues
}
```

Thus, if more than one mutex is passed, the constructor of `scoped_lock` uses the variadic convenience function `lock(...)`, which guarantees that the call does not result in a deadlock (the standard notes: "*A deadlock avoidance algorithm such as try-and-back-off must be used, but the specific algorithm is not specified to avoid over-constraining implementations*").

If only one mutex is passed to the constructor of a `scoped_lock`, it simply locks the mutex. Thus, in a `scoped_lock` with a single constructor argument acts like a `lock_guard`). For this reason, you can replace all usages of `lock_guard` by `scoped_lock`.

If no mutex is passed, the lock guard has no effect.

Note that you can also adopt multiple locks:

```
// lock both issue lists:
{
  std::lock(allIssuesMx, openIssuesMx);       //Note: deadlock avoidance algorithm used
  std::scoped_lock lg(allIssuesMx, openIssuesMx, std::adopt_lock);
  ...   // manipulate both allIssues and openIssues
}
```

24.1.2 `std::shared_mutex`

C++14 added a `shared_timed_mutex` to support read/write locks, where multiple threads concurrently read a value, while from time to time a thread might update the value. Because on some platforms mutexes that don't support timed locks can be implemented more efficient, now the type `shared_mutex` was introduced (as as `std::mutex` exists besides `std::timed_mutex` since C++11).

`shared_mutex` is defined in header `<shared_mutex>` and supports the following operations:

- for exclusive locks: `lock()`, `try_lock()`, `unlock()`
- for shared read-access: `lock_shared()`, `try_lock_shared()`, `unlock_shared()`
- `native_handle()`

That is, unlike `shared_times_mutex` it doesn't support `try_lock_for()`, `try_lock_until()`, `try_lock_shared_for()`, and `try_lock_shared_until()`.

Using a shared_mutex

The way to use a shared_mutex is as follows: Assume you have a shared vector, which is usually read by multiple threads, but from time to time modified:

```
#include <shared_mutex>
#include <mutex>
...
std::vector<double> v;       // shared resource
std::shared_mutex vMutex;    // control access to v (shared_timed_mutex in C++14)
```

To have shared read-access (so that multiple readers do not block each other), you use a shared_lock, which is a lock guard for shared read access (introduced with C++14). For example:

```
if (std::shared_lock sl(vMutex); v.size() > 0) {
  ...   // (shared) read access to the elements of vector v
}
```

Only for an exclusive write access you use an exclusive lock guard, which might be either a simple lock_guard or scoped_lock (as just introduced) or a sophisticated unique_lock. For example:

```
{
  std::scoped_lock sl(vMutex);
  ...   // exclusive write read access to the vector v
}
```

24.2 is_always_lock_free() for Atomics

You can now check with a C++ library feature whether a specific atomic type can always be used without locks. For example:

```
if constexpr(std::atomic<int>::is_always_lock_free) {
  ...
}
else {
  ...
}
```

If the value is true, then for any object of the corresponding atomic type is_lock_free() yields true:

```
if constexpr(atomic<T>::is_always_lock_free) {
  static_assert(atomic<T>().is_lock_free());  // never fails
}
```

If available, the value fits to the value of the corresponding macro, which had to used before C++17. For example, if and only if ATOMIC_INT_LOCK_FREE yields 2 (which stands for "always"), then std::atomic<int>::is_always_lock_free() yields true:

```
if constexpr(std::atomic<int>::is_always_lock_free) {
  // ATOMIC_INT_LOCK_FREE == 2
  ...
}
else {
  // ATOMIC_INT_LOCK_FREE == 0 || ATOMIC_INT_LOCK_FREE == 1
  ...
}
```

The reason to replace the macro by a static member is to have more type safety and support the use of this checks in tricky generic code (e.g., using SFINAE).

Remember that `std::atomic<>` can also be used for trivially copyable types. Thus, you can also check, whether your own structure would need locks if used atomically. For example:

```
template<auto SZ>
struct Data {
  bool set;
  int values[SZ];
  double average;
};

if constexpr(std::atomic<Data<4>>::is_always_lock_free) {
  ...
}
else {
  ...
}
```

24.3 Cache-Line Sizes

Sometimes for a program it is important to deal with cache-line sizes:

- On one hand, it is important for concurrency that different objects accessed by different threads don't belong to the same cache-line. Otherwise the same amount of memory has to be synchronized between different threads when they are accessed concurrently.[1]
- On the other hand, you might have the goal to place multiple objects in the same cache-line so accessing the first object gives direct access to the others instead of loading them in the cache.

For this, the C++ standard library introduces two inline variables in header `<new>`:

```
namespace std {
  inline constexpr size_t hardware_destructive_interference_size;
  inline constexpr size_t hardware_constructive_interference_size;
```

[1] Accessing multiple objects by different threads concurrently is usually safe in C++, but the necessary synchronization might degrade the performance of the program.

```
}
```

These objects have the following implementation-defined values:

- `hardware_destructive_interference_size` is the recommended minimum offset between two objects that might be accessed by different threads concurrently to avoid worse performance because the same L1 cache line is affected.
- `hardware_constructive_interference_size` is the recommended maximum size of contiguous memory within two objects are placed in the same L1 cache line.

Both values are only hints because the ideal value might depend on the exact architecture. These constants are the best values a compiler can provide dealing with the variety of platforms supported by the generated code. So, if you know better, use specific values, but using these values is better than any assumed fixed size for code supporting multiple platforms.

The values are both at least `alignof(std::max_align_t)`. Usually the value is the same. However, semantically they represent different purposes to use different objects, so you should use them accordingly as follows:

- If you want to access two different (atomic) objects by ***different threads***:

```
struct Data {
  alignas(std::hardware_destructive_interference_size) int valueForOneThread;
  alignas(std::hardware_destructive_interference_size) int valueForAnotherThread;
};
```

- If you want to access two different (atomic) objects by ***the same thread***:

```
struct Data {
  int valueForOneThread;
  int valueForTheSameThread;
};

// double-check we have best performance due to shared cache-line:
static_assert(sizeof(Data) <= std::hardware_constructive_interference_size);

// ensure objects are properly aligned:
alignas(sizeof(Data)) Data myDataForAThread;
```

24.4 Afternotes

`scoped_locks` were originally proposed as modification of `lock_guard` to become variadic by Mike Spertus in https://wg21.link/n4470, which was accepted as https://wg21.link/p0156r0. However, because this turned out to be an ABI breakage, the new name `scoped_lock` was introduced by Mike Spertus with https://wg21.link/p0156r2 and finally accepted.

The `shared_mutex` was first proposed together with all other mutexes for C++11 by Howard Hinnant in https://wg21.link/n2406. However, it took time to convince the C++ standardization

committee that all proposed mutexes are useful. So, the finally accepted wording was formulated for C++17 by Gor Nishanov in https://wg21.link/n4508.

The `std::atomic<>` static member `std::is_always_lock_free` was first proposed by Olivier Giroux, JF Bastien, and Jeff Snyder in https://wg21.link/n4509. The finally accepted wording was also formulated by Olivier Giroux, JF Bastien, and Jeff Snyder in https://wg21.link/p0152r1.

The hardware interference (cache-line) sizes were first proposed by JF Bastien and Olivier Giroux in https://wg21.link/n4523. The finally accepted wording was also formulated by JF Bastien and Olivier Giroux in https://wg21.link/p0154r1.

Part V

Expert Utilities

This part introduces new language and library features that the average application programmer usually doesn't have to know. It might cover tools for programmers of foundation libraries, of specific modes, or in special contexts.

This page is intentionally left blank

Chapter 25

`new` and `delete` with Over-Aligned Data

Since C++11 you can specify *over-aligned* types, having a bigger alignment than the default alignment by using the `alignas` specifier. For example:

```
struct alignas(32) MyType32 {
  int i;
  char c;
  std::string s[4];
};

MyType32 val1;               // 32-bytes aligned
alignas(64) MyType32 val2;   // 64-bytes aligned
```

Note that the alignment value has to be a power of 2 and specifying any value less than the default alignment for the type is an error.[1]

However, ***dynamic/heap allocation*** of over-aligned data is not handled correctly in C++11 and C++14. Using operator `new` for over-aligned types by default ignores the requested alignment, so that a type usually 63-bytes aligned might, for example, only be 8-bytes or 16-bytes aligned.

This gap was closed with C++17. The new behavior has the consequence that new overloads with an alignment argument are provided to be able to provide your own implementations of operator `new` for over-aligned data.

[1] Some compilers accept and ignore alignment values less than the default alignment with a warning or even silently.

25.1 Using new with Alignments

By using an over-aligned type such as:

```
struct alignas(32) MyType32 {
  int i;
  char c;
  std::string s[4];
};
```

a new expression now guarantees that the requested heap memory is aligned as requested (provided over-alignment is supported):

```
MyType32* p = new MyType32;      // since C++17 guaranteed to be 32-bytes aligned
...
```

Before C++17, the request was not guaranteed to be 32-bytes aligned.[2]

As usual, without any value for initialization the object is default initialized, which means that available constructors are called, but (sub)objects of fundamental type have an undefined value. For this reason, you better use *list initialization* with curly braces to ensure that the (sub)objects either have their default value or 0/false/nullptr:

```
MyType32* p = new MyType32{};        // aligned and initialized
```

25.1.1 Distinct Dynamic/Heap Memory Arenas

Note that the request for aligned memory might result in a call to get the memory from a disjoint memory allocation mechanism. For this reason, a request for aligned memory might require a specific corresponding request to deallocate the aligned data. It is *possible* that the memory is allocated with the C11 function aligned_alloc() (which is now also available in C++17). In that case, deallocation with free() would still be fine so that there is no difference compared to memory allocated with malloc().

However, other implementations of new and delete are allowed for platforms, which leads to the requirement that different internal functions have to be used to deallocate default-aligned and over-aligned data. For example, on Windows _aligned_malloc() is usually used, which requires to use _aligned_free() as counterpart.[3]

In contrast to the C standard, the C++ standard respects this situation and therefore conceptionally assumes that there are two disjoint, non-interoperable *memory arenas*, one for default-aligned and one for over-aligned data. Most of the time compilers know how to handle this correctly:

```
std::string* p1 = new std::string;   // using default-aligned memory operations
```

[2] Compilers/platforms don't have to support over-aligned data. In that case a request to over-align should not compile.

[3] The reason is that the Windows operating systems provide no ability to request aligned storage, so that the calls over-allocate and align manually. As a consequence, support for aligned_alloc() will be unlikely in the near future, because support for the existing Windows platforms will still be required.

```
MyType32* p2 = new MyType32;          // using over-aligned memory operations
...
delete p1;                            // using default-aligned memory operations
delete p2;                            // using over-aligned memory operations
```

But sometimes the programmer has to do the right thing as we will see in the remaining sections of this chapter.

25.1.2 Passing the Alignment with the new Expression

There is also way to request a specific over-alignment for a specific call of new. For example:

```
#include <new>   // for align_val_t
...

std::string* p = new(std::align_val_t{64}) std::string;   // 64-bytes aligned
MyType32* p = new(std::align_val_t{64}) MyType32{};       // 64-bytes aligned
...
```

The type std::align_val_t is defined in header <new> as follows:

```
namespace std {
  enum class align_val_t : size_t {
  };
}
```

It is provided to be able now to pass alignment requests to the corresponding implementation of operator new(). Remember that operator new() can be implemented in different ways in C++:

- As a **global** function (different overloads are provided by default, which can be replaced by the programmer).
- As **type-specific** implementations, which can be provided by the programmer and have higher priority than the global overloads.

However, this is the first example, where special care has to be taken to deal correctly with the different dynamic memory arenas, because when specifying the alignment with the new expression the compiler can't use the type to know whether and which alignment was requested. The programmer has to specify which delete to call.[4]

Unfortunately, there is no delete operator, where you can pass an additional argument, you have to call the corresponding operator delete() directly, which means that you have to know, which of the multiple overloads are implemented: In fact, in this example one of the following functions for an object of type T could be called:

```
void T::operator delete(void* ptr, std::size_t size, std::align_val_t align);
```

[4] This is not the first time where the type system is not good enough to call the right implementation of delete. The first example was that it is up to the programmer to ensure that delete[] is called instead of delete if arrays were allocated.

```
void T::operator delete(void* ptr, std::align_val_t align);
void T::operator delete(void* ptr, std::size_t size);
void T::operator delete(void* ptr);
void ::operator delete(void* ptr, std::size_t size, std::align_val_t align);
void ::operator delete(void* ptr, std::align_val_t align);
void ::operator delete(void* ptr, std::size_t size);
void ::operator delete(void* ptr);
```

Yes, it is that complicated, which I will explain later in detail. For the moment use one of three options:

1. Don't use over-alignment directly in `new` expressions.
2. Provide implementations of `operator new()` and `operator delete()` that use the same memory arena (so that calling `delete` is always fine).
3. Provide type-specific implementations of `operator delete()` that match those of `operator new()` and call them directly instead of using `delete` expressions.

Note that you can't use a `typedef` or `using` declaration instead:

```
using MyType64 = alignas(64) MyType32;   // ERROR
typedef alignas(64) MyType32 MyType64;   // ERROR
...
MyType64* p = new MyType64;              // and therefore not possible
```

The reason is that a `typedef` or `using` declaration is only a new name/alias for the original type and what is requested here is a different type following different rules for alignment.

If you want to call an aligned new with getting the `nullptr` as a return value instead of throwing `std::bad_alloc`, you can do this as follows:

```
// allocate a 64-bytes aligned string (nullptr if none):
std::string* p = new(std::align_val_t{64}, std::nothrow) std::string;
if (p != nullptr) {
  ...
}
```

25.2 Implementing `operator new()` for Aligned Memory

In C++ you can provide your own implementation of allocating and deallocating memory when `new` and `delete` are called. This mechanism now also supports passing an alignment parameter.

25.2.1 Implementing Aligned Allocation Before C++17

Globally, C++ provides overloads of `operator new()` and `operator delete()`, which are used unless type-specific implementations are defined. If type-specific implementations of these operators exists, they are used. Note that having one type-specific `operator new()` disables using any of the global `operator new()` implementations for that type (the same applies to `delete`, `new[]`, and `delete[]`).

That is, each time you call new for a type T a corresponding call of either a type specific T::operator new() or (if none exists) the global ::operator new() is called:

```
auto p = new T;     // tries to call a type-specific operator new() (if any)
                    // of if none tries to call a global ::operator new()
```

The same way each time you call delete for a type T a corresponding call of either a type specific T::operator delete() or the global ::operator delete() is called. If arrays are allocated/deallocated, the corresponding type-specific or global operators operator new[]() and operator delete[]() are called.

Before C++17, a requested alignment was not automatically passed to these functions and the default mechanisms allocated dynamic memory without considering the alignment. An over-aligned type always needed its own implementations of operator new() and operator delete() to be correctly aligned on dynamic memory. Even worse, there was no portable way to perform the request for over-aligned dynamic memory.

As a consequence, for example, you had to define something along the lines of following:

lang/alignednew11.hpp

```
#include <cstddef>    // for std::size_t
#include <string>
#if __STDC_VERSION >= 201112L
#include <stdlib.h>   // for aligned_alloc()
#else
#include <malloc.h>   // for _aligned_malloc() or memalign()
#endif

struct alignas(32) MyType32 {
  int i;
  char c;
  std::string s[4];
  ...
  static void* operator new (std::size_t size) {
      // allocate memory for requested alignment:
#if __STDC_VERSION >= 201112L
      // use API of C11:
      return aligned_alloc(alignof(MyType32), size);
#else
#ifdef _MSC_VER
      // use API of Windows:
      return _aligned_malloc(size, alignof(MyType32));
#else
      // use API of Linux:
      return memalign(alignof(MyType32), size);
#endif
#endif
```

```
  }

  static void operator delete (void* p) {
      // deallocate memory for requested alignment:
#ifdef _MSC_VER
      // use special API of Windows:
      _aligned_free(p);
#else
      // C11/Linux can use the general free():
      free(p);
#endif
  }
  // since C++14:
  static void operator delete (void* p, std::size_t size) {
      MyType32::operator delete(p);  // use the non-sized delete
  }
  ...
  // also for arrays (new[] and delete[])
};
```

Note that since C++14 you can provide a `size` argument for the `delete` operator. However, it might happen that the size is not available (e.g., when dealing with incomplete types), and there are cases where platforms can chose whether or not to pass a size argument to `operator delete()`. For this reason, you should always replace both the unsized and the sized overload of `operator delete()` since C++14. Letting one calling the other usually is fine.

With this definition the following code behaved correctly:

lang/alignednew11.cpp

```
#include "alignednew11.hpp"

int main()
{
  auto p = new MyType32;
  ...
  delete p;
}
```

As written, since C++17, you can skip the overhead to implement operations to allocate/deallocate aligned data. The example works well even without defining `operator new()` and `operator delete()` for your type:

lang/alignednew17.cpp

```
#include <string>
```

```
struct alignas(32) MyType32 {
  int i;
  char c;
  std::string s[4];
  ...
};

int main()
{
  auto p = new MyType32;   // allocates 32-bytes aligned memory since C++17
  ...
  delete p;
}
```

25.2.2 Implementing Type-Specific operator new()

If you have to implement your own implementation of operator new() and operator delete(), there is now support for over-aligned data. In practice, the corresponding code for type-specific implementations looks since C++17 as follows:

lang/alignednew.hpp

```
#include <cstddef>    // for std::size_t
#include <new>        // for std::align_val_t
#include <cstdlib>    // for malloc(), aligned_alloc(), free()
#include <string>

struct alignas(32) MyType32 {
  int i;
  char c;
  std::string s[4];
  ...
  static void* operator new (std::size_t size) {
    // called for default-aligned data:
    std::cout << "MyType32::new() with size " << size << '\n';
    return ::operator new(size);
  }
  static void* operator new (std::size_t size, std::align_val_t align) {
    // called for over-aligned data:
    std::cout << "MyType32::new() with size " << size
              << " and alignment " << static_cast<std::size_t>(align)
              << '\n';
    return ::operator new(size, align);
```

```
  }

  static void operator delete (void* p) {
    // called for default-aligned data:
    std::cout << "MyType32::delete() without alignment\n";
    ::operator delete(p);
  }
  static void operator delete (void* p, std::size_t size) {
    MyType32::operator delete(p);          // use the non-sized delete
  }
  static void operator delete (void* p, std::align_val_t align) {
    // called for default-aligned data:
    std::cout << "MyType32::delete() with alignment\n";
    ::operator delete(p, align);
  }
  static void operator delete (void* p, std::size_t size,
                               std::align_val_t align) {
    MyType32::operator delete(p, align);   // use the non-sized delete
  }

  // also for arrays (operator new[] and operator delete[])
  ...
};
```

In principle, we only need the overloads for the additional alignment parameter and call the functions to allocate and deallocate aligned memory. The most portable way is to call the functions globally provided for over-aligned (de)allocation:

```
static void* operator new (std::size_t size, std::align_val_t align) {
  ...
  return ::operator new(size, align);
}
...
static void operator delete (void* p, std::align_val_t align) {
  ...
  ::operator delete(p);
}
```

You could also directly call the C11 functions for aligned allocation:

```
static void* operator new (std::size_t size, std::align_val_t align) {
  ...
  return std::aligned_alloc(static_cast<size_t>(align), size);
}
...
static void operator delete (void* p, std::align_val_t align) {
```

```
  ...
  std::free(p);
}
```

However, due to the problem Windows has with `aligned_alloc()` in practice we need special handling to be portable, then:

```
static void* operator new (std::size_t size, std::align_val_t align) {
  ...
#ifdef _MSC_VER
  // Windows-specific API:
  return aligned_malloc(size, static_cast<size_t>(align));
#else
  // standard C++17 API:
  return std::aligned_alloc(static_cast<size_t>(align), size);
#endif
}

static void operator delete (void* p, std::align_val_t align) {
  ...
#ifdef _MSC_VER
  // Windows-specific API:
  _aligned_free(p);
#else
  // standard C++17 API:
  std::free(p);
#endif
}
```

Note that all allocation functions take the alignment parameter as type `size_t`, which means that we have to use the static cast to convert the value from type `std::align_val_t`.

In addition, you might want to declare the `operator new()` overloads with the `[[nodiscard]]` attribute:[5]

```
[[nodiscard]] static void* operator new (std::size_t size) {
  ...
}

[[nodiscard]] static void* operator new (std::size_t size,
                                         std::align_val_t align) {
  ...
}
```

[5] In C++20 the default implementations of `operator new()` will have these attribute.

It is rare but (as you can see here) possible to call `operator new()` directly (not using a `new` expression). With `[[nodiscard]]` compilers will detect then, if the caller forgot to use the return value, which would result in a memory leak.

When is `operator new()` Called?

As introduced, we now can have two overloads of `operator new()`:

- The version with only the `size` argument, which is also supported before C++17, is in general provided for requests of default-aligned data.

  However, it can also serve as fallback, if a version for over-aligned data is not provided.
- The version with the additional `align` argument, which has special support since C++17, is in general provided for requests of over-aligned data.

Which overload is used does *not* necessarily depend on whether `alignas` is used. It depends on the platform-specific definition of over-aligned data.

A compiler switches from default to over alignment according to a general alignment value, which you can find in the new preprocessor constant

```
__STDCPP_DEFAULT_NEW_ALIGNMENT__
```

That is, with any alignment larger than this constant a call of `new` switches from trying to call

```
operator new(std::size_t)
```

to a trial to call

```
operator new(std::size_t, std::align_val_t)
```

As a consequence, the output of the following code might vary from platform to platform:

```
struct alignas(32) MyType32 {
  ...
  static void* operator new (std::size_t size) {
    std::cout << "MyType32::new() with size " << size << '\n';
    return ::operator new(size);
  }
  static void* operator new (std::size_t size, std::align_val_t align) {
    std::cout << "MyType32::new() with size " << size
              << " and alignment " << static_cast<std::size_t>(align) << '\n';
    return ::operator new(size, align);
    ::operator delete(p);
  }
  ...
};

auto p = new MyType32;
```

If the default alignment is 32 (or less and the code compiles), the expression `new MyType32` will call the first overload of `operator new()` with only the size parameter, so that the output is something like:[6]

```
MyType32::new() with size 128
```

If the default alignment is less than 32, the second overload of `operator new()` for two arguments will be called, so that the output becomes something like:

```
MyType32::new() with size 128 and alignment 32
```

Type-Specific Fallbacks

If the `std::align_val_t` overloads are not provided for a type-specific `operator new()`, the overloads without this argument are used as fallbacks. Thus, a class that does only provide `operator new()` overloads supported before C++17 still compiles and has the same behavior (note that for the global `operator new()` this is not the case):

```
struct NonalignedNewOnly {
  ...
  static void* operator new (std::size_t size) {
    ...
  }
  ...   // no operator new(std::size_t, std::align_val_t align)
};

auto p = new NonalignedNewOnly;  // OK: operator new(size_t) used
```

The opposite is not true. If a type only provides the overloads with the alignment parameter, any trial to allocate storage with `new` using the default alignment will fail:

```
struct AlignedNewOnly {
  ...   // no operator new(std::size_t)
  static void* operator new (std::size_t size, std::align_val_t align) {
    return std::aligned_alloc(static_cast<size_t>(align), size);
  }
};

auto p = new AlignedNewOnly;  // ERROR: no operator new() for default alignment
```

It would also be an error, if for the type an alignment is requested that is (less than) the default alignment.

[6] The size might vary depending on how big an `int` and a `std::string` is on the platform.

Requesting an Alignment in the new expression

If you pass an requested alignment in the new expression, the passed alignment argument is always passed and has to be supported by the operator new(). In fact, alignment arguments are handled as any other additional argument you can pass to new expressions: They are passed as they are as additional parameter to operator new().

Thus, a call such as:

```
std::string* p = new(std::align_val_t{64}) std::string;   // 64-bytes aligned
```

will ***always*** try to call:

```
operator new(std::size_t, std::align_val_t)
```

A size-only overload would *not* serve as a fallback here.

If you have a specific alignment request for an over-aligned type, the behavior is even more interesting. If, for example, you call:

```
MyType32* p = new(std::align_val_t{64}) MyType32{};
```

and MyType32 is over-aligned, the compiler first tries to call

```
operator new(std::size_t, std::align_val_t, std::align_val_t)
```

with 32 as the second argument (the general over-alignment of the type) and 64 as third argument (the requested specific alignment). Only as a fallback,

```
operator new(std::size_t, std::align_val_t)
```

is called with 64 as the requested specific alignment. In principle, you could provide an overload for the three arguments to implement specific behavior when for over-aligned types a specific alignment is requested.

Again note that if you need special deallocation functions for over-aligned data, you have to call the right deallocation function when passing the alignment in the new expression:

```
std::string* p1 = new(std::align_val_t{64}) std::string{};
MyType32* p2 = new(std::align_val_t{64}) MyType32{};
...
::operator delete(p2, std::align_val_t{64});          // !!!
MyType32::operator delete(p1, std::align_val_t{64});  // !!!
```

This means that the new expressions in this example will call

```
operator new(std::size_t size, std::align_val_t align);
```

while the delete expressions will call one of the following two operations for default-aligned data:

```
operator delete(void* ptr, std::align_val_t align);
operator delete(void* ptr, std::size_t size, std::align_val_t align);
```

and one of the following four operations for over-aligned data:

```
operator delete(void* ptr, std::align_val_t typealign, std::align_val_t align);
operator delete(void* ptr, std::size_t size, std::align_val_t typealign,
                                             std::align_val_t align);
```

```
operator delete(void* ptr, std::align_val_t align);
operator delete(void* ptr, std::size_t size, std::align_val_t align);
```

25.3 Implementing Global `operator new()`

By default, a C++ platform now provide a significant number of global overloads for `operator new()` and `delete()` (including the corresponding array versions):

```
void* ::operator new(std::size_t);
void* ::operator new(std::size_t, std::align_val_t);
void* ::operator new(std::size_t, const std::nothrow_t&) noexcept;
void* ::operator new(std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;

void ::operator delete(void*) noexcept;
void ::operator delete(void*, std::size_t) noexcept;
void ::operator delete(void*, std::align_val_t) noexcept;
void ::operator delete(void*, std::size_t, std::align_val_t) noexcept;
void ::operator delete(void*, const std::nothrow_t&) noexcept;
void ::operator delete(void*, std::align_val_t, const std::nothrow_t&) noexcept;

void* ::operator new[](std::size_t);
void* ::operator new[](std::size_t, std::align_val_t);
void* ::operator new[](std::size_t, const std::nothrow_t&) noexcept;
void* ::operator new[](std::size_t, std::align_val_t, const std::nothrow_t&) noexcept;

void ::operator delete[](void*) noexcept;
void ::operator delete[](void*, std::size_t) noexcept;
void ::operator delete[](void*, std::align_val_t) noexcept;
void ::operator delete[](void*, std::size_t, std::align_val_t) noexcept;
void ::operator delete[](void*, const std::nothrow_t&) noexcept;
void ::operator delete[](void*, std::align_val_t, const std::nothrow_t&) noexcept;
```

If you want to implement your own memory management (e.g., to be able to debug dynamic memory calls), you don't have to override them all. It is enough to implement the following basic functions, because by default all other functions (including all array versions) call one of these basic functions:

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

In principle, the default of the sized versions of `operator delete()` also just call the unsized versions. However, this might change in future, therefore it is required that you implement both (some compiler warn if you don't do it).

25.3.1 Backward Incompatibilities

Note that the behavior of the following program silently changes with C++17:

lang/alignednewincomp.cpp

```
#include <cstddef>   // for std::size_t
#include <cstdlib>   // for std::malloc()
#include <cstdio>    // for std::printf()

void* operator new (std::size_t size)
{
  std::printf("::new called with size: %zu\n", size);
  return ::std::malloc(size);
}

int main()
{
  struct alignas(64) S {
    int i;
  };

  S* p = new S;  // calls our operator new only before C++17
}
```

In C++14, the global `::operator new(size_t)` overload was called for all `new` expressions so that the program always had the following output:[7]

```
::new called with size: 64
```

Since C++17, the behavior of this program changes, because now the default overload the over-aligned data

```
::operator new(size_t, align_val_t)
```

is called here, which is *not* replaced. As a result, the program will no longer output the line above.[8]

Note that this problem only applies to the global `operator new()`. If the type specific `operator new()` is defined for `S`, the operator is still also used as a fallback for over-aligned data so that such a program behaves as before C++17.

[7] There might be additional output for other initializations allocating memory on the heap.

[8] Some compilers warn before C++17 about calling `new` for over-aligned data, because as introduced the alignment was not handled properly before C++17.

Note also that `printf()` is used intentionally here to avoid that an output to `std::cout` allocates memory while we are allocating memory, which might result in nasty errors (core dumps at best).

25.4 Tracking all ::new Calls

The following program demonstrates how to use the new `operator new()` overloads in combination with inline variables and `[[nodiscard]]` to track all calls of `::new` just by including this header file:

lang/tracknew.hpp

```
#ifndef TRACKNEW_HPP
#define TRACKNEW_HPP

#include <new>        // for std::align_val_t
#include <cstdio>     // for printf()
#include <cstdlib>    // for malloc() and aligned_alloc()
#ifdef _MSC_VER
#include <malloc.h>   // for _aligned_malloc() and _aligned_free()
#endif

class TrackNew {
 private:
  static inline int numMalloc = 0;     // num malloc calls
  static inline size_t sumSize = 0;    // bytes allocated so far
  static inline bool doTrace = false; // tracing enabled
  static inline bool inNew = false;    // don't track output inside new overloads
 public:
  static void reset() {                // reset new/memory counters
    numMalloc = 0;
    sumSize = 0;
  }

  static void trace(bool b) {          // enable/disable tracing
    doTrace = b;
  }

  // implementation of tracked allocation:
  static void* allocate(std::size_t size, std::size_t align,
                        const char* call) {
    // track and trace the allocation:
    ++numMalloc;
    sumSize += size;
    void* p;
```

```
    if (align == 0) {
      p = std::malloc(size);
    }
    else {
#ifdef _MSC_VER
      p = _aligned_malloc(size, align);      // Windows API
#else
      p = std::aligned_alloc(align, size);   // C++17 API
#endif
    }
    if (doTrace) {
      // DON'T use std::cout here because it might allocate memory
      // while we are allocating memory (core dump at best)
      printf("#%d %s ", numMalloc, call);
      printf("(%zu bytes, ", size);
      if (align > 0) {
        printf("%zu-bytes aligned) ", align);
      }
      else {
        printf("def-aligned) ");
      }
      printf("=> %p (total: %zu Bytes)\n", (void*)p, sumSize);
    }
    return p;
  }

  static void status() {               // print current state
    printf("%d allocations for %zu bytes\n", numMalloc, sumSize);
  }
};

[[nodiscard]]
void* operator new (std::size_t size) {
  return TrackNew::allocate(size, 0, "::new");
}

[[nodiscard]]
void* operator new (std::size_t size, std::align_val_t align) {
  return TrackNew::allocate(size, static_cast<size_t>(align),
                            "::new aligned");
}

[[nodiscard]]
void* operator new[] (std::size_t size) {
```

```
  return TrackNew::allocate(size, 0, "::new[]");
}

[[nodiscard]]
void* operator new[] (std::size_t size, std::align_val_t align) {
  return TrackNew::allocate(size, static_cast<size_t>(align),
                            "::new[] aligned");
}

// ensure deallocations match:
void operator delete (void* p) noexcept {
  std::free(p);
}
void operator delete (void* p, std::size_t) noexcept {
  ::operator delete(p);
}
void operator delete (void* p, std::align_val_t) noexcept {
#ifdef _MSC_VER
  _aligned_free(p);  // Windows API
#else
  std::free(p);      // C++17 API
#endif
}
void operator delete (void* p, std::size_t,
                              std::align_val_t align) noexcept {
  ::operator delete(p, align);
}

#endif // TRACKNEW_HPP
```

Consider using this header file in the following CPP file:

lang/tracknew.cpp

```
#include "tracknew.hpp"
#include <iostream>
#include <string>

int main()
{
  TrackNew::reset();
  TrackNew::trace(true);
  std::string s = "string value with 26 chars";
  auto p1 = new std::string{"an initial value with even 35 chars"};
```

```
  auto p2 = new(std::align_val_t{64}) std::string[4];
  auto p3 = new std::string[4] { "7 chars", "x", "or 11 chars",
                                 "a string value with 28 chars" };
  TrackNew::status();
  ...
  delete p1;
  delete[] p2;
  delete[] p3;
}
```

The output depends on when the tracking is initialized and how many allocations are performed for other initializations. But it should contain something like the following lines:

```
#1 ::new (27 bytes, def-aligned) => 0x8002ccc0 (total: 27 Bytes)
#2 ::new (24 bytes, def-aligned) => 0x8004cd28 (total: 51 Bytes)
#3 ::new (36 bytes, def-aligned) => 0x8004cd48 (total: 87 Bytes)
#4 ::new[] aligned (100 bytes, 64-bytes aligned) => 0x8004cd80 (total: 187 Bytes)
#5 ::new[] (100 bytes, def-aligned) => 0x8004cde8 (total: 287 Bytes)
#6 ::new (29 bytes, def-aligned) => 0x8004ce50 (total: 316 Bytes)
6 allocations for 316 bytes
```

The first output is, for example, to initialize the memory for the value of `s`. Note that the value might be larger depending on the allocation strategy of the `std::string` class.

The next two lines written are cause by the second request:

```
auto p1 = new std::string{"an initial value with even 35 chars"};
```

It allocates 24 bytes for the core string object plus 36 bytes for the initial value of the string (again, the values might differ).

The third call requests an 64-bytes array of 4 strings.

The final call again performs two allocations: one for the array and one for the initial value of the last string. Yes, only for the last string because this implementations uses the *small/short string optimization* (SSO), which stores strings usually up to 15 characters in data members instead of allocating heap memory at all. Other implementations might perform 5 allocations here.

25.5 Afternotes

Alignment for heap/dynamic memory allocation was first proposed by Clark Nelson in https://wg21.link/n3396. The finally accepted wording was formulated by Clark Nelson in https://wg21.link/p0035r4.

Chapter 26

Other Library Improvements for Experts

There are some further improvements to the C++ standard library for experts such as foundation library programmers, which are described in this chapter.

26.1 Low-Level Conversions between Character Sequences and Numeric Values

Converting integral values to character sequences and vice versa has been an issue since C. While C provides `sprintf()` and `sscanf()`, C++ first introduced string streams, which however need a lot of resources. With C++11 convenient functions such as `std::to_string` and `std::stoi()` were introduced, which take `std::string` arguments only.

C++17 introduced new elementary string conversion functions with the following abilities (as quoted from the initial proposal):

- No runtime parsing of format strings
- No dynamic memory allocation inherently required by the interface
- No consideration of locales
- No indirection through function pointers required
- Prevention of buffer overruns
- When parsing a string, errors are distinguishable from valid numbers
- When parsing a string, whitespace or decorations are not silently ignored

In addition for floating-point numbers, this feature will provide a round-trip guarantee that values converted to a character sequence and converted back result in the original value.

The functions are provided in header file `<charconv>`.[1]

26.1.1 Example Usage

Two overloaded functions are provided:

- `std::to_char()` converts numeric values to a given character sequence.
- `from_chars()` converts a given character sequence to a numeric value.

`from_chars()`

For example:

```
#include <charconv>

const char* str = "12 monkeys";
int value;
std::from_chars_result res = std::from_chars(str, str+10,
                                             value);
```

After a successful parsing `value` contains the parsed value (12 in this example). The result value is the following structure:[2]

```
struct from_chars_result {
  const char* ptr;
  std::errc ec;
};
```

After the call, `ptr` refers to the first character not parsed as part of the number (or the passed second argument, if all characters were passed) and `ec` contains an error condition of type `std::errc` or is equal to `std::errc{}` if the conversion was successful. Thus, you can check the result as follows:

```
if (res.ec != std::errc{}) {
  ...   // error handling
}
```

Note that there is no implicit conversion to `bool` for `std::errc` so that you can't check the value as follows:

```
if (res.ec) {    // ERROR: no implicit conversion to bool
```

or:

---

[1] Note that the accepted wording for C++17 first added them to `<utility`, which was changed via a defect report after C++17 was standardized, because this created circular dependencies (see https://wg21.link/p0682r1).

[2] Note that the accepted wording for C++17 declared `ec` as a `std::error_code` which was also changed via a defect report after C++17 was standardized (see https://wg21.link/p0682r1).

```
if (!res.ec) {   // ERROR: no operator! defined
```

However, by using structured bindings and `if` with Initialization you can write:

```
if (auto [ptr, ec] = std::from_chars(str, str+10, value); ec != std::errc{}) {
  ...   // error handling
}
```

Another example is the parsing of a passed string view.

`to_chars()`

For example:

```
#include <charconv>

int value = 42;
char str[10];
std::to_chars_result res = std::to_chars(str, str+10,
                                         value);
```

After a successful conversion `str` contains the character sequence representing the passed value (`42` in this example). The result value is the following structure:[3]

```
struct to_chars_result {
  char* ptr;
  std::errc ec;
};
```

After the call, `ptr` refers to the character after the last written character and `ec` contains an error condition of type `std::errc` or is equal to `std::errc{}` if the conversion was successful. Thus, you can check the result as follows:

```
if (res.ec != std::errc{}) {
  ...   // error handling
}
else {
  process (str, res.ptr - str);   // pass characters and length
}
```

Note again that there is no implicit conversion to `bool` for `std::errc` so that you can't check the value as follows:

```
if (res.ec) {    // ERROR: no implicit conversion to bool
```

or:

```
if (!res.ec) {   // ERROR: no operator! defined
```

[3] Note that the accepted wording for C++17 declared `ec` as a `std::error_code` which was also changed via a defect report after C++17 was standardized (see https://wg21.link/p0682r1).

Again, by using structured bindings and `if` with Initialization you can write:

```
if (auto [ptr, ec] = std::to_chars(str, str+10, value); ec != std::errc{}) {
  ...  // error handling
}
else {
  process (str, res.ptr - str);  // pass characters and length
}
```

Note that this behavior is safer and easier to implement using the existing `std::to_string()` function. Using `std::to_char()` only makes sense if further processing just directly needs the written character sequence.

26.2 Afternotes

Low-Level Conversions between character sequences and numeric values were first proposed by Jens Maurer in https://wg21.link/p0067r0. The finally accepted wording was formulated by Jens Maurer in https://wg21.link/p0067r5. However, significant clarifications and a new header file was assigned as a defect report against C++17 by Jens Maurer in https://wg21.link/p0682r1.

Glossary

This glossary is a short description of the most important non-trivial technical terms that are used in this book.

B

bitmask type
A scoped enumeration type (`enum class`), for which only the bit operators are defined. You need a `static_cast<>()` to use its integral value or use it as a Boolean value.

F

full specialization
An alternative definition for a (*primary*) template, which no longer depends on any template parameter.

I

incomplete type
A class that is declared but not defined, an array of unknown size, an enumeration type without the underlying type defined, `void` (optionally with `const` and/or `volatile`), or an array of incomplete element type.

P

partial specialization
An alternative definition for a (*primary*) template, which still depends on one or more template parameters.

V

variable template
A templified variable. It allows us to define variables or static members by substituting the template parameters by specific types or values.

variadic template
A template with a template parameter that represents an arbitrary number of types or values.

Index

B

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

This page is intentionally left blank