Modern C++ Programming Cookbook

Second Edition

Master C++ core language and standard library features, with over 100 recipes, updated to C++20

Marius Bancila

Packt

BIRMINGHAM - MUMBAI

Modern C++ Programming Cookbook

Second Edition

Producer: Ben Renow-Clarke
Acquisition Editor – Peer Reviews: Suresh Jain
Project Editors: Carol Lewis and Tom Jacob
Content Development Editor: Alex Patterson
Copy Editor: Safis Editing
Technical Editor: Saby D'silva
Proofreader: Safis Editing
Indexer: Priyanka Dhadke
Presentation Designer: Sandip Tadge

First published: May 2017
Second Edition: September 2020

Production reference: 1090920

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-80020-898-8

www.packt.com

Contributors

About the author

Marius Bancila is a software engineer with almost two decades of experience in developing solutions for the industrial and financial sectors. He is the author of *The Modern C++ Challenge* and coauthor of *Learn C# Programming*. He works as a software architect and is focused on Microsoft technologies, mainly developing desktop applications with C++ and C#, but not solely. He is passionate about sharing his technical expertise with others and, for that reason, he has been recognized as a Microsoft MVP for C++ and later developer technologies since 2006.

I would like to thank all the people at Packt that worked on this project and helped to make a better book updated with the latest C++ changes. To Carol, Alex, Tom, and Saby for their efforts and coordination of the project. And to my family for their support during the time spent writing this book.

About the reviewer

Steve Oualline wrote his first program at age 11. He's been programming ever since. He has worked at a variety of programming jobs since then.

Table of Contents

Preface

C++ is one of the most popular and most widely used programming languages, and it has been like that for three decades. Designed with a focus on performance, efficiency, and flexibility, C++ combines paradigms such as object-oriented, imperative, generic, and functional programming. C++ is standardized by the **International Organization for Standardization** (**ISO**) and has undergone massive changes over the last decade. With the standardization of C++11, the language has entered into a new age, which has been widely referred to as modern C++. Type inference, move semantics, lambda expressions, smart pointers, uniform initialization, variadic templates, and many other recent features have changed the way we write code in C++ to the point that it almost looks like a new programming language. This change is being further advanced with the release of the C++20 standard that is supposed to happen during 2020. The new standard includes many new changes to the language, such as modules, concepts, and coroutines, as well as to the standard library, such as ranges, text formatting, and calendars.

This book addresses many of the new features included in C++11, C++14, C++17, and the forthcoming C++20. This book is organized in recipes, each covering one particular language or library feature, or a common problem that developers face and its typical solution using modern C++. Through more than 130 recipes, you will learn to master both core language features and the standard libraries, including those for strings, containers, algorithms, iterators, streams, regular expressions, threads, filesystem, atomic operations, utilities, and ranges.

This second edition of the book took several months to write, and during this time the work on the C++20 standard has been completed. However, at the time of writing this preface, the standard is yet to be approved and will be published later this year.

More than 30 new or updated recipes in this book cover C++20 features, including modules, concepts, coroutines, ranges, threads and synchronization mechanisms, text formatting, calendars and time zones, immediate functions, the three-way comparison operator, and the new `span` class.

All the recipes in the book contain code samples that show how to use a feature or how to solve a problem. These code samples have been written using Visual Studio 2019, but have been also compiled using Clang and GCC. Since the support for various language and library features has been gradually added to all these compilers, it is recommended that you use the latest version to ensure that all of them are supported. At the time of writing this preface, the latest versions are GCC 10.1, Clang 12.0 (in progress), and VC++ 2019 version 14.27 (from Visual Studio 2019 version 16.7). Although all these compilers are C++17 complete, the support for C++20 varies from compiler to compiler. Please refer to `https://en.cppreference.com/w/cpp/compiler_support` to check your compiler's support for C++20 features.

Who this book is for

This book is intended for all C++ developers, regardless of their experience level. The typical reader is an entry- or medium-level C++ developer who wants to master the language and become a prolific modern C++ developer. The experienced C++ developer will find a good reference for many C++11, C++14, C++17, and C++20 language and library features that may come in handy from time to time. The book consists of more than 130 recipes that are simple, intermediate, or advanced. However, they all require prior knowledge of C++, and that includes functions, classes, templates, namespaces, macros, and others. Therefore, if you are not familiar with the language, it is recommended that you first read an introductory book to familiarize yourself with the core aspects, and then proceed with this book.

What this book covers

Chapter 1, Learning Modern Core Language Features, teaches you about modern core language features, including type inference, uniform initialization, scoped enumerations, range-based for loops, structured bindings, class template argument deduction, and others.

Chapter 2, Working with Numbers and Strings, discusses how to convert between numbers and strings, generate pseudo-random numbers, work with regular expressions and various types of string, as well as how to format text using the C++20 text formatting library.

Chapter 3, Exploring Functions, dives into defaulted and deleted functions, variadic templates, lambda expressions, and higher-order functions.

Chapter 4, Preprocessing and Compilation, takes a look at various aspects of compilation, from how to perform conditional compilation, to compile-time assertions, code generation, and hinting the compiler with attributes.

Chapter 5, Standard Library Containers, Algorithms, and Iterators, introduces you to several standard containers, many algorithms, and teaches you how to write your own random-access iterator.

Chapter 6, General-Purpose Utilities, dives into the `chrono` library, including the C++20 calendars and time zones support; the `any`, `optional`, `variant`, and `span` types; and type traits.

Chapter 7, Working with Files and Streams, explains how to read and write data to/from streams, use I/O manipulators to control streams, and explores the `filesystem` library.

Chapter 8, Leveraging Threading and Concurrency, teaches you how to work with threads, mutexes, locks, condition variables, promises, futures, atomic types, as well as the C++20 latches, barriers, and semaphores.

Chapter 9, Robustness and Performance, focuses on exceptions, constant correctness, type casts, smart pointers, and move semantics.

Chapter 10, Implementing Patterns and Idioms, covers various useful patterns and idioms, such as the pimpl idiom, the non-virtual interface idiom, and the curiously recurring template pattern.

Chapter 11, Exploring Testing Frameworks, gives you a kickstart with three of the most widely used testing frameworks, Boost.Test, Google Test, and Catch2.

Chapter 12, C++20 Core Features, introduces you to the most important new additions to the C++20 standard—modules, concepts, coroutines, and ranges.

To get the most out of this book

The code presented in the book is available for download from `https://github.com/PacktPublishing/Modern-Cpp-Cookbook-Second-Edition`, although I encourage you to try writing all the samples by yourself. In order to compile them, you need VC++ 2019 16.7 on Windows and GCC 10.1 or Clang 12.0 on Linux and Mac. If you don't have the latest version of the compiler, or you want to try another compiler, you can use one that is available online.

Although there are various online platforms that you could use, I recommend Wandbox, available at `https://wandbox.org/`, and Compiler Explorer, available at `https://godbolt.org/`.

Download the example code files

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `http://www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Modern-CPP-Programming-Cookbook-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781800208988_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "The geometry module was defined in a file called `geometry.ixx/.cppm`, although any file name would have had the same result."

A block of code is set as follows:

```
static std::map<
  std::string,
  std::function<std::unique_ptr<Image>()>> mapping
{
  { "bmp", []() {return std::make_unique<BitmapImage>(); } },
  { "png", []() {return std::make_unique<PngImage>(); } },
  { "jpg", []() {return std::make_unique<JpgImage>(); } }
};
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are highlighted:

```
static std::map<
  std::string,
  std::function<std::unique_ptr<Image>()>> mapping
{
  { "bmp", []() {return std::make_unique<BitmapImage>(); } },
  { "png", []() {return std::make_unique<PngImage>(); } },
  { "jpg", []() {return std::make_unique<JpgImage>(); } }
};
```

Any command-line input or output is written as follows:

```
running thread 140296854550272
running thread 140296846157568
running thread 140296837764864
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com`, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, `packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1

Learning Modern Core Language Features

The C++ language has gone through a major transformation in the past decade with the development and release of C++11 and then, later, with its newer versions: C++14, C++17, and C++20. These new standards have introduced new concepts, simplified and extended existing syntax and semantics, and overall transformed the way we write code. C++11 looks like a new language, and code written using the new standards is called modern C++ code.

The recipes included in this chapter are as follows:

- Using `auto` whenever possible
- Creating type aliases and alias templates
- Understanding uniform initialization
- Understanding the various forms of non-static member initialization
- Controlling and querying object alignment
- Using scoped enumerations
- Using `override` and `final` for virtual methods
- Using range-based for loops to iterate on a range
- Enabling range-based for loops for custom types
- Using explicit constructors and conversion operators to avoid implicit conversion
- Using unnamed namespaces instead of static globals

- Using inline namespaces for symbol versioning
- Using structured bindings to handle multi-return values
- Simplifying code with class template argument deduction

Let's start by learning about automatic type deduction.

Using auto whenever possible

Automatic type deduction is one of the most important and widely used features in modern C++. The new C++ standards have made it possible to use `auto` as a placeholder for types in various contexts and let the compiler deduce the actual type. In C++11, `auto` can be used for declaring local variables and for the return type of a function with a trailing return type. In C++14, `auto` can be used for the return type of a function without specifying a trailing type and for parameter declarations in lambda expressions. Future standard versions are likely to expand the use of `auto` to even more cases. The use of `auto` in these contexts has several important benefits, all of which will be discussed in the *How it works...* section. Developers should be aware of them, and prefer `auto` whenever possible. An actual term was coined for this by Andrei Alexandrescu and promoted by Herb Sutter—**almost always auto** (**AAA**).

How to do it...

Consider using `auto` as a placeholder for the actual type in the following situations:

- To declare local variables with the form `auto name = expression` when you do not want to commit to a specific type:

```
auto i = 42;          // int
auto d = 42.5;        // double
auto s = "text";      // char const *
auto v = { 1, 2, 3 }; // std::initializer_list<int>
```

- To declare local variables with the `auto name = type-id { expression }` form when you need to commit to a specific type:

```
auto b  = new char[10]{ 0 };           // char*
auto s1 = std::string {"text"};        // std::string
auto v1 = std::vector<int> { 1, 2, 3 }; // std::vector<int>
auto p  = std::make_shared<int>(42);   // std::shared_ptr<int>
```

- To declare named lambda functions, with the form `auto name = lambda-expression`, unless the lambda needs to be passed or returned to a function:

```
auto upper = [](char const c) {return toupper(c); };
```

- To declare lambda parameters and return values:

```
auto add = [](auto const a, auto const b) {return a + b;};
```

- To declare a function return type when you don't want to commit to a specific type:

```
template <typename F, typename T>
auto apply(F&& f, T value)
{
  return f(value);
}
```

How it works...

The `auto` specifier is basically a placeholder for an actual type. When using `auto`, the compiler deduces the actual type from the following instances:

- From the type of expression used to initialize a variable, when `auto` is used to declare variables.
- From the trailing return type or the type of the return expression of a function, when `auto` is used as a placeholder for the return type of a function.

In some cases, it is necessary to commit to a specific type. For instance, in the first example in the previous section, the compiler deduces the type of `s` to be `char const *`. If the intention was to have an `std::string`, then the type must be specified explicitly. Similarly, the type of `v` was deduced as `std::initializer_list<int>`. However, the intention could be to have an `std::vector<int>`. In such cases, the type must be specified explicitly on the right side of the assignment.

There are some important benefits of using the `auto` specifier instead of actual types; the following is a list of, perhaps, the most important ones:

- It is not possible to leave a variable uninitialized. This is a common mistake that developers make when declaring variables specifying the actual type. However, this is not possible with `auto`, which requires an initialization of the variable in order to deduce the type.
- Using `auto` ensures that you always use the correct type and that implicit conversion will not occur. Consider the following example where we retrieve the size of a vector to a local variable. In the first case, the type of the variable is `int`, though the `size()` method returns `size_t`. This means an implicit conversion from `size_t` to `int` will occur. However, using `auto` for the type will deduce the correct type; that is, `size_t`:

```
auto v = std::vector<int>{ 1, 2, 3 };
```

```
// implicit conversion, possible loss of data
int size1 = v.size();

// OK
auto size2 = v.size();

// ill-formed (warning in gcc/clang, error in VC++)
auto size3 = int{ v.size() };
```

- Using `auto` promotes good object-oriented practices, such as preferring interfaces over implementations. The fewer the number of types specified, the more generic the code is and more open to future changes, which is a fundamental principle of object-oriented programming.
- It means less typing and less concern for actual types that we don't care about anyway. It is very often the case that even though we explicitly specify the type, we don't actually care about it. A very common case is with iterators, but there are many more. When you want to iterate over a range, you don't care about the actual type of the iterator. You are only interested in the iterator itself; so, using `auto` saves time used for typing possibly long names and helps you focus on actual code and not type names. In the following example, in the first `for` loop, we explicitly use the type of the iterator. It is a lot of text to type; the long statements can actually make the code less readable, and you also need to know the type name that you actually don't care about. The second loop with the `auto` specifier looks simpler and saves you from typing and caring about actual types:

```
std::map<int, std::string> m;
for (std::map<int, std::string>::const_iterator
  it = m.cbegin();
  it != m.cend(); ++it)
{ /*...*/ }

for (auto it = m.cbegin(); it != m.cend(); ++it)
{ /*...*/ }
```

- Declaring variables with `auto` provides a consistent coding style with the type always in the right-hand side. If you allocate objects dynamically, you need to write the type both on the left and right side of the assignment, for example, `int* p = new int(42)`. With `auto`, the type is specified only once on the right side.

However, there are some gotchas when using auto:

- The auto specifier is only a placeholder for the type, not for the const/volatile and references specifiers. If you need a const/volatile and/or reference type, then you need to specify them explicitly. In the following example, foo.get() returns a reference to int; when the variable x is initialized from the return value, the type deduced by the compiler is int, not int&. Therefore, any change made to x will not propagate to foo.x_. In order to do so, we should use auto&:

```
class foo {
  int x_;
public:
  foo(int const x = 0) :x_{ x } {}
  int& get() { return x_; }
};

foo f(42);
auto x = f.get();
x = 100;
std::cout << f.get() << '\n'; // prints 42
```

- It is not possible to use auto for types that are not moveable:

```
auto ai = std::atomic<int>(42); // error
```

- It is not possible to use auto for multi-word types, such as long long, long double, or struct foo. However, in the first case, the possible workarounds are to use literals or type aliases; as for the second, using struct/class in that form is only supported in C++ for C compatibility and should be avoided anyway:

```
auto l1 = long long{ 42 }; // error

using llong = long long;
auto l2 = llong{ 42 };     // OK
auto l3 = 42LL;            // OK
```

- If you use the auto specifier but still need to know the type, you can do so in most IDEs by putting the cursor over a variable, for instance. If you leave the IDE, however, that is not possible anymore, and the only way to know the actual type is to deduce it yourself from the initialization expression, which could mean searching through the code for function return types.

The `auto` can be used to specify the return type from a function. In C++11, this requires a trailing return type in the function declaration. In C++14, this has been relaxed, and the type of the return value is deduced by the compiler from the `return` expression. If there are multiple return values, they should have the same type:

```
// C++11
auto func1(int const i) -> int
{ return 2*i; }

// C++14
auto func2(int const i)
{ return 2*i; }
```

As mentioned earlier, `auto` does not retain `const`/`volatile` and reference qualifiers. This leads to problems with `auto` as a placeholder for the return type from a function. To explain this, let's consider the preceding example with `foo.get()`. This time, we have a wrapper function called `proxy_get()` that takes a reference to a `foo`, calls `get()`, and returns the value returned by `get()`, which is an `int&`. However, the compiler will deduce the return type of `proxy_get()` as being `int`, not `int&`. Trying to assign that value to an `int&` fails with an error:

```
class foo
{
  int x_;
public:
  foo(int const x = 0) :x_{ x } {}
  int& get() { return x_; }
};

auto proxy_get(foo& f) { return f.get(); }

auto f = foo{ 42 };
auto& x = proxy_get(f); // cannot convert from 'int' to 'int &'
```

To fix this, we need to actually return `auto&`. However, this is a problem with templates and perfect forwarding the return type without knowing whether it is a value or a reference. The solution to this problem in C++14 is `decltype(auto)`, which will correctly deduce the type:

```
decltype(auto) proxy_get(foo& f) { return f.get(); }
auto f = foo{ 42 };
decltype(auto) x = proxy_get(f);
```

The `decltype` specifier is used to inspect the declared type of an entity or an expression. It's mostly useful when declaring types are cumbersome or not possible at all to declare with the standard notation. Examples of this include declaring lambda types and types that depend on template parameters.

The last important case where `auto` can be used is with lambdas. As of C++14, both lambda return types and lambda parameter types can be `auto`. Such a lambda is called a *generic lambda* because the closure type defined by the lambda has a templated call operator. The following shows a generic lambda that takes two `auto` parameters and returns the result of applying `operator+` to the actual types:

```
auto ladd = [] (auto const a, auto const b) { return a + b; };
struct
{
  template<typename T, typename U>
  auto operator () (T const a, U const b) const { return a+b; }
} L;
```

This lambda can be used to add anything for which the `operator+` is defined, as shown in the following snippet:

```
auto i = ladd(40, 2);            // 42
auto s = ladd("forty"s, "two"s); // "fortytwo"s
```

In this example, we used the `ladd` lambda to add two integers and to concatenate to `std::string` objects (using the C++14 user-defined literal `operator ""s`).

See also

- *Creating type aliases and alias templates* to learn about aliases for types
- *Understanding uniform initialization* to see how brace-initialization works

Creating type aliases and alias templates

In C++, it is possible to create synonyms that can be used instead of a type name. This is achieved by creating a `typedef` declaration. This is useful in several cases, such as creating shorter or more meaningful names for a type or names for function pointers. However, `typedef` declarations cannot be used with templates to create template type aliases. An `std::vector<T>`, for instance, is not a type (`std::vector<int>` is a type), but a sort of family of all types that can be created when the type placeholder `T` is replaced with an actual type.

In C++11, a type alias is a name for another already declared type, and an alias template is a name for another already declared template. Both of these types of aliases are introduced with a new `using` syntax.

How to do it...

- Create type aliases with the form `using identifier = type-id`, as in the following examples:

```
using byte     = unsigned char;
using byte_ptr = unsigned char *;
using array_t  = int[10];
using fn       = void(byte, double);

void func(byte b, double d) { '*...*/ }

byte b{42};
byte_ptr pb = new byte[10] {0};
array_t a{0,1,2,3,4,5,6,7,8,9};
fn* f = func;
```

- Create alias templates with the form `template<template-params-list> identifier = type-id`, as in the following examples:

```
template <class T>
class custom_allocator { /* ... */ };

template <typename T>
using vec_t = std::vector<T, custom_allocator<T>>;

vec_t<int>           vi;
vec_t<std::string>   vs;
```

For consistency and readability, you should do the following:

- Not mix `typedef` and `using` declarations when creating aliases
- Prefer the `using` syntax to create names of function pointer types

How it works...

A `typedef` declaration introduces a synonym (an alias, in other words) for a type. It does not introduce another type (like a `class`, `struct`, `union`, or `enum` declaration). Type names introduced with a `typedef` declaration follow the same hiding rules as identifier names. They can also be redeclared, but only to refer to the same type (therefore, you can have valid multiple `typedef` declarations that introduce the same type name synonym in a translation unit, as long as it is a synonym for the same type). The following are typical examples of `typedef` declarations:

```
typedef unsigned char   byte;
typedef unsigned char * byte_ptr;
typedef int             array_t[10];
typedef void(*fn)(byte, double);

template<typename T>
class foo {
  typedef T value_type;
};

typedef std::vector<int> vint_t;
```

A type alias declaration is equivalent to a `typedef` declaration. It can appear in a block scope, class scope, or namespace scope. According to C++11 paragraph 7.1.3.2:

> *"A typedef-name can also be introduced by an alias declaration. The identifier following the using keyword becomes a typedef-name and the optional attribute-specifier-seq following the identifier appertains to that typedef-name. It has the same semantics as if it were introduced by the typedef specifier. In particular, it does not define a new type and it shall not appear in the type-id."*

An alias declaration is, however, more readable and clearer about the actual type that is aliased when it comes to creating aliases for array types and function pointer types. In the examples from the *How to do it...* section, it is easily understandable that `array_t` is a name for the type array of 10 integers, while `fn` is a name for a function type that takes two parameters of the type `byte` and `double` and returns `void`. This is also consistent with the syntax for declaring `std::function` objects (for example, `std::function<void(byte, double)> f`).

It is important to take note of the following things:

- Alias templates cannot be partially or explicitly specialized.
- Alias templates are never deduced by template argument deduction when deducing a template parameter.
- The type produced when specializing an alias template is not allowed to directly or indirectly make use of its own type.

The driving purpose of the new syntax is to define alias templates. These are templates that, when specialized, are equivalent to the result of substituting the template arguments of the alias template for the template parameters in the type-id.

See also

- *Simplifying code with class template argument deduction* to learn how to use class templates without explicitly specifying template arguments

Understanding uniform initialization

Brace-initialization is a uniform method for initializing data in C++11. For this reason, it is also called *uniform initialization*. It is arguably one of the most important features from C++11 that developers should understand and use. It removes previous distinctions between initializing fundamental types, aggregate and non-aggregate types, and arrays and standard containers.

Getting ready

To continue with this recipe, you need to be familiar with direct initialization, which initializes an object from an explicit set of constructor arguments, and copy initialization, which initializes an object from another object. The following is a simple example of both types of initialization:

```
std::string s1("test");   // direct initialization
std::string s2 = "test";  // copy initialization
```

With these in mind, let's explore how to perform uniform initialization.

How to do it...

To uniformly initialize objects regardless of their type, use the brace-initialization form {}, which can be used for both direct initialization and copy initialization. When used with brace-initialization, these are called direct-list and copy-list-initialization:

```
T object {other};   // direct-list-initialization
T object = {other}; // copy-list-initialization
```

Examples of uniform initialization are as follows:

- Standard containers:

```
std::vector<int> v { 1, 2, 3 };
std::map<int, std::string> m { {1, "one"}, { 2, "two" }};
```

- Dynamically allocated arrays:

```
int* arr2 = new int[3]{ 1, 2, 3 };
```

- Arrays:

```
int arr1[3] { 1, 2, 3 };
```

- Built-in types:

```
int i { 42 };
double d { 1.2 };
```

- User-defined types:

```
class foo
{
  int a_;
  double b_;
public:
  foo():a_(0), b_(0) {}
  foo(int a, double b = 0.0):a_(a), b_(b) {}
};

foo f1{};
foo f2{ 42, 1.2 };
foo f3{ 42 };
```

- User-defined POD types:

```
struct bar { int a_; double b_;};
bar b{ 42, 1.2 };
```

How it works...

Before C++11, objects required different types of initialization based on their type:

- Fundamental types could be initialized using assignment:

```
int a = 42;
double b = 1.2;
```

- Class objects could also be initialized using assignment from a single value if they had a conversion constructor (prior to C++11, a constructor with a single parameter was called a *conversion constructor*):

```
class foo
{
  int a_;
public:
  foo(int a):a_(a) {}
};
foo f1 = 42;
```

- Non-aggregate classes could be initialized with parentheses (the functional form) when arguments were provided and only without any parentheses when default initialization was performed (call to the default constructor). In the next example, `foo` is the structure defined in the *How to do it...* section:

```
foo f1;          // default initialization
foo f2(42, 1.2);
foo f3(42);
foo f4();        // function declaration
```

- Aggregate and POD types could be initialized with brace-initialization. In the following example, `bar` is the structure defined in the *How to do it...* section:

```
bar b = {42, 1.2};
int a[] = {1, 2, 3, 4, 5};
```

A **Plain Old Data** (POD) type is a type that is both trivial (has special members that are compiler-provided or explicitly defaulted and occupy a contiguous memory area) and has a standard layout (a class that does not contain language features, such as virtual functions, which are incompatible with the C language, and all members have the same access control). The concept of POD types has been deprecated in C++20 in favor of trivial and standard layout types.

Apart from the different methods of initializing the data, there are also some limitations. For instance, the only way to initialize a standard container (apart from copy constructing) is to first declare an object and then insert elements into it; `std::vector` was an exception because it is possible to assign values from an array that can be initialized prior using aggregate initialization. On the other hand, however, dynamically allocated aggregates could not be initialized directly.

All the examples in the *How to do it...* section use direct initialization, but copy initialization is also possible with brace-initialization. These two forms, direct and copy initialization, may be equivalent in most cases, but copy initialization is less permissive because it does not consider explicit constructors in its implicit conversion sequence, which must produce an object directly from the initializer, whereas direct initialization expects an implicit conversion from the initializer to an argument of the constructor. Dynamically allocated arrays can only be initialized using direct initialization.

Of the classes shown in the preceding examples, `foo` is the one class that has both a default constructor and a constructor with parameters. To use the default constructor to perform default initialization, we need to use empty braces; that is, `{}`. To use the constructor with parameters, we need to provide the values for all the arguments in braces `{}`. Unlike non-aggregate types, where default initialization means invoking the default constructor, for aggregate types, default initialization means initializing with zeros.

Initialization of standard containers, such as the vector and the map, also shown previously, is possible because all standard containers have an additional constructor in C++11 that takes an argument of the type `std::initializer_list<T>`. This is basically a lightweight proxy over an array of elements of the type `T const`. These constructors then initialize the internal data from the values in the initializer list.

The way initialization using `std::initializer_list` works is as follows:

- The compiler resolves the types of the elements in the initialization list (all the elements must have the same type).
- The compiler creates an array with the elements in the initializer list.
- The compiler creates an `std::initializer_list<T>` object to wrap the previously created array.
- The `std::initializer_list<T>` object is passed as an argument to the constructor.

An initializer list always takes precedence over other constructors where brace-initialization is used. If such a constructor exists for a class, it will be called when brace-initialization is performed:

```
class foo
{
  int a_;
  int b_;
public:
  foo() :a_(0), b_(0) {}

  foo(int a, int b = 0) :a_(a), b_(b) {}
  foo(std::initializer_list<int> l) {}
};

foo f{ 1, 2 }; // calls constructor with initializer_list<int>
```

The precedence rule applies to any function, not just constructors. In the following example, two overloads of the same function exist. Calling the function with an initializer list resolves to a call to the overload with an `std::initializer_list`:

```
void func(int const a, int const b, int const c)
{
  std::cout << a << b << c << '\n';
}

void func(std::initializer_list<int> const list)
{
  for (auto const & e : list)
    std::cout << e << '\n';
}

func({ 1,2,3 }); // calls second overload
```

This, however, has the potential of leading to bugs. Let's take, for example, the `std::vector` type. Among the constructors of the vector, there is one that has a single argument, representing the initial number of elements to be allocated, and another one that has an `std::initializer_list` as an argument. If the intention is to create a vector with a preallocated size, using brace-initialization will not work as the constructor with the `std::initializer_list` will be the best overload to be called:

```
std::vector<int> v {5};
```

The preceding code does not create a vector with five elements, but a vector with one element with a value of `5`. To be able to actually create a vector with five elements, initialization with the parentheses form must be used:

```
std::vector<int> v (5);
```

Another thing to note is that brace-initialization does not allow narrowing conversion. According to the C++ standard (refer to paragraph 8.5.4 of the standard), a narrowing conversion is an implicit conversion:

> "*- From a floating-point type to an integer type.*
>
> *- From long double to double or float, or from double to float, except where the source is a constant expression and the actual value after conversion is within the range of values that can be represented (even if it cannot be represented exactly).*
>
> *- From an integer type or unscoped enumeration type to a floating-point type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted to its original type.*
>
> *- From an integer type or unscoped enumeration type to an integer type that cannot represent all the values of the original type, except where the source is a constant expression and the actual value after conversion will fit into the target type and will produce the original value when converted to its original type.*"

The following declarations trigger compiler errors because they require a narrowing conversion:

```
int i{ 1.2 };          // error

double d = 47 / 13;
float f1{ d };         // error
```