BARTŁOMIEJ FILIPEK

C++ INITIALIZATION STORY

A Guide Through All Initialization Options and Related C++ Areas

{C++ STORIES}

C++ Initialization Story

A Guide Through All Initialization Options and Related C++ Areas

Bartłomiej Filipek

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

This version was published on 2023-06-19

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

For Wiola, Mikołaj and Feliks

Contents

About the Book

Initialization in C++ is a hot topic! The internet is full of discussions about best practices, and there are even funny memes on that subject. The situation is not surprising, as there are more than a dozen ways to initialize a simple integer value, complex rules for the auto-type deduction, data members, and object lifetime nuances.

And here comes the book.

Throughout this text, you will learn practical options to initialize various categories of variables and data members in Modern C++. More specifically, this text teaches multiple types of initialization, constructors, non-static data member initialization, inline variables, designated initializers, and more. Additionally, you'll see the changes and new techniques from C++11 to C++20 and lots of examples to round out your understanding.

The plan is to explain most (if not all) parts of initialization, learn lots of excellent C++ techniques, and see what happens under the hood.

Why should you read this book?

With Modern C++ (since C++11), we have many new features to streamline work and simplify our code. One area of improvement is initialization. Modern C++ added new initialization rules, trying to make it easy while keeping old behavior and compatibility (mainly from the C language). Sometimes the rules might seem confusing and complex, though, and even the ISO committee might need to correct some things along the way. The book will help you navigate through those principles and understand this topic better. What's more, initialization is just one aspect of this text. You'll learn all related topics around classes, constructors, destructors, object lifetime, or even how the compiler processes data at start-up.

Learning objectives

The goal is to equip you with the following knowledge:

- Explain rules about object initialization, including regular variables, data members, and non-local objects.

- How to implement special member functions (constructors, destructors, copy/move operations) and when they are helpful.
- How to efficiently initialize non-static data members using C++11 features like non-static data member initialization, inheriting, and delegating constructors.
- How to streamline working with static variables and static data members with inline variables from C++17.
- How to work with container-like members, non-copyable data members (like `const` data members) or move-able only data members, or even lambdas.
- What is an aggregate, and how to create such objects with designated initializers from C++20.

The structure of the book

The book contains 14 chapters in the following structure:

- Chapters 1 to 5 create a foundation for the rest of the book. They cover basic initialization rules, constructors, destructors, and the basics of data members.
- Chapter 6 describes type deduction that can be used to declare objects: `auto`, `decltype`, Almost Always Auto rule, structured bindings.
- Chapter 7 is a quiz with 10 questions from the first "part" of the book.
- Chapter 8 describes Non-static Data Member Initialization (NSDMI), a powerful feature from C++11 that improves how we work with data members. At the end of the chapter, you can solve a few exercises.
- Chapter 9 discusses how to initialize container-like data members.
- Chapter 10 contains information about non-regular data members and how to handle them in a class. You'll learn about `const` data members, `unique_ptr` as a data member, and references.
- Chapter 11 describes static non-local variables, static objects, various storage duration options, and `inline` variables from C++17 and `constinit` from C++20.
- Chapter 12 moves to C++20 and describes Designated Initializers, a handy feature based on similar thing from the C language.
- Chapter 13 shows various techniques like passing strings into constructors, strong typing, CRTP class counter, Copy and Swap idiom, and more.
- Chapter 14 is the final quiz with questions from the whole book.

And there are two appendices:

- Appendix A - a handy guide about rules for compiler-generated special member functions.
- Appendix B - answers to quizzes and exercises.

Who is this book for?

The book is intended for beginner/intermediate C++ programmers who want to learn various aspects of initialization in Modern C++ (from C++11 to C++20).

You should know at least some of the basics of creating and using custom classes.

This text is also helpful for experienced programmers who know older C++ standards and want to move into C++17/C++20.

Prerequisites

- You should have basic knowledge of C++ expressions and primitive types.
- You should be able to implement an elementary class with several data members. Know how to create and manipulate objects of such a class in a basic way.

Reader feedback & errata

If you spot an error, a typo, a grammar mistake, or anything else (especially logical issues!) that should be corrected, please send your feedback to bartek@cppstories.com or submit an issue at github.com/fenbf/cppinitbook_public/issues[1].

Here's the errata with the list of fixes:

www.cppstories.com/p/cppinitbook/[2]

Your feedback matters! Writing an honest review can help with the book promotion and the quality of my further work. The book has a dedicated page at GoodReads. Please share your feedback at:

C++ Initialization Story by Bartłomiej Filipek @Goodreads[3].

Or write a review at Amazon if you get this book in print form.

[1] https://github.com/fenbf/cppinitbook_public/issues
[2] https://www.cppstories.com/p/cppinitbook/
[3] https://www.goodreads.com/book/show/62606823-c-initialization-story

Example code

You can find source code of all examples in this separate GitHub public repository.

https://github.com/fenbf/cppinitbook_public/tree/main/examples

You can browse individual files or download the whole branch:

https://github.com/fenbf/cppinitbook_public/archive/refs/heads/main.zip

Code license

The code for the book is available under the MIT License model.

Formatting & Layout

Code samples are presented in a monospaced font, similar to the following example:

Title Of the Example

```
#include <iostream>

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

Or shorter snippets (without a title and sometimes `include` statements):

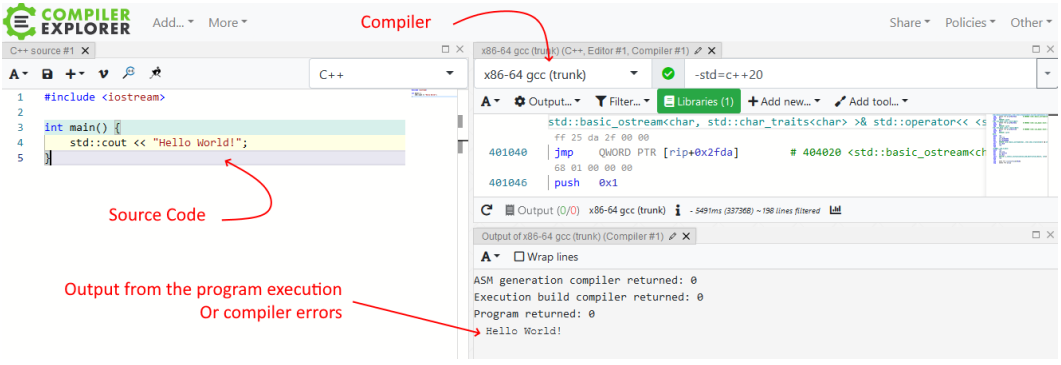
```
int foo() {
    return std::clamp(100, 1000, 1001);
}
```

When available, you'll also see a link to online compilers where you can play with the code. For example:

Example title. Run @Compiler Explorer

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
}
```

You can click on the link in the title, and then it should open the website of a given online compiler (in the above case, it's Compiler Explorer). You can compile the sample, see the output, and experiment with the code directly in your browser. Here's a basic overview of Compiler Explorer:



A Compiler Explorer layout used in the book

Snippets of longer programs might be shortened to present only the core mechanics. They may lack some `#include` statements or have a "compressed" line. Click on the Online Compiler link to see the full version of the program or see them in the public repository.

Recommendation for Compiler Explorer and C++ Reference

When executing the examples on Compiler Explorer, you may select a term (keyword, class, function, container or other) and then right-click (or equivalent). A context menu will appear and you can select `Search on CppReference`, which will take you to the C++ Reference documentation of the corresponding browser language, if such C++ Reference version exists[4].

[4]Thanks to Javier Estrada for suggesting this cool tip!

Syntax highlighting limitations

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

For example:

- The first method of a class is not highlighted - First method of class not highlighted in C++ · Issue #791[5].
- Template method is not highlighted C++ lexer doesn't recognize function if return type is templated · Issue #1138[6].
- Modern C++ attributes are sometimes not appropriately recognized.

Other issues for C++ and Pygments: C++ Issues · github/pygments/pygments[7].

Special sections

Throughout the book, you can also see the following sections:

This is an Information Box with extra notes related to the current section.

This is a Warning Box with potential risks and threats related to a given topic.

> This is a Quote Box. In the book, it's often used to quote the C++ Standard.

[5] https://github.com/pygments/pygments/issues/791
[6] https://github.com/pygments/pygments/issues/1138
[7] https://github.com/pygments/pygments/issues?q=is%3Aissue+is%3Aopen+C%2B%2B

About the Author

Bartłomiej (Bartek) Filipek is a C++ software developer from the beautiful city of Cracow in Southern Poland. He started his professional career in 2007 and in 2010 he graduated from Jagiellonian University with a Masters Degree in Computer Science.

Bartek currently works at Xara[8], where he develops features for advanced document editors. He also has experience with desktop graphics applications, game development, large-scale systems for aviation, writing graphics drivers and even biofeedback. In the past, Bartek has also taught programming (mostly game and graphics programming courses) at local universities in Cracow.

Since 2011 Bartek has been regularly blogging at cppstories.com[9] (started as bfilipek.com[10]). The blog focuses on core C++ and getting up-to-date with the C++ Standards. He's also a co-organiser of the C++ User Group in Cracow[11]. You can hear Bartek in one @CppCast episode[12] where he talks about C++17, blogging and text processing.

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

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

Bartek is the author of C++17 In Detail[13] and C++ Lambda Story[14].

[8] http://www.xara.com/
[9] https://www.cppstories.com]
[10] https://www.bfilipek.com
[11] https://www.meetup.com/C-User-Group-Cracow/
[12] http://cppcast.com/2018/04/bartlomiej-filipek/
[13] https://leanpub.com/cpp17indetail
[14] https://leanpub.com/cpplambda

Acknowledgements

This book wouldn't be possible without valuable input from many C++ experts and friends.

I especially would like to thank to the following people:

- JFT (John Taylor),
- Mariusz Jaskółka,
- Florin Chertes (see his profile at LinkedIn[15]),
- Konrad Jaśkowiec (see his profile at LinkedIn[16]),
- Professor Boguslaw Cyganek (see his profile at AGH university page[17]),
- Dawid Pilarski (see his blog at panicsoftware.com[18]),
- Javier Estrada (see his blog at javierestrada.blog[19]),
- Jonathan Boccara (from fluentcpp.com[20]),
- Andreas Fertig (see his blog at andreasfertig.blog[21]),
- Peter Sommerlad (see his website and training info at sommerlad.ch[22]),
- Timur Doumler (see his website at timur.audio[23] and his Twitter[24]),
- Michael Goldshteyn, Software Architect.

They spent a lot of time on finding even little things that could be improved and extended.

Last but not least, I got a lot of valuable feedback from my blog readers, Patreon Discord Server (See C++Stories @Patreon[25]), and discussions at C++ Polska[26]. Thank you all!

With all of the help from those kind people, the book quality got better and better!

[15] https://www.linkedin.com/in/florin-ioan-chertes-41b6845/
[16] https://pl.linkedin.com/in/konrad-ja%C5%9Bkowiec-84585159
[17] https://home.agh.edu.pl/~cyganek/
[18] https://blog.panicsoftware.com/
[19] https://javierestrada.blog/
[20] https://www.fluentcpp.com/
[21] https://andreasfertig.blog/
[22] https://sommerlad.ch/
[23] https://timur.audio/
[24] https://twitter.com/timur_audio
[25] https://www.patreon.com/cppstories
[26] https://cpp-polska.pl/

Revision History

- 20th June 2022 - The first public version! Missing parts: some sections in 10. Containers as Data Members, some sections in 11. Non-regular Data Members.
- 22nd June 2022 - new sections on NSDMI, direct init and parens, more about inheriting constructors, link to GoodReads, wording, hotfixes.
- 24th June 2022 - updated the "copy and move constructor" chapter, typos and small wording improvements.
- 16th July 2022 - Containers as Data Members chapter rewritten, `noexcept` consistency and `noexcept` move operations advantages in the move constructor section, wording, fixes, layout.
- 13th September 2022 - changed title to "C++ Initialization Story", adapted book structure, rewritten "Non-local objects" chapter (previously only on `inline` variables), new extracted chapter on Techniques, new section on CRTP.
- 18th November 2022 - heavily updated and completed "Non-regular data members" chapter, `constinit` and `thread_local` sections in the "Non-local objects" chapter, filled the "implicit conversion" section in the Constructors chapter.
- 23rd December 2022 - content completed! Added Deduction chapter, filled missing sections in the Techniques chapter. Layout improvements, a few more questions, exercises and fixes.
- 4th February 2023 - updated layout, wording improvements, fixes, improve consistency of examples.
- 27th February 2023 - the first edition in Print!
- 13th April 2023 - the final quiz questions 6th and 8th clarification, small layout fixes.
- 19th June 2023 - a batch of minor fixes and clarifications based on user feedback.

1. Local Variables and Simple Types

Let's start simple and ask, "what is initialization?" When we go to the definition from C++Reference[1], we can read:

> *Initialization* of a variable provides its initial value at the time of construction.

We can translate this definition to the following example:

```
void foo() {
    int x = 42;
    // ... use 'x' later...
}
```

Above, we have a function with a local variable `x`. The variable is declared as `integer` and initialized with the value `42`. This is not the only way you can assign that initial value. Here are some more options:

```
struct Point { int x; int y; };           // declare a custom type
Point createPoint(int x) { return {x, -x}; }
int main() {
    int x { 42 };                         // list initialization
    double y = { 100.0 };                 // copy list initialization
    auto ptr = std::make_unique<float>(90.5f); // auto type deduction
    auto z = createPoint(42);             // through a factory function
    std::string s (10, 'x');              // calling a constructor
    Point p { 10 };                       // aggregate initialization
    std::array<float, 100> numbers { 1.1f, 2.2f }; // array initialization
    // ...
}
```

[1] https://en.cppreference.com/w/cpp/language/initialization

You can also come up with many other forms of setting a value. We can also extend the syntax on class data members, `static` variables, thread locals, or even dynamic memory allocations.

In theory, initialization is a simple task: "put a value into a memory location of a newly created variable". However, such action relates to many different parts of an application (local vs. non-local scope) and various places in the memory (like stack vs. heap). That's why the syntax or the behavior might be slightly different.

In C++, we have at least the following forms of initialization:

- aggregate initialization
- constant initialization
- default initialization
- direct initialization
- copy initialization
- list initialization
- reference initialization
- value initialization
- zero initialization
- plus related topics like copy elision, static variables, conversion sequences, constructors, assignment, dynamic memory, storage, and more.

While the list sounds complex, we'll move through those topics step by step revealing core concepts. Later we'll address more advanced examples and see what happens inside the C++ machinery.

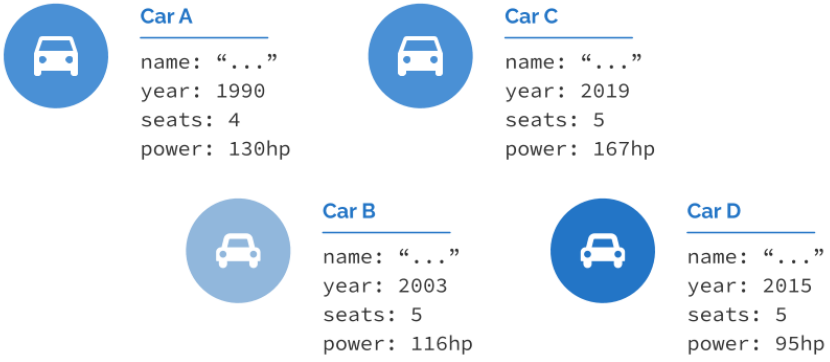
While we can explain most cases on integers and other numerical types, it's best to work on something more practical. The book starts with some elementary custom types, then considers various issues we might have with their early implementations. Later the types will expand, giving us more context and compelling use cases.

Starting with simple types

Defining a class or a struct (a custom type) in C++ allows you to model your problem domain and solve problems more naturally. Rather than working with a bunch of variables and functions, it's best to group them and provide a consistent API (Application Programming

Interface). C++ provides a set of built-in types, including boolean, integral, character, and floating-point. Additionally, you can use objects from the Standard Library, like various collections, `std::string`, `std::vector`, `std::map`, `std::set`, and many others. You can collect these essential components and build your types.

To create a background for our main topic, let's start with a type representing Car Information for a car listing app. A system reads the car/truck information from a database and displays it in the application. For an easy start, the type holds four members: name (a `std::string`), production year, number of seats, and engine power.



Below there's the first version of the code for that `CarInfo` type:

Ex 1.1. Simple `CarInfo` structure. Run @Compiler Explorer

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

struct CarInfo {
    std::string name;
    unsigned year;
    unsigned seats;
    double power;
};

int main() {
    CarInfo firstCar;
    firstCar.name = "Renault Megane";
    firstCar.year = 2003;
    firstCar.seats = 5;
```

```
    firstCar.power = 116;
    std::cout << "name: " << firstCar.name << '\n';
    std::cout << "year: " << firstCar.year << '\n';
    std::cout << "seats: " << firstCar.seats << '\n';
    std::cout << "power (hp): " << firstCar.power << '\n';
}
```

In the above example, we defined a simple structure that holds data for a `CarInfo`. The code is super simple, contains some issues, and follows the style of C++03. In the following few chapters, I'll guide you through the code and help you understand the problems and how to eliminate them. We'll also modernize it to include the latest C++ (up to C++20) features.

First: `name`, `year`, `seats` and `power` are called *non-static data members*. Each instance of the `CarInfo` class has its own set of those members. In other words, we group variables to create a representation for models in our problem domain. A user-defined type might also have *static data members*, which are data shared between all instances of a given type. For example, we could imagine a *static* member variable called `numAllCars` that would indicate the total number of cars created in our program. We'll talk about *static data members* later in chapter 11 Static Variables.

Now, let's investigate the code in detail. The definition and the declaration of the variable `firstCar` in the `main()` function:

```
CarInfo firstCar;
```

It is called *default initialization* and, since our `struct` is simple, will leave all data members of built-in types with *indeterminate values*. Similarly, you can get the same (potentially buggy effect) for simple types when declared in function (as such variables have automatic storage duration) [2]:

```
void foo() {
    int i;     // indeterminate value!
    double d;  // indeterminate value!
}
```

The `std::string` data member `name`, on the other hand, will have an empty state (an empty string) because its default constructor will be called. More on that later.

[2] In contrast, `static` and thread-local objects will be zero-initialized.

Once the object is created and uninitialized, we can access its members and set proper values. By default, `struct` has public access to its members (and `class` has private access). This way, we can access and change their values directly.

What is "Automatic Storage Duration" ?

All objects in a program have four possible ways to be "stored": automatic, static, thread, or dynamic. Automatic means that the storage is allocated at the start of the scope, like in a function. Most local variables have automatic storage duration (except those declared as `static`, `extern`, or `thread_local`). We'll talk about this more in the separate chapter on non-local objects[3].

Setting values to zero

You might feel very unsatisfied that after creating a `CarInfo` object, most data members have some indeterminate values. We can fix this and make sure data is at least set to "zero". Have a look:

Ex 1.2. Value initialization for `CarInfo` structure. Run @Compiler Explorer

```
CarInfo emptyCar{};
std::cout << "name: " << emptyCar.name << '\n';
std::cout << "year: " << emptyCar.year << '\n';
std::cout << "seats: " << emptyCar.seats << '\n';
std::cout << "power (hp): " << emptyCar.power << '\n';
```

The output:

```
name:
year: 0
seats: 0
power (hp): 0
```

The initialization with empty braces `{}` is called *value initialization* and by default (for built-in types and classes with default constructors that are neither user-provided nor deleted), sets data to "zero" (adapted for different types). This is similar to declaring and defining the following variables:

[3] chapterinlinevars

```
int i{};     // i == 0
double d{};  // d == 0.0
std::string s{}; // s is an empty string

int j = {}; // other form of value initialization
std::string str = {}; // ...
```

This time the storage duration doesn't matter, and value initialization works the same for static, dynamic, thread-local, or automatic variables. For types with default constructors (more on that later), the code will call them and, in the case of `string s;` will initialize it to an empty string.

Initialization with aggregates

Our structure is very simple, and for such types, C++ has special rules where we can initialize their internal values with so-called *aggregate initialization*. We can use such syntax also for arrays. Here are some basic examples:

Ex 1.3. Aggregate Initialization basic syntax. Run @Compiler Explorer

```
// arrays:
int arr[] { 1, 2, 3, 4 };
float numbers[] = { 0.1f, 1.1f, 2.2f, 3.f, 4.f, 5. };
int nums[10] { 1 }; // 1, and then all 0s

// structures:
struct Point { int x; int y; };
struct Line { Point p1; Point p2; };
Line longLone {0, 0, 100, 100};
Line anotherLine = {100}; // rest set to 0
Line shortLine {{-10, -10}, {10, 10}}; // nested
```

In summary, for the above code:

- Each array element, or non-static class member, in order of array subscript/appearance in the class definition, is copy-initialized from the corresponding clause of the initializer list.

- You can use list initialization for arrays, and when the number of elements is not provided, the compiler will deduce the count.
- If you pass fewer elements in the initializer list than the number of elements in the array, the remaining elements will be value initialized. For built-in types, it means the value of zero.
- For structures, you can use a single initializer list or nested one; the expansion will be recursive.
- If you provide fewer values than the number of data members in the aggregate, then the remaining data members (in the declaration order) will be effectively value initialized.

The first bullet point says that each element is *copy initialized*. We'll return to this topic and explain the difference between a copy vs. direct initialization syntax once we know explicit constructors.

For our structure, we can write the following test code:

Ex 1.4. Aggregate initialization for the `CarInfo` structure. Run @Compiler Explorer

```
struct CarInfo {
    std::string name;
    unsigned year;
    unsigned seats;
    double power;
};

void printInfo(const CarInfo& c) {
    std::cout << c.name << ", "
              << c.year << " year, "
              << c.seats << " seats, "
              << c.power << " hp\n";
}

int main() {
    CarInfo firstCar{"Megane", 2003, 5, 116 };
    printInfo(firstCar);
    CarInfo partial{"unknown"};
    printInfo(partial);
    CarInfo largeCar{"large car", 1975, 10};
    printInfo(largeCar);
}
```

This will output:

```
Megane, 2003 year, 5 seats, 116 hp
unknown, 0 year, 0 seats, 0 hp
large car, 1975 year, 10 seats, 0 hp
```

To give you the full picture, as of C++20, here's the definition of an *aggregate type* from the C++ Standard: dcl.init.aggr[4].

> An aggregate is an array or a class type with:
>
> - no user-provided, explicit, or inherited constructors
> - no private or protected non-static data members
> - no virtual functions, and
> - no virtual, private, or protected base classes

Don't worry if you're not familiar with all of the cases listed above. We'll discuss them along the way and see more aggregates in the further parts. There's also a dedicated chapter about Aggregates and Designated Initialization in C++20.

Default data member initialization

What if you want to provide some default value for your data member? With value initialization, you can get zeros for various types, but sometimes it might not be good enough.

Since C++14, we can leverage Non-static Data Member Initializers (NSDMI), also called Default Member Initializers, to provide default values for aggregates. Have a look:

[4] https://timsong-cpp.github.io/cppwp/n4868/dcl.init.aggr#:initialization,aggregate

Ex 1.5. Default member initialization and aggregates. Run @Compiler Explorer

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

struct CarInfo {
    std::string name { "unknown" };
    unsigned year { 1920 };
    unsigned seats { 4 };
    double power { 100. };
};

void printInfo(const CarInfo& c) { /* */ }

int main() {
    CarInfo unknown;
    printInfo(unknown);
    CarInfo zeroed{};
    printInfo(zeroed);
    CarInfo partial{"large car", 1975};
    printInfo(partial);
}
```

This will print:

```
unknown, 1920 year, 4 seats, 100 hp
unknown, 1920 year, 4 seats, 100 hp
large car, 1975 year, 4 seats, 100 hp
```

The syntax is quite intuitive; you can initialize a data member at the place where it's declared. This can prevent accidental bugs where your data has some indeterminate value. As you can see from the example, even if you use default initialization or value initialization, data members will get values that were provided in the `struct` declaration. If you give fewer values in the aggregate initializer, the remaining members will get their defaults from the declaration.

Technically, in-class member initializers have been available since C++11, but aggregate types weren't supported initially. In this section, we've only scratched the surface of this handy technique. See the dedicated chapter for this topic: Non-static data member initialization chapter.

Summary

In this chapter, we covered some simple custom types and looked at ways to initialize their data members. We went from objects with indeterminate values to zero initialization, and then we learned about aggregates and techniques to provide default values.

Things to keep in mind:

- Default initialization for objects and variables yields indeterminate values for built-in types or default-initialize complex types (like `std::string` and set it to an empty string). That's why it's essential to **be sure your objects and simple variables are always initialized**.
- Value initialization like `int x{};` for built-in types effectively yields zero initialization for them so that they will be zero (in their type).
- With value initialization `CarInfo car{};` all data members will be zero-initialized (for built-in types) or default initialized for complex types.
- Aggregates are simple types or arrays with all public data members; we can initialize them with an aggregate initialization syntax.
- Thanks to the in-class member initializer feature, you can provide default values for your data members.

What's next?

While simple types are handy, in C++, we often need to build large objects where data members depend on each other or have invariants. In such cases, it's best to hide them behind member functions and give access to them under certain conditions. That's why in the next chapter, we'll look at `class`'s and **constructors**. We'll also expand the knowledge that we got so far.

2. Classes and Initialization With Constructors

In the previous chapter, you've seen that C++ might treat simple structures with all public data members as an aggregate class. Still, aggregates might not be enough if we want better data encapsulation and a more complex class API. For full flexibility in C++, we can leverage constructors that are special member functions invoked when an object is created.

A simple class type

As a background example, let's create a type that will hold some elementary network data. To complicate things, we'd like to compute a basic checksum for the data part. Such a checksum might be handy for checking if the data was transferred correctly across the Internet (read more @Wikipedia[1]).

Ex 2.1. Simple `DataPacket` class. Run @Compiler Explorer

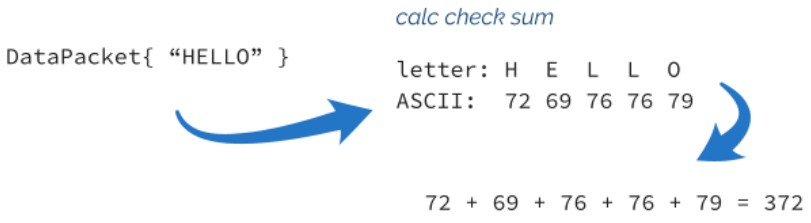
```
#include <iostream>
#include <numeric>

size_t calcCheckSum(const std::string& s) {
    return std::accumulate(s.begin(), s.end(), static_cast<size_t>(0));
}

class DataPacket {
private:
    std::string data_;
    size_t checkSum_;
    size_t serverId_;

public:
    const std::string& getData() const { return data_; }
```

[1] https://en.wikipedia.org/wiki/Checksum

```
    void setData(const std::string& data) {
        data_ = data;
        checkSum_ = calcCheckSum(data);
    }
    size_t getCheckSum() const { return checkSum_; }
    size_t getServerId() const { return serverId_; }
    void setServerId(size_t serverId) { serverId_ = serverId; }
};
```

The class above contains three *non-static data members*: `data_`, `checkSum_` and `serverID_`. I'm using the underscore suffix to indicate private data members, a common practice in many codebases. See Google C++ Style Guide[2].

To keep things simple, I implemented the `calcCheckSum` function in terms of `std::accumulate()`, which is an algorithm from the C++ Standard Library. This code starts from `0` (we can use `0UZ` since C++23 instead of explicit `static_cast`) and adds numerical values of letters from the input `std::string`. For example, for `"HELLO"`, we'll get the following computations:



Calculating simple checksum for a string

`DataPacket` has so-called getters and setters - functions that return or change a particular data member. For example `getData()` returns the `data_` data member, while `setData(...)` allows to change it.

One important topic is that getters usually have `const` applied at the end. This means that a given member function is constant and cannot change the value of the members (unless they are `mutable`). If you have a `const` object, you can only call its `const` member functions. Applying `const` might improve program design as it's usually easier to reason about the state

[2] https://google.github.io/styleguide/cppguide.html#Variable_Names

of `const` instances. For more information, see this C++ core guideline: Con.2: By default, make member functions const[3].

Member functions might also have `noexcept` specifier applied. However, this topic is outside the scope of the book and won't be covered. You can find more @C++Reference - `noexcept` specifier[4].

Here's the continuation of the example where we create and use the object of the `DataPacket` class:

Ex 2.2. Simple `DataPacket` class, continuation. Run @Compiler Explorer

```
int main() {
    DataPacket packet;
    packet.setData("Programming World");
    std::cout << packet.getCheckSum() << '\n';
}
```

The code doesn't access data members directly but calls member functions to operate on the object and change its properties.

You can notice `public` and `private` parts in the class declaration. The order of those sections is just a coding convention and they group elements together based on their *access modifier*. In short, a member under the `public` keyword can be accessed from the outside (like calling a member function or accessing a data member). On the other hand, members under the `private` section cannot be accessed from [5]. In C++, you can also add `protected` to your class declaration, which means that member functions or fields are not accessible outside. Still, they are accessible to all inherited classes (assuming `public` inheritance, members become `private` outside, but `public` to derived types, see more about different inheritance options @C++Reference[6]).

For example, in the `main()` function above, I cannot write:

[3] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#con2-by-default-make-member-functions-const
[4] https://en.cppreference.com/w/cpp/language/noexcept_spec
[5] Unless accessed by `friend` functions or classes.
[6] https://en.cppreference.com/w/cpp/language/access

```
DataPacket packet;
packet.serverId = 10; // error: 'size_t DataPacket::serverId'
                      // is private within this context
```

The only difference between `class` and `struct` in C++ is that `class` has `private` as the default access modifier and `private` inheritance, while `struct` has both specified as `public`. Some C++ guidelines, for example, Google Style Guide see this link[7], suggest using `struct` only for smaller, "passive" types, with only public data members. The C++ Core Guidelines also recommend using `class` if any member is not public; see C++ Core Guidelines - C.8[8].

Since our class doesn't have any user-defined constructors (more on them in the next section), we can also use value initialization syntax to set values to zero or default values:

Ex 2.3. Value initialization for the `DataPacket` class. Run @Compiler Explorer

```
int main() {
    DataPacket packet{};
    std::cout << "data: " << packet.getData() << '\n';
    std::cout << "checkSum: " << packet.getCheckSum() << '\n';
    std::cout << "serverId: " << packet.getServerId() << '\n';
}
```

This will generate the following output:

```
data:
checkSum: 0
serverId: 0
```

However, the main difference now is that because we moved the data members to the private section, the class is **not an aggregate**. That's why we cannot use aggregate initialization to set all values at once. To fix this, we need to look at constructors. And that is the plan for further sections.

[7] https://google.github.io/styleguide/cppguide.html#Structs_vs._Classes

[8] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c8-use-class-rather-than-struct-if-any-member-is-non-public

Basics of constructors

A constructor is a special member function without a name, but we declare it using the enclosing class name. You cannot invoke a constructor like other member functions. Instead, the compiler calls it when an object of its class is being initialized. It has the following basic syntax:

```
class/struct ClassName {
  // ...
/*explicit*/ ClassName(parameter-list) = default/=delete
  : base-class-initializer
  , member-init
  { /*body*/ }
  // ...
};
```

A constructor has the following parts:

- constructor has no name, but we define it using the name of the class,
- optional `explicit` - keyword to block implicit conversions on a given class type,
- `ClassName` - the name of the given class type (they have to match),
- `parameter-list`- a list of parameters, as in a regular function, might be empty
- optional `= default`/`=delete` specifies if a constructor should be `deleted` (not present) or defaulted by the compiler,
- `:` - indicates the start of the member/base initialization list, required when `base-class-initializer` or `member-init` lists are present,
- optional `base-class-initializer` - a list of base classes' constructors that we explicitly want to call,
- optional `member-init` - a list of data members where we can directly initialize them,
- `{/*body*/}` - a function body.

You can also apply `noexcept`, `[[attributes]]`, `constexpr`, `consteval` on a constructor, but the full explanation of those additional properties goes beyond the scope of the book. Read more at C++Reference - Constructors and member initializer lists[9].

[9] https://en.cppreference.com/w/cpp/language/constructor

Let's have a look at one snippet:

```
class Product {
public:
    Product() : id_{-1}, name_{"none"} { } // a default constructor
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name} { }

private:
    int id_;
    std::string name_;
};
```

The above example shows a class `Product` with two constructors. The first one is called a *default constructor*; it has no arguments. The second one takes two arguments. As you can notice, C++ allows multiple constructors that look like overloaded functions (they differ by the number or types of arguments). Each constructor also has a regular function body where you can execute some code; in our case, they are both empty for now. I also applied the `explicit` keyword on the second constructor; we'll talk about it later.

The primary function of constructors is to perform some actions at the start of a lifetime of an object. Usually, it means data member initialization, resource allocation (opening a file, a socket, memory allocation), or even doing some special logic (like logging).

In our case, constructors touch only data members inside a special section of constructors called *member initializer list*: like, `id_{-1}, name_{"none"}`. Inside this initializer list, we can also call constructors of base classes (if any). Later, we'll address inheritance in the Inheritance section.

The *member initializer list* is more efficient than using the body of a constructor. Sometimes it's even the only option to initialize the value, as with types that are not assignable. See the following alternative:

```
class Product {
public:
    Product() { id_ = 0; name_ = "none"; }

private:
    int id_;
    std::string name_;
};
```

The code will yield the same values for data members as in the previous example, but the data members are set in two steps rather than one. With the *member initializer list* data members are set directly, same as calling: `int id_ { 0 }` or `std::string name_ {"none"}`. On the other hand, if we use assignment in the constructor body, it requires two steps:

```
// step 1: default init:
int id_; // indeterminate value!
std::string name_; // default ctor called
// step 2: assignment:
id_ = 0;
name_ = "none";
```

While this might not be a big issue for built-in simple types like `int` , you'll need some more CPU cycles for larger objects like `strings`.

There's also one important aspect about the *initializer list*: the order of initialization. This is covered in The C++ Specification: 11.10.3 Classes[10]:

> Non-static data members are initialized in the order they were declared in the class definition (regardless of the order of the mem-initializers).

When I write:

[10] https://timsong-cpp.github.io/cppwp/n4868/class.base.init#13.3

```
class Product {
public:
    Product() : name_{"none"}, id_{-1} { }

private:
    int id_;
    std::string name_;
};
```

The values will be set correctly, but the order will differ from what we think. A compiler might show us a warning in this case. Here's the warning from GCC compiled with `-Wall` option (experiment @Compiler Explorer[11]):

```
<source>: In constructor 'Product::Product()':
<source>:15:17: warning: 'Product::name_' will be initialized after [-Wreorder]
   15 |     std::string name_;
      |                 ^~~~~
<source>:14:9: warning:   'int Product::id_' [-Wreorder]
   14 |     int id_;
      |         ^~~
```

The initialization order might be critical when you imply some dependency on the values. For example, we can write the following artificial sample:

```
struct S {
    int x;
    int y;
    int z;

    S(): x{0}, y{1}, z{x+y} { }
    // S(): y{0}, z{0}, x{z+y}, { }
};
```

In the above example, the first constructor initializes `x` and `y` and then uses those values to initialize `z`. This is complicated and might be hard to read, but it works correctly. On the other hand, in the second (commented out) constructor, the order of initialization will create

[11] https://godbolt.org/z/jE77169qd

an undefined behavior for initializing `x`, as `z` and `y` won't be initialized yet. It's best to avoid such dependencies to minimize the risk of bugs.

Let's see how a constructor works by creating some objects of the `Product` class:

```
Product none;
```

In the first example, we created the `none` object, which is default constructed. The compiler will call our default constructor; thus, the data members will be initialized to `id_ = -1` and `name_ = "none"`.

```
Product car(10, "car");
```

The example uses the form of *direct initialization* which calls the constructor with two arguments. After the call data members will be: `id_ = 10` and `name_ = "car"`.

And the last example:

```
Product tvSet{100, "tv set" };
```

This time we also called a constructor with two arguments, but the syntax is called * direct list initialization* - `"{}"`. Please notice that I also used this form of initialization inside the *initializer list* in constructors.

Here's the complete example:

Ex 2.4. Constructors for the Product class. Run @Compiler Explorer

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

class Product {
public:
    Product() : id_{-1}, name_{"none"} { } // a default constructor
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name} { }

    int Id() const { return id_; }
    std::string Name() const { return name_; }
```

```
private:
    int id_;
    std::string name_;
};

int main() {
    Product none;
    std::cout << none.Id() << ", " << none.Name() << '\n';

    Product car(10, "super car");
    std::cout << car.Id() << ", " << car.Name() << '\n';

    Product tvSet{77, "tv set" };
    std::cout << tvSet.Id() << ", " << tvSet.Name() << '\n';
}
```

You might also scratch your head and ask why I declared the `name` parameter as `const std::string&` rather than just `std::string&`. First, we don't want to modify this parameter in the constructor's body. What's more, `const T&` - `const` references can bind to "temporary" objects like a string literal `"super car"`. Without a `const` reference, we would have to pass some named `string` object. Alternatively, we can pass the name by value and perform a "move operation" on that argument. Further in the book, I'll address this topic in detail, see chapter: A Use Case - Best Way to Initialize string Data Members.

More on uniform initialization

Content available in the full version of the book.

Body of a constructor

After the member initializer list, each constructor has a regular function body, `{ ... }`, where you can perform additional steps to modify variables or call other functions. The only difference between a regular function and a constructor is that a constructor cannot return any values. Typically, a constructor throws an exception to report an error.

Here's a small example that shows how to add some logging into a constructor body and throw an exception on error:

Ex 2.7. Logging in a constructor. Run @Compiler Explorer

```
#include <iostream>
#include <stdexcept> // for std::invalid_argument

constexpr int LOWEST_ID_VALUE = -100;

class Product {
public:
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name}
    {
        std::cout << "Product(): " << id_ << ", " << name_ << '\n';
        if (id_ < LOWEST_ID_VALUE)
            throw std::invalid_argument{"id lower than LOWEST_ID_VALUE!"};
    }

    std::string Name() const { return name_; }

private:
    int id_;
    std::string name_;
};

int main() {
    try {
        Product car(10, "car");
        std::cout << car.Name() << " created\n";
        Product box(-101, "box");
        std::cout << box.Name() << " created\n";
    }
    catch (const std::exception& ex) {
        std::cout << "Error - " << ex.what() << '\n';
    }
}
```

The above example shows a constructor that performs logging and basic parameter checking. It uses a `LOWEST_ID_VALUE`, a global constant marked with the `constexpr` keyword (the second time we used this keyword).

The `constexpr` specifier has been available since C++11 and guarantees that a value is available at compile time for *constant expressions*. For example, you can use such a variable to set the number of elements in a C-style array. It's often perceived as a "type-safe macro definition". The keyword applies to all built-in trivial types like integral values, floating-point, or even character literals (but not `std::string`); there's also a way to declare custom `constexpr`-ready types. You can also create a function to be `constexpr` and possibly evaluate it at compile-time; however, we won't cover such functions in this book. See more at C++Reference - `constexpr`[12].

If you run this program, you can see the following output:

```
Product(): 10, car
car created
Product(): -101, box
Error - id cannot be lower than LOWEST_ID_VALUE!
```

Please notice that while two constructors were called, we can see that only the first one succeeded. Since the constructor for `box` threw an exception, this object is not treated as fully created. More on that later, when we'll talk about destructors.

Adding constructors to `DataPacket`

After the introduction, we can start adding constructors to our `DataPacket` class.

Ex 2.8. Adding constructors. Run @Compiler Explorer

```
class DataPacket {
    std::string data_;
    size_t checkSum_;
    size_t serverId_;

public:
    DataPacket()
    : data_{}
    , checkSum_{0}
    , serverId_{0}
```

[12] https://en.cppreference.com/w/cpp/language/constexpr

```
    { }

    explicit DataPacket(const std::string& data, size_t serverId)
    : data_{data}
    , checkSum_{calcCheckSum(data)}
    , serverId_{serverId}
    { }

    const std::string& getData() const { return data_; }
    void setData(const std::string& data) {
        data_ = data;
        checkSum_ = calcCheckSum(data);
    }
    size_t getCheckSum() const { return checkSum_; }

    void setServerId(size_t id) { serverId_ = id; }
    size_t getServerId() const { return serverId_; }
};
```

And here's the demo code that creates some objects:

Ex 2.9. Adding constructors, Demo. Run @Compiler Explorer

```
void printInfo(const DataPacket& packet) {
    std::cout << "data: " << packet.getData() << '\n';
    std::cout << "checkSum: " << packet.getCheckSum() << '\n';
    std::cout << "serverId: " << packet.getServerId() << '\n';
}

int main() {
    DataPacket empty;
    printInfo(empty);
    DataPacket zeroed{};
    printInfo(zeroed);
    DataPacket packet{"Hello World", 101};
    printInfo(packet);
    DataPacket reply{"Hi, how are you?", 404};
    printInfo(reply);
}
```

The output:

```
data:
checkSum: 0
serverId: 0
data:
checkSum: 0
serverId: 0
data: Hello World
checkSum: 1052
serverId: 101
data: Hi, how are you?
checkSum: 1375
serverId: 404
```

In the above example, we used two constructors:

- The first one is a default constructor and initializes data members to default values. It will be called for default and value initialization.
- The second constructor takes several arguments and matches them with data members. This constructor makes it easy to pass parameters all at once (previously, we needed to call setters). This one takes two parameters, but we can initialize as many data members as we need. For example, the constructors ensure the `checkSum_` variable matches `data_`. Since those two members are related, thanks to constructors and the `setData` member function, we keep the relation safe.

We can also use default member initializers inside a class, but we'll address that in detail in a separate chapter.

Compiler-generated default constructors

While C++ allows you to implement various constructors, it can make your life easier by automatically declaring and defining an implicit default constructor.

In other words, if you write a class type with no default constructor:

```
class Example {
public:
    std::string Name() const { return name_; }

private:
    std::string name_;
};
```

Then the compiler will create an implicit empty constructor:

```
inline Example() noexcept { }
```

A simple rule is that if a class has no user-declared constructors, the compiler will create a default one if possible.

Have a look:

Ex 2.10. Implicit default constructor. Run @Compiler Explorer

```
struct Value {
    int x;
};

struct CtorValue {
    CtorValue(int v): x{v} { }
    int x;
};

int main() {
    Value v;          // fine, default constructor available
    // CtorValue y;      // error! no default ctor available
    CtorValue z{10}; // using custom ctor
}
```

As you can see above, the compiler will create an implicit default constructor for the `Value` class (since it has no other constructors), but it won't generate a default constructor for the `CtorValue` class. Also, notice that `Value::x` will have an indeterminate value as a default constructor is empty and won't set any value for `x`.

Default constructors only default-initialize data members, so in the case of built-in types, it means indeterminate values!

You can control the creation of such a default constructor using two keywords, `default` and `delete`. In short, `default` tells the compiler to use the default implementation, while `delete` blocks the implementation.

Ex 2.11. Default and Delete Constructors. Run @Compiler Explorer

```
struct Value {
    Value() = default;

    int x;
};

struct CtorValue {
    CtorValue() = default;
    CtorValue(int v): x{v} { }
    int x;
};

struct DeletedValue {
    DeletedValue() = delete;
    DeletedValue(int v): x{v} { }
    int x;
};

int main() {
    Value v;         // fine, default constructor available
    CtorValue y;     // ok now, default ctor available
    CtorValue z{10}; // using custom ctor
    // DeletedValue w;   // err, deleted ctor!
    DeletedValue u{10}; // using custom ctor
}
```

In the above example, you can see that we declare `Value() = default;` this tells the compiler to create an empty (doing nothing) implementation. Also, in the `CtorValue` class, we also use the same technique, and, as you can notice, the default construction works now.

The third class has `= delete` as its default constructor, and you'll get an error if you want to create an object of this class using its default constructor.

The implicit default constructor won't be created if your type has data members that are not default-constructible or inherits from a type that is not default-constructible. That includes references, `const` data members, unions, and others. See the complete list here @C++Reference[13].

You may also ask what's the difference between `Value() = default` and `Value() { }` they both are "empty". Still, according to the C++ Standard the second constructor is considered *user-declared* or *user-provided* and has some consequences in the type characteristics. We'll cover that later once we cover copy constructors in the section: Trivial classes and user-declared/user-provided default constructors.

Explicit constructors

Content available in the full version of the book.

Difference between direct and copy initialization

Content available in the full version of the book.

Even more

Content available in the full version of the book.

Constructor summary

This chapter was probably the longest, as we had to prepare the background for the rest of the book. Once you know the basics of how data members can be initialized through constructors, we can move further and explore various new C++ features and examples.

Now, it's essential to summarize two other types of constructors: copy and move. Read on to the next chapter.

[13] https://en.cppreference.com/w/cpp/language/default_constructor#Deleted_implicitly-declared_default_constructor

3. Copy and Move Constructors

Regular constructors allow you to invoke some logic and initialize data members when an object is created from a list of arguments. But C++ also has two special constructor types that let you control a situation when an object is created using an instance of the same class type. Those constructors are called copy and move constructors. Let's have a look.

Copy constructor

A copy constructor is a special member function taking an object of the same type as the first argument, usually by `const` reference.

```
ClassName(const ClassName&);
```

Technically it might have other parameters, but they all have to have default values assigned.

It's used and called when you create an object using a variable of the same type, to be precise, when you use *copy initialization*.

```
Product base { 42, "base product" }; // an initial object

// various forms of initialization, where a copy constructor is called
Product other { base };
Product another(base);
Product oneMore = base;
Product arr[] = { base, other, oneMore };
```

Implementing a copy constructor might be necessary when your class has data members that shouldn't be shallow copied, like pointers, resource ids (like file handles), etc.

A canonical implementation of a copy constructor

Implementing a copy constructor is straightforward and very similar to regular constructors. The only difference is that you have a single parameter which is a (`const`) reference to an object of that same type.

For the `Product` class, we can write the following:

```
class Product {
public:
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name}
    {
        std::cout << "Product(): " << id_ << ", " << name_ << '\n';
    }

    // copy constructor
    Product(const Product& other)
        : id_{other.id_}, name_{other.name_}
    { }

private:
    int id_;
    std::string name_;
};
```

As you can see, the copy constructor uses the member initialization list to copy the data from `other`. Please notice that there's no need to use public getters, as we have access to all private data members. The compiler requires you to use a reference, so writing `Product(Product other)` won't be treated as a copy constructor.

A copy constructor can also take a `non-const` argument like `Product(Product& other)`. However, such a constructor might modify the `other` object and might be hard to reason about the code. It might be better to use move semantics and move constructors when you want to "steal" the guts of some other object.

Here's another example where logging is enabled:

Ex 3.1. An example of a logging copy constructor. Run @Compiler Explorer

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

class Product {
public:
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name}
    {
        std::cout << "Product(): " << id_ << ", " << name_ << '\n';
    }

    Product(const Product& other)
        : id_{other.id_}, name_{other.name_}
    {
        std::cout << "Product(copy): " << id_ << ", " << name_ << '\n';
    }

    const std::string& Name() const { return name_; }

private:
    int id_;
    std::string name_;
};

int main() {
    Product base { 42, "base product" }; // an initial object
    std::cout << base.Name() << " created\n";
    std::cout << "Product other { base };\n";
    Product other { base };
    std::cout << "Product another(base);\n";
    Product another(base);
    std::cout << "Product oneMore = base;\n";
    Product oneMore = base;
    std::cout << "Product arr[] = { base, other, oneMore };\n";
    Product arr[] = { base, other, oneMore };
}
```

If you run the code, you should see the following output:

```
Product(): 42, base product
base product created
Product other { base };
Product(copy): 42, base product
Product another(base);
Product(copy): 42, base product
Product oneMore = base;
Product(copy): 42, base product
Product arr[] = { base, other, oneMore };
Product(copy): 42, base product
Product(copy): 42, base product
Product(copy): 42, base product
```

In the first line, we construct `base product`, and then use it to copy-construct all other instances.

Copy constructors can be marked with `explicit`, but this is not a common practice and might prevent copy initialization.

A compiler-generated copy constructor

Content available in the full version of the book.

Move constructor

Move constructors take rvalue references of the same type.

```
ClassName(ClassName&&);
```

In short, rvalue references are temporary objects, usually appearing on the right-hand side of an expression and which value is about to expire.

For example:

```
std::string hello { "Hello"}; // lvalue, a regular object
std::string world { "World"}; // lvalue
std::string msg = hello + world;
```

Above, the expression `hello + world` creates a temporary object. It doesn't have a name, and we cannot access it easily. Such temporary objects will end their lifetime immediately after the expression completes (unless it's assigned to a `const` or rvalue reference[1]), so we can steal resources from them safely. It doesn't make sense in the case of built-in types like integers or floats, as we need to copy values anyway. But in the case of strings or memory buffers, we can avoid data copy and just reassign the pointers.

Move constructors are a way to support the case with initialization from temporary objects. In many cases, they are an optimization over regular copy constructor calls. Additionally, they can also be used to pass "ownership" of the resource, for example, with smart pointers.

You can mark a regular object as expiring with the `std::move` function when you have a regular object with a name. This tells the compiler that the object's value is no longer needed, so it's safe to "steal" resources from it.

Have a look at this example:

Ex 3.3. Move Constructor. Run @Compiler Explorer

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

class Product {
public:
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name}
    {
        std::cout << "Product(): " << id_ << ", " << name_ << '\n';
    }

    Product(Product&& other)
        : id_{other.id_}, name_{std::move(other.name_)}
    {
        std::cout << "Product(move): " << id_ << ", " << name_ << '\n';
    }
```

[1]The lifetime of a temporary object may be extended by binding to a const lvalue reference or to an rvalue reference. See more at https://en.cppreference.com/w/cpp/language/lifetime.

```
    const std::string& name() const { return name_; }

private:
    int id_;
    std::string name_;
};

int main() {
    Product tvSet {100, "tv set"};
    std::cout << tvSet.name() << " created...\n";
    Product setV2 { std::move(tvSet) };
    std::cout << setV2.name() << " created...\n";
    std::cout << "old value: " << tvSet.name() << '\n';
}
```

When you run the code, you can see the following output:

```
Product(): 100, tv set
tv set created...
Product(move): 100, tv set
tv set created...
old value:
```

As you can see, we create the first object, and then mark it as expiring. This gives a chance for the compiler to call the move constructor.

```
Product(Product&& other)
        : id_(other.id_), name_(std::move(other.name_))
```

The above implementation is similar, but we need to pay attention to details. Since `id_` is just an integer, all we can do is copy the value. We cannot perform any optimizations here. As for the `name_` member, we can initialize it with `std::move(other.name_)`. We encounter the first problem, `other.name_` is a name, so not a temporary (a temporary has no name); we can not move (take, steal) its contents. That is why we tell the compiler to interpret it as temporary by using the expression `std::move(other.name_)`. This will invoke the move constructor for `std::string`, and, potentially, "steal" the buffer from `other.name_`.

The move constructor must ensure that the other object is left in an unspecified but valid state. In our case, we can see it in the last line of the output. The line `old value:` ends with nothing, so the string was simply cleared.

Move constructors can be marked with `explicit`, but it's not a common practice and might affect generic code that relies on implicit move constructors (like standard algorithms).

`noexcept` and move constructors

Content available in the full version of the book.

A compiler-generated move constructor

Content available in the full version of the book.

Distinguishing from assignment

Content available in the full version of the book.

Adding logging to constructors

As an exercise, let's add logging to our `DataPacket` class and see when each constructor is called:

Ex 3.6. Logging in the `DataPacket` class. Run @Compiler Explorer

```
class DataPacket {
    std::string data_;
    size_t checkSum_;
    size_t serverId_;

public:
    DataPacket()
    : data_{}
    , checkSum_{0}
    , serverId_{0}
    { }

    explicit DataPacket(const std::string& data, size_t serverId)
    : data_{data}
    , checkSum_{calcCheckSum(data)}
    , serverId_{serverId}
    {
        std::cout << "Ctor for \"" << data_ << "\"\n";
    }

    DataPacket(const DataPacket& other)
    : data_{other.data_}
    , checkSum_{other.checkSum_}
    , serverId_{other.serverId_}
    {
        std::cout << "Copy ctor for \"" << data_ << "\"\n";
    }

    DataPacket(DataPacket&& other)
    : data_{std::move(other.data_)}   // move string member...
    , checkSum_{other.checkSum_}       // no need to move built-in types...
    , serverId_{other.serverId_}
    {
        other.checkSum_ = 0; // leave this in a proper state
        std::cout << "Move ctor for \"" << data_ << "\"\n";
    }

```

```
38      DataPacket& operator=(const DataPacket& other) {
39          if (this != &other) {
40              data_ = other.data_;
41              checkSum_ = other.checkSum_;
42              serverId_ = other.serverId_;
43              std::cout << "Assignment for \"" << data_ << "\"\n";
44          }
45          return *this;
46      }
47
48      DataPacket& operator=(DataPacket&& other) {
49          if (this != &other) {
50              data_ = std::move(other.data_);
51              checkSum_ = other.checkSum_;
52              other.checkSum_ = 0; // leave this in a proper state
53              serverId_ = other.serverId_;
54              std::cout << "Move Assignment for \"" << data_ << "\"\n";
55          }
56          return *this;
57      }
58
59      // getters/setters
60  };
```

And here's the `main()` function:

Ex 3.6. Logging in the `DataPacket` class, the main function. Run @Compiler Explorer

```
1   int main() {
2       DataPacket firstMsg {"first msg", 101 };
3       DataPacket copyMsg { firstMsg };
4
5       DataPacket secondMsg { "second msg", 202 };
6       copyMsg = secondMsg;
7
8       DataPacket movedMsg { std::move(secondMsg)};
9       // now we stole the data, so it should be empty...
10      std::cout << "secondMsg's data after move ctor): \""
11                << secondMsg.getData() << "\", sum: "
```

```
12                << secondMsg.getCheckSum() << '\n';
13
14      movedMsg = std::move(firstMsg);
15
16      // now we stole the name, so it should be empty...
17      std::cout << "firstMsg's data after move ctor): \""
18                << firstMsg.getData() << "\", sum: "
19                << firstMsg.getCheckSum() << '\n';
20  }
```

When you run the example, you should see the following output:

```
Ctor for "first msg"
Copy ctor for "first msg"
Ctor for "second msg"
Assignment for "second msg"
Move ctor for "second msg"
secondMsg's data after move ctor): "", sum: 0
Move Assignment for "first msg"
firstMsg's data after move ctor): "", sum: 0
```

The example creates several `DataPacket` objects, and with each creation, you can see that the compiler invokes the appropriate constructor or an assignment operator. For instance, in **line 3**, we need a copy constructor call. On the other hand, **line 5** shows an assignment (`copyMsg` already exists). In the last section of `main()`, **lines 8 and 14**, there are calls to `std::move()`, which marks `secondMsg` and `firstMsg` as an rvalue reference, from which the contents could be moved. This means that the object is unimportant later, and we can "steal" from it. In this case, the compiler will call a move constructor or move assignment operator.

Trivial classes and user-declared/user-provided default constructors

Content available in the full version of the book.

4. Delegating and Inheriting Constructors

Content available in the full version of the book.

Limitations

Content available in the full version of the book.

Inheritance

Content available in the full version of the book.

Inheriting constructors

In our previous example with `DebugPropertyInfo` we didn't have any new data members, only some new member functions. The code showed a single constructor called the base class constructor. Since C++11, you can tell the compiler to "reuse" the code:

Ex 4.4. Inheriting constructors. Run @Compiler Explorer

```
class DebugDataPacket : public DataPacket {
public:
    using DataPacket::DataPacket;

    void DebugPrint(std::ostream& os) {
        os << getData() << ", " << getCheckSum() << '\n';
    }
};

int main() {
```

```
11      DebugDataPacket hello{"hello!", 404};
12      hello.DebugPrint(std::cout);
13  }
```

Consider **line 3** - `using DataPacket::DataPacket;`. This tells the compiler that it can use **all** constructors from the base class, ignoring access modifiers. It means that all public constructors are visible and can be called, but the protected will still be `protected` in that context. Still, if you want to limit the access to constructors, you must explicitly write constructors for `DebugDataPacket`.

We completed all information about constructors, but it's good to mention one more thing: destructors. See in the next chapter.

5. Destructors

While constructors are responsible for various situations where an object is created, C++ also offers a way to handle object destruction. C++ doesn't provide any form of garbage collection available in many popular programming languages, but thanks to precise lifetime specification, you can be confident when your object will be destroyed.

Each class has a special member function called a destructor. If you don't write one, the compiler prepares a default implementation. A destructor is called when an object ends its lifetime. In most cases, it means that an object goes out of the scope (for stack-allocated variables), or when a delete operator is called (for heap-allocated variables). Additionally, when you have a user-defined class, it will automatically call destructors for its data members. For more information about lifetime, see a good summary at C++Reference page[1].

Basics

Before we move on, it would be good to expand our terminology. So far I mentioned "object" to refer to entities of some type and relied on our "intuition" on how to access such entities. But the C++ Standard defines an *object* in the following terms (simplified, based on C++ Draft - intro.object[2]):

> The constructs in a C++ program create, destroy, refer to, access, and manipulate objects. An object is created by a definition, by a new-expression, by an operation that implicitly creates objects, or when a temporary object is created. An object occupies a region of storage in its period of construction, throughout its lifetime, and in its period of destruction.

And continuing:

[1] https://en.cppreference.com/w/cpp/language/lifetime
[2] https://timsong-cpp.github.io/cppwp/n4868/intro.object#1

- An object can have a name,
- An object has a storage duration which influences its lifetime,
- An object has a type,
- Objects can contain other objects, called subobjects. A subobject can be a member subobject, a base class subobject, or an array element.

Here's a basic scenario for a destructor that handles a case where the lifetime of an object ends:

Ex 5.1. A logging destructor. Run @Compiler Explorer

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

class Product {
public:
    explicit Product(const char* name, unsigned id)
    : name_(name)
    , id_(id)
    {
        std::cout << name << ", id " << id << '\n';
    }

    ~Product() {
        std::cout << name_ << " destructor...\n";
    }

    std::string Name() const { return name_; }
    unsigned Id() const { return id_; }

private:
    std::string name_;
    unsigned id_;
};
```

The example contains the following special member function:

```
~Product() {
    std::cout << name_ << " destructor...\n";
}
```

The syntax is unique as it has no parameters and has the ~ prefix. You can also have only one destructor in a class. What's more, a destructor doesn't return any value.

Now, let's create two objects of that type:

Ex 5.1. A logging destructor, continuation. Run @Compiler Explorer

```
int main() {
   {
       Product tvset("TV Set", 123);
   }
   {
       Product car("Mustang", 999);
   }
}
```

In our case, the constructor and the destructor is used to perform the logging. When you run the example, you'll see the following output:

```
TV Set, id 123
TV Set destructor...
Mustang, id 999
Mustang destructor...
```

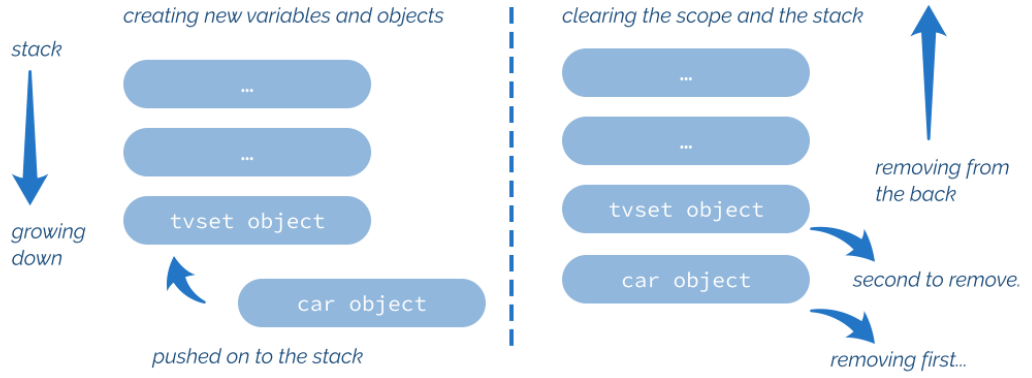
I specifically enclosed objects (created on the stack) in separate scopes so that their lifetime ends when their scope ends. On the other hand, if we have code:

```
int main() {
    Product tvset("TV Set", 123);
    Product car("Mustang", 999);
}
```

Then both `tvset` and `car` share the same lifetime scope so that we can expect the following output:

```
TV Set, id 123
Mustang, id 999
Mustang destructor...
TV Set destructor..
```

As you can see, the destructors are called in the reverse order of how they were created. It's because the stack is a LIFO structure (Last In First Out). `tvset` was created first and added to the stack, then `car` is added. When the function goes out of the scope, the stack is cleared, taking elements in the reverse order. So `car` is deleted first, and then `tvset`. This is illustrated by the following diagram:



Adding and removing objects from the stack.

Objects allocated on the heap

Content available in the full version of the book.

Destructors and data members

Content available in the full version of the book.

Virtual destructors and polymorphism

Content available in the full version of the book.

Partially created objects

Content available in the full version of the book.

Use Cases

The primary use case for destructors is when you need to release resources allocated in a constructor. For example, you allocate some memory when the object is created, and then the memory must be released to avoid memory leaks. Similarly, you can open a file or a database connection, and then you must ensure the file or the connection is closed when the object goes out of scope. Fortunately, in Modern C++, there are fewer and fewer places where you need custom destructors. For example, when your data members are standard containers (like `std::vector<int>`, or `std::map<std::string, int>`) in your classes, then you can rely on default destructors to do the job. Standard containers like `std::vector<int>` might allocate memory buffers, but they also manage that buffer and release it properly, so you don't need to take any action when using them in a class.

A compiler-generated destructor

Content available in the full version of the book.

6. Initialization and Type Deduction

Content available in the full version of the book.

7. Quiz on Constructors

Congratulations!

You've just completed the section on the basics and constructors.

Here's a quick quiz. Try answering the following questions, and then we will continue our journey :)

1. Can a constructor have a different name than the class name?

1. Yes
2. No
3. Yes, but it can be only named `self()`

2. What operations are called in the following code? Pick one option.

```
std::string s { "Hello World" };
std::string other = s;
```

1. A constructor is called for `s`. Then, as assignment operation is called for `other`.
2. A constructor is called for `s`, and then a copy constructor is called to create `other`.
3. A constructor is called for `s`, and then another regular constructor is called for `other`.

More questions available in the full version of the book.

8. Non-Static Data Member Initialization

You've learned a lot of techniques related to constructors! You can initialize data members in various constructors, delegate them to reuse code, and inherit them from base classes. Yet, we can still improve on assigning default values for data members. I mentioned this feature in the first chapter, where we gave default values for aggregates. We can do the same for classes. And in this chapter, we'll look at the full syntax and options related to this feature.

Please have a look at the example below:

Ex 8.1. NSDMI Basics. Run @CompilerExplorer

```
class DataPacket {
    std::string data_;
    size_t checkSum_ { 0 };
    size_t serverId_ { 0 };

public:
    DataPacket() = default;

    DataPacket(const std::string& data, size_t serverId)
    : data_{data}
    , checkSum_{calcCheckSum(data)}
    , serverId_{serverId}
    { }

    // getters and setters...
};
```

As you can see, the variables are assigned their default values individually in their place of declaration. There's no need to set values inside a constructor. It's much better than using a default constructor because it combines declaration and initialization code. This way, it's harder to leave data members uninitialized!

Let's explore this handy feature of Modern C++ in detail.

How it works

This section shows how the compiler "expands" the code to initialize data members.

For a simple declaration:

```
struct SimpleType {
    int field { 0 };
};
```

The code has to behave similarly as you'd define a constructor [1]:

```
struct SimpleType {
    SimpleType() : field(0) { }

    int field;
};
```

Experiment with the basic code below:

Ex 8.2. Basic Non-static data member initialization. Run @Compiler Explorer

```
#include <iostream>

struct SimpleType {
    int field { 0 };
};

int main() {
    SimpleType st;
    std::cout << "st.field is " << st.field << '\n';
}
```

As a small exercise, you can experiment with the above sample and assign different values to the `field` data member.

[1] Technically, those types will be different as the version without the constructor will be considered an aggregate type, but for the purpose of the discussion, it's not essential now.

Investigation

With some "machinery," we can see when the compiler performs the initialization.

Let's consider the following type:

```
struct SimpleType {
    int a { initA() };
    std::string b { initB() };

    // ...
};
```

The implementation of `initA()` and `initB()` functions have side effects, and they log extra messages:

```
int initA() {
    std::cout << "initA() called\n";
    return 1;
}

std::string initB() {
    std::cout << "initB() called\n";
    return "Hello";
}
```

This allows us to see when the code is called.

Experiments

Now, we can experiment and write some additional constructors:

```
struct SimpleType {
    int a { initA() };
    std::string b { initB() };

    SimpleType() { }
    SimpleType(int x) : a(x) { }
};
```

Next, we can run our test and see the results.

Ex 8.3. Calling init functions. Live code @Compiler Explorer

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

int initA() {
    std::cout << "initA() called\n";
    return 1;
}

std::string initB() {
    std::cout << "initB() called\n";
    return "Hello";
}

struct SimpleType {
    int a { initA() };
    std::string b { initB() };

    SimpleType() { }
    SimpleType(int x) : a(x) { }
};

int main() {
    std::cout << "SimpleType t0\n";
    SimpleType t0;
    std::cout << "SimpleType t1(10)\n";
    SimpleType t1(10);
}
```

After running the code, we can see the following output:

```
SimpleType t1
initA() called
initB() called
SimpleType t1(10)
initB() called
```

You can observe the following:

`t0` is default-initialized; therefore, both fields are initialized with their default values. In other words, the compiler calls `{initA()}` and `{initB{}}`. Please notice that they are initialized in the order they appear in the class/struct declaration.

In the second case, for `t1`, only one value is default initialized, and the other comes from the constructor parameter.

As you might already guess, the compiler initializes the fields as if the fields were initialized in a "member initialization list". Therefore, they get the default values before the constructor's body is invoked.

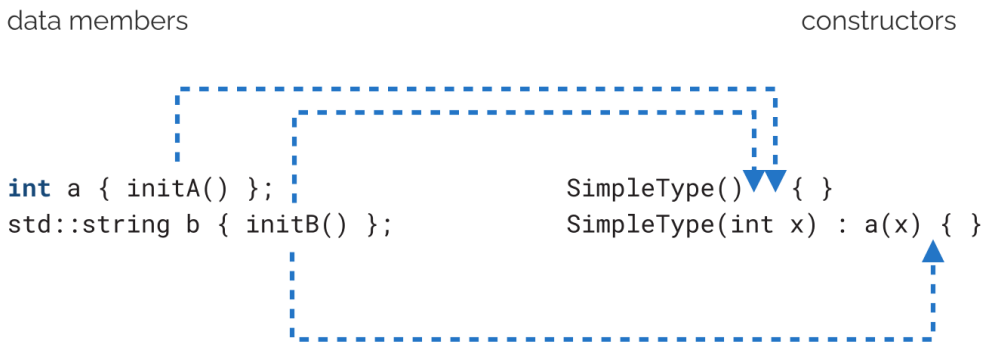
In other words, the compiler "conceptually" expands the code:

```
struct SimpleType {
    int a { initA() };
    std::string b { initB() };

    SimpleType() { }
    SimpleType(int x) : a(x) { }
};
```

Into:

```
struct SimpleType {
    int a;
    std::string b;

    SimpleType() : a(initA()), b(initB()) { }
    SimpleType(int x) : a(x), b(initB())  { }
};
```

We can also visualize it using the following diagram:



Other forms of NSDMI

Content available in the full version of the book.

Copy constructor and NSDMI

Content available in the full version of the book.

Move constructor and NSDMI

Content available in the full version of the book.

C++14 changes

Originally, in C++11, if you used default member initialization, your class couldn't be an aggregate type:

```
struct Point { float x = 1.0f; float y = 2.0f; };

// won't compile in C++11
Point myPt { 10.0f, 11.0f };
```

The above code won't work when compiling with the C++11 flag because you cannot aggregate-initialize our `Point` structure. It's not an aggregate.

Fortunately, C++14 provides a solution to this problem, and that's this line:

```
Point myPt { 10.0f, 11.0f};
```

The code works as expected now. You can see and play with the full code below:

Ex 10.1. Aggregates and NSDMI in C++14. Run @CompilerExplorer

```
#include <iostream>

struct Point { float x = 1.0f; float y = 2.0f; };

int main()
{
    Point myPt { 10.0f };
    std::cout << myPt.x << ", " << myPt.y << '\n';
}
```

C++20 changes

Content available in the full version of the book.

Limitations of NSDMI

Content available in the full version of the book.

NSDMI: Advantages and Disadvantages

Let's summarize non-static data member initialization.

Advantages of NSDMI

Content available in the full version of the book.

Any negative sides of NSDMI?

Content available in the full version of the book.

NSDMI summary

Before C++11, the best way to initialize data members was through a member initialization list inside a constructor. Thanks to C++11, we can now initialize data members in the place where we declare them, and the initialization happens just before the constructor body kicks in.

In the chapter, we covered the syntax, how it works with various types of constructors and the limitations. You also saw changes made in C++14 (aggregate classes) and missing bitfield initialization fixed in C++20.

The C++ Core Guidelines advise using NSDMI in at least two sections.

C++ Core Guidelines - C.48[2]:

> **C.48** Prefer in-class initializers to member initializers in constructors for constant initializers:
>
> **Reason**: Makes it explicit that the same value is expected to be used in all constructors. Avoids repetition. Avoids maintenance problems. It leads to the shortest and most efficient code.

And in C++ Core Guidelines - C.45[3]

[2] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c48-prefer-in-class-initializers-to-member-initializers-in-constructors-for-constant-initializers

[3] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c45-dont-define-a-default-constructor-that-only-initializes-data-members-use-in-class-member-initializers-instead

> **C.45**Don't define a default constructor that only initializes data members; use in-class member initializers instead
>
> **Reason**: Using in-class member initializers lets the compiler generate the function for you. The compiler-generated function can be more efficient.

If you like to read more about NSDMI, I highly recommend reading the book "Embracing Modern C++ Safely", chapter 2, page 318. There's a whole section on advanced cases for this powerful C++ feature.

NSDMI: Exercises

Check your skills with two coding exercises.

The first exercise

Content available in the full version of the book.

The second exercise

Content available in the full version of the book.

9. Containers as Data Members

`CarInfo`, `DataPacket`, and `Product` types used relatively simple data members like integers, doubles, or strings. While `std::string` is, in fact, a container (of characters), we tend to use it as an elementary type. In this section, I'd like to discuss more complex data members like arrays, vectors, or maps.

The basics

Content available in the full version of the book.

Using `std::initializer` list

(*) this section will be added in the future.

Example implementation

Content available in the full version of the book.

10. Non-regular Data Members

Thus far, we spoke about mutable non-static data members like integers, doubles, or strings. Such objects are regular, meaning they are copyable, default constructible, and equally comparable. In C++20 there's even a concept for that purpose: `std::regular`, see @C++Reference[1].

However, you can also have other categories of objects in a class. For example, a custom type might contain constant data members, pointers, references, or moveable only fields like unique pointers or mutexes. For such members, we have immediate issues with default copy constructors (the compiler won't create them).

In this chapter, we'll shed some light on such cases.

Constant non-static data members

Content available in the full version of the book.

References as data members

(*) this section will be added in the future.

Pointers as data members

(*) this section will be added in the future.

Moveable-only data members

(*) this section will be added in the future.

[1] https://en.cppreference.com/w/cpp/concepts/regular

Summary

Having discussed other categories of non-static data members, we can now examine static data members. How to use them in Modern C++? See the next chapter.

11. Inline Variables in C++17

In this chapter, you'll see how to enhance and simplify code using inline variables from C++17.

About static data members

In general, each and every instance (object) of a class has non-static data members as its own data fields; each instance is separate from the other. If we consider a type (a class) representing a Fruit and it has a data member named "mass", then each particular instance of that Fruit class has a "mass" member belonging to it. If we have 10 Fruit objects, the "mass" data member is replicated ten times. On the other hand, each type can also have static data members that are not bound to any instance of the class. In the case of our Fruit class, we can specify a so-called static variable named "default mass", accessible to each Fruit instance, but it wouldn't be part of any instance. In other words, it's like a global variable in the namespace of the Fruit type.

Consider the following example:

Ex 12.1. Simple static Data Member. Run @Compiler Explorer

```
#include <iostream>

struct Value {
    int x;

    static int y;
};

int Value::y = 0; // definition

int main() {
    Value v { 10 };
    std::cout << "sizeof(int): " << sizeof(int) << '\n';
    std::cout << "sizeof(Value): " << sizeof(Value) << '\n';
```

```
    std::cout << "v.x: " << v.x << '\n';
    Value::y = 10;
    std::cout << "Value::y: " << Value::y << '\n';
}
```

When you run this program, you'll see the following output:

```
sizeof(int): 4
sizeof(Value): 4
v.x: 10
Value::y: 10
```

`static int y` declared in the scope of the `Value` class created a variable that is not part of any `Value` type instance. You can see that it doesn't contribute to the size of the whole class. It's the same as the size of the `int` type.

In the further sections, let's consider a more practical use case for such class members.

Motivation for inline variables

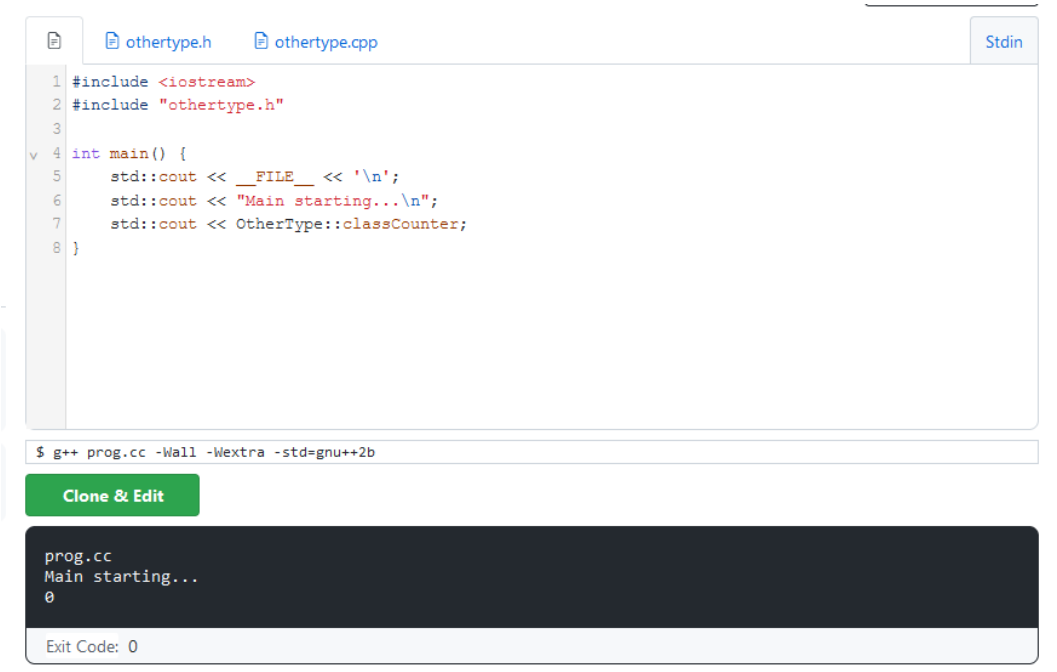
In C++11/14, if you wanted to add a static data member to a class, you needed to declare it and define it later in one compilation unit. In the example from the previous section, we defined it in the same compilation unit as the `main()` function. Commonly, such variables are defined in the corresponding implementation file.

For example:

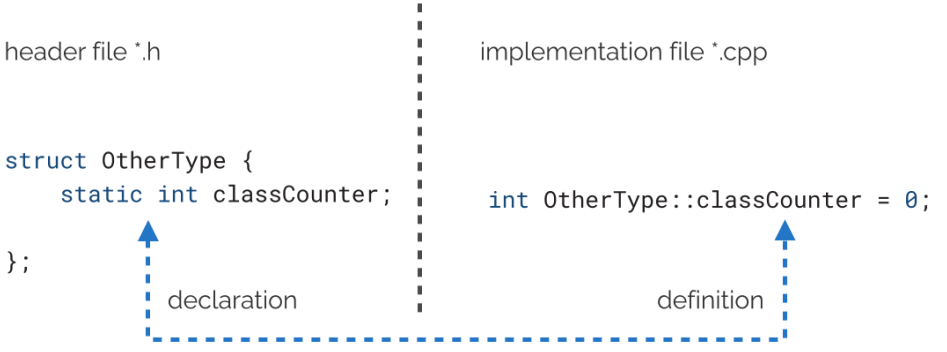
Ex 12.2. Static data member, multiple files. Run @Wandbox

```
// a header file:
struct OtherType {
    static int classCounter;

    // ...
};

// implementation, cpp file
int OtherType::classCounter = 0;
```

This time we also used Wandbox online compiler - as it's easy to create and compile multiple files:



As you can see above, `classCounter` is an `int`, and you have to write it twice: in a header file and then in the CPP file.



The only exception to this rule (even before C++11) is a static constant integral variable that

you can declare and initialize in one place:

```
class MyType {
    static const int ImportantValue = 42;
};
```

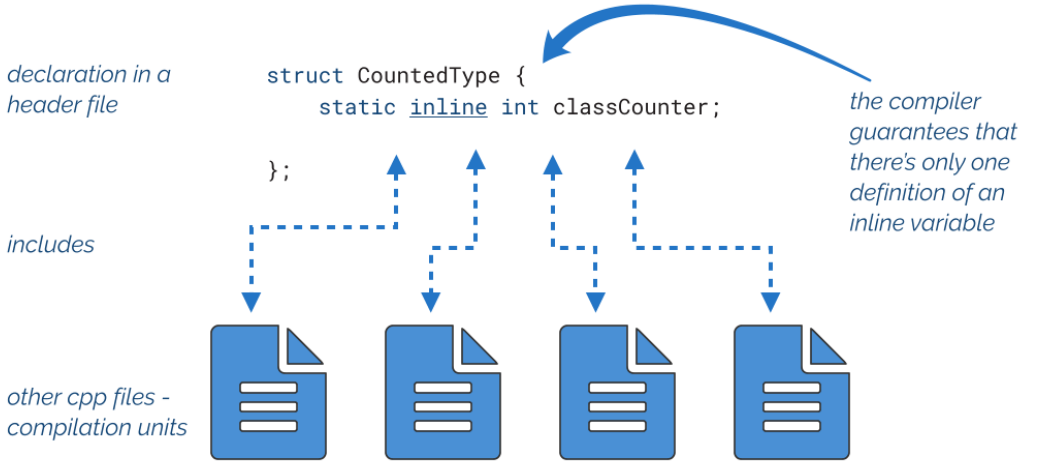
You do not have to define `ImportantValue` in a CPP file.

Fortunately, C++17 gave us **inline variables**, which means we can define a `static inline` variable inside a class without defining them in a CPP file.

Ex 12.3. Static inline member. Run @Wandbox

```
// a header file, C++17:
struct OtherType {
    static inline int classCounter = 0;

    // ...
};
```

The compiler (and the linker) guarantees that there's precisely one definition of this static variable for all translation units that include the class declaration. Inline variables remain static class variables, so they will be initialized before the `main()` function is called.



This feature makes it much easier to develop header-only libraries because there's no need to create CPP files for static variables or use hacks to keep them in a header file (for example, by creating static member functions with static variables inside).

See the example below:

```
// CountedType.h
struct CountedType {
    static inline int classCounter = 0;

    // simple counting... only ctor and dtor implemented...
    CountedType() { ++classCounter; }
    ~CountedType() { --classCounter; }
};
```

And the `main()` function:

Ex 12.4. Static inline member. Run @Wandbox

```
#include <iostream>
#include "CountedType.h"

int main() {
    {
        CountedType c0;
        CountedType c1;
        std::cout << CountedType::classCounter << '\n';
    }
    std::cout << CountedType::classCounter << '\n';
}
```

The code above declares `classCounter` inside `CountedType`, which is a static data member. The class is defined in a separate header file. Thanks to C++17, we can declare the variable as `inline`. Then, there's no need to write a corresponding definition later. Without `inline`, the code wouldn't compile.

Later, in the `main()` function, the example creates two objects of `CountedType`. The static variable is incremented when there's a call to the constructor. When an object is destroyed, the variable is decremented. We can output this value and see the current count of objects.

Exercise for inline variables

Content available in the full version of the book.

Global inline variables

Content available in the full version of the book.

Constexpr and inline variables

Content available in the full version of the book.

12. Aggregates and Designated Initializers in C++20

Across the book, you've seen a lot of cases for intuitively simple structures with all public data members. Such types, along with arrays, are called *Aggregates*. In this chapter, we'll look at some C++20 changes and new ways to initialize such objects.

Aggregates in C++20

To sum up, as of C++20, here's the definition of an *aggregate type* from the C++ Standard: dcl.init.aggr[1].

> An aggregate is an array or a class type with:
>
> - no user-provided, explicit, or inherited constructors
> - no private or protected non-static data members
> - no virtual functions, and
> - no virtual, private, or protected base classes

Here are some examples of aggregates:

[1] https://timsong-cpp.github.io/cppwp/n4868/dcl.init.aggr#:initialization,aggregate

Ex 13.1. Aggregate classes, several examples. Run @Compiler Explorer

```
struct Base { int x {42}; };
struct Derived : Base { int y; };

struct Param {
    std::string name;
    int val;
    void Parse();  // member functions allowed
};

int main() {
    Derived d {100, 1000};
    std::cout << "d.x " << d.x << ", d.y " << d.y << '\n';
    Derived d2 { 1 };
    std::cout << "d2.x " << d2.x << ", d2.y " << d2.y << '\n';
    Param p {"value", 10};
    std::cout << "p.name " << p.name << ", p.val " << p.val << '\n';

    double arr[] { 1.1, 2.2, 3.3, 4.4};
    std::cout << "arr[0] " << arr[0] << '\n';
    Param params[] {{"val", 10}, {"name", 42}};
    std::cout << "params[0].name " << params[0].name << '\n';
}
```

In C++20, in some limited cases, you can also use parens `X(args...)` to initialize an aggregate:

```
// C++20 and parens:
Point pt (1, 2);
// Point pt = (1, 2); // doesn't work

double params[] (9.81, 3.14, 1.44);
// double paramsDeduced[] = (9.81, 3.14, 1.44); // won't deduce
int arrX[10]  (1, 2, 3, 4); // rest is 0
```

Such improvement helps, especially in a generic template code where you want to work with various types of objects. For example, the following code wasn't possible until C++20:

Ex 13.2. Aggregates and parens for `make_unique`. Run @Compiler Explorer

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

int main() {
    auto ptr = std::make_unique<Point>(10, 20);
}
```

`make_unique` takes a variable number of arguments and passes them to a constructor. This function uses parens to call the constructor. Since aggregates has no user-declared constructors, then such syntax generated errors. With the C++20 change, the code works fine now. Suppose you want to dig more into this topic. In that case, I highly recommend reading C++20's parenthesized aggregate initialization has some downsides – Arthur O'Dwyer[2], which discusses pros and cons of this new initialization syntax.

The basics of Designated Initializers

The C++20 Standard also gives us another handy way to initialize data members. The new feature is called designated initializers, which might be familiar to C programmers.

As of C++20, to initialize an aggregate object, you can write the following:

```
Type obj = { .designator = val, .designator { val2 }, ... };
```

For example:

```
struct Point { double x; double y; };
Point p { .x = 10.0, .y = 20.0 };
```

Designator points to a name of a non-static data member from our class, like `.x` or `.y`.

One of the main reasons to use this new kind of initialization is to increase readability. Compare the following initialization forms:

[2]https://quuxplusone.github.io/blog/2022/06/03/aggregate-parens-init-considered-kinda-bad/

```
struct Date {
    int year;
    int month;
    int day;
};

// new
Date inFutureCpp20 { .year = 2050, .month = 4, .day = 10 };
// old
Date inFutureOld   { 2050, 4, 10 };
```

In the case of the `Date` class, it might be unclear what the order of days/month or month/days is. With designated initializers (`inFutureCpp20`), it's very easy to see the order of data members.

Rules

Content available in the full version of the book.

Advantages of designated initialization

- Readability: A designator points to the specific data member, so it's impossible to make mistakes here.
- Flexibility: You can skip some data members and rely on default values for others.
- Compatibility with C: In C99, it's popular to use a similar form of initialization (although even more relaxed). With the C++20 feature, it's possible to have very similar code and share it.
- Standardization: Some compilers, like GCC or Clang, already had some extensions for this feature, so it's a natural step to enable it in all compilers.

Examples

Content available in the full version of the book.

Summary

As you can see, designated initializers are handy and usually more readable way of initializing aggregate types. The new technique is also common in other programming languages, like C or Python, so having it in C++ makes the programming experience even better.

The summary example

Content available in the full version of the book.

Compiler support

Here's a table with support for the features we discussed:

Content available in the full version of the book.

13. Techniques and Use Cases

Across the book, we've touched on many different topics, sometimes only in a theoretical way. In this chapter, however, I grouped many of those features and demonstrated their benefits in several practical use cases.

You'll learn about the following aspects:

- Strong types and the `explicit` keyword,
- Initializing string data members,
- Copy and Swap Idiom as a potential simplification of copy and move operations,
- CRTP,
- Creating a simple resource manager (RAII) class.
- Factory With Self-Registering Types
- And more!

Let's start.

Using `explicit` for strong types

If you recall the first chapter, I used `double` to indicate horsepower (hp) inside the `CarInfo` structure. However, we might quickly encounter a problem where we forget about the unit and treat it as Watts instead. Can we somehow limit such problematic cases?

The answer is positive, and the main idea is to wrap the data member `double power` in a separate class type with `explicit` constructors. That it will be harder to misuse it, such an approach is called *Strong Typing*.

Have a look at two similar wrapper types:

Ex 13.1. Strong types and area units classes. Run @Compiler Explorer

```
constexpr double ToWattsRatio { 745.699872 };

class HorsePower;

class WattPower {
public:
    WattPower() = default;
    explicit WattPower(double p) : power_{p} { }
    explicit WattPower(const HorsePower& h);

    double getValue() const { return power_; }
private:
    double power_ {0.};
};

class HorsePower {
public:
    HorsePower() = default;
    explicit HorsePower(double p) : power_{p} { }
    explicit HorsePower(const WattPower& w);

    double getValue() const { return power_; }
private:
    double power_ {0.};
};
```

As you can see, we have two types that use `explicit` constructors to initialize their private data members. To create an object, you have to write the correct type name explicitly, and thus it should limit the chance of mistakes.

And here is the implementation of the converting constructors as well as stream operators for easy output:

Ex 13.2. Strong Types and area units, implementation. Run @Compiler Explorer

```
constexpr double ToWattsRatio { 745.699872 };

class HorsePower;

class WattPower {
    /* as before */
};

class HorsePower {
    /* as before */
};

WattPower::WattPower(const HorsePower& h)
: power_{h.getValue()*ToWattsRatio}
{ }

HorsePower::HorsePower(const WattPower& w)
: power_{w.getValue()/ToWattsRatio}
{ }

std::ostream& operator<<(std::ostream& os, const WattPower& w) {
    os << w.getValue() << "W";
    return os;
}

std::ostream& operator<<(std::ostream& os, const HorsePower& h) {
    os << h.getValue() << "hp";
    return os;
}
```

The interface allows us to convert between various units safely.

```
//HorsePower hp = 10.; // not possible, copy initialization
HorsePower hp{ 10. }; // fine
WattPower w { 1. }; // fine
WattPower watts { hp }; // fine, performs the proper conversion for us!
```

Additionally, we have the output support that writes out the proper unit name.

We can use the solution now:

```
void printInfo(const CarInfo& c) {
    std::cout << c.name << ", "
              << c.year << " year, "
              << c.seats << " seats, "
              << c.power << '\n';
}

int main() {
    CarInfo firstCar{"Megane", 2003, 5, HorsePower{116}};
    printInfo(firstCar);
    CarInfo superCar{"Ferrari", 2022, 2, HorsePower{300}};
    printInfo(superCar);
    superCar.power = HorsePower{WattPower{500000}};
    printInfo(superCar);
}
```

And we'll get the following output:

```
Megane, 2003 year, 5 seats, 116hp
Ferrari, 2022 year, 2 seats, 300hp
Ferrari, 2022 year, 2 seats, 670.511hp
```

While I had to be more explicit and write the types, the code can be safer as it's harder to type something accidentally.

In C++11, you can also leverage user-defined literals to allow easier creation of objects. Especially useful for units, string, numerical types, time, and dates. For example, We could create a named literal `_m2` and then write `50.0_m2` to create an instance rather than `SqMeters{50.2}`. See more at C++Reference - User-defined literals[1].

[1] https://en.cppreference.com/w/cpp/language/user_literal

For more information about Strong Types, I highly recommend reading many articles on the Fluent C++ blog. For example, start with this one: Strong types for strong interfaces - Fluent C++[2].

Best way to initialize `string` data members

Content available in the full version of the book.

The copy and swap idiom

Content available in the full version of the book. ## CRTP class counter {#sectioncrtp}

Content available in the full version of the book.

Several initialization types in one class

As the demo of various initialization techniques, I'd like to show code that creates N random "application windows."

Here are the core points of the demo:

- A `Window` class contains basic parameters like name (on the title bar), width, height, and some flags (bits per pixel, visibility).
- The demo selects a random number X and will try to generate X `Window` objects.
- Each object will have a random name composed of predefined words and a random size.
- The application prints each window using `std::cout`.
- As an additional check, an `InstanceCounter` class counts the number of `Window` objects. We can use this helper to verify the correctness of the demo.

Here's the first part that defines the `Flags` object:

[2]https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/

Ex 13.5. The Flags type. Run @Compiler Explorer

```
struct Flags {
    unsigned bppMode_ : 4 { 0 }; // bits per pixel
    unsigned visible_ : 1 { 1 };
    unsigned extData  : 2 { 0 };
};
```

Here's the main class:

Ex 13.5. The Window type. Run @Compiler Explorer

```
class Window : public InstanceCounter<Window> {
    static constexpr unsigned default_width { 1028 };
    static constexpr unsigned default_height { 768 };
    static constexpr unsigned default_bpp { 8 };

    unsigned width_ { default_width };
    unsigned height_ { default_height };
    Flags flags_ {.bppMode_ { default_bpp } };
    std::string title_ { "Default Window" };

public:
    Window() = default;
    explicit Window(std::string title) : title_(std::move(title)) { }
    Window(std::string title, unsigned w, unsigned h) :
    width_(w), height_(h), title_(std::move(title)) {}

    friend std::ostream& operator<<(std::ostream& os, const Window& w) {
        os << w.title_ << ": " << w.width_ << "x" << w.height_;
        return os;
    }
};
```

The `Window` class uses several features discussed in the book:

- NSDMI to initialize data members,
- designated initializers from C++20, combined with NSDMI for the `flags_` data member,

- Custom constructors that offer several options to initialize the data members,
- We inherit from `InstanceCounter`, so each constructor invocation for the `Window` will also invoke the appropriate constructor in `InstanceCounter`. Similarly, the `InstanceCounter` destructor will be nicely called from the implicit default destructor of the `Window` class.

And now the final demo code:

Ex 13.5. The Window type. Run @Compiler Explorer

```
void WindowDemo() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 20);

    const int windowCount = std::uniform_int_distribution<>(2, 10)(gen);
    std::cout << "Generating " << windowCount << " random Windows\n";

    const std::array adjs { "regular ", "empty ", "blue ", "super " };
    const std::array nouns { "app", "tool", "console", "game" };
    const std::array sizes { 1080u, 1920u, 768u, 320u, 640u, 3840u, 800u };

    std::vector<Window> windows;
    for (int i = 0; i < windowCount; ++i) {
        auto r = distrib(gen);
        auto r2 = distrib(gen);
        auto name = std::string { adjs[(r + i) % adjs.size()] } +
                    nouns[r2 % nouns.size()];
        Window w{name, sizes[r2 % sizes.size()],
                 sizes[r % sizes.size()]};
        windows.push_back(w);
    }

    for (const auto& w : windows)
        std::cout << w << '\n';

    std::cout << "Created " << Window::GetInstanceCounter() << " Windows\n";
}

int main() {
```

```
    WindowDemo();

    if (Window::GetInstanceCounter() != 0) {
        std::cout << Window::GetInstanceCounter()
                  << " Windows are still alive!\n";
    }
}
```

Here's the possible output:

```
Generating 8 random Windows
super tool: 320x320
regular tool: 320x640
super game: 1080x768
super game: 640x1080
regular tool: 1920x3840
empty tool: 1920x3840
blue game: 320x768
empty console: 320x320
Created 8 Windows
```

In `WindowDemo`, the code declares some basic data and generates a random number. Later, in the main loop, we generate random numbers to pick values from `adjs`, `nouns`, and `sizes` arrays. Once the data is ready, I can create a `Window` object and place it in the `std::vector`. To show the creation of the `Window` object, I used `push_back` on a vector, but we can optimize it and call `emplace_back`, which doesn't need a temporary object:

```
windows.emplace_back(name, sizes[r2 % sizes.size()], sizes[r % sizes.size()]);
```

Later there's another loop that prints all windows.

In the code, I didn't have to specify the full type for `std::array<Type, Count>` as the compiler could deduce everything for me! Thanks to Class Type Argument Deduction (CTAD) and Deduction guides from C++17, the compiler can help us save some typing. See more @C++Reference - deduction guides for array[3].

[3] https://en.cppreference.com/w/cpp/container/array/deduction_guides

The code uses `InstanceCounter` as a bonus debugging facility to ensure we have the correct number of active objects. When `WindowDemo()` finishes, all instances should be removed, and we can double-check it inside `main()`.

Vector like RAII object

(*) this section will be added in the future.

Factory with self-registering types

(*) this section will be added in the future.

Summary

(*) this section will be added in the future.

14. The Final Quiz

Check your knowledge from this mini-book!

1. Which C++ Standard did add in-class default member initializers?

1. C++98
2. C++11
3. C++14
4. C++17

2. Can you use `auto` type deduction for non-static data members?

1. Yes, since C++11
2. No
3. Yes, since C++20

3. Do you need to define a `static inline` data member in a separate `cpp` file?

1. No, the definition happens at the same place where a static inline member is declared.
2. Yes, the compiler needs the definition in a cpp file.
3. Yes, the compiler needs a definition in all translation units that use this variable.

4. Can a static inline variable be non-constant?

1. Yes, it's just a regular variable.
2. No, inline variables must be constant.

5. Consider the following code:

```
struct S {
    int a { 10 };
    int b { 42 };
};
```

What's the output of the following line?

```
S s { 1 };
std::cout << s.a << ", " << s.b;
```

1. `1, 0`
2. `10, 42`
3. `1, 42`

6. Consider the following code:

```
class C {
    C(int x) : a(x) { }

    int a { 10 };
    int b { 42 };
};

C c(0);
```

More questions available in the full version of the book.

Appendix A - Quiz and Exercises Answers

Content available in the full version of the book.

References

Related materials and links about data member initialization in C++:

Proposals for C++ features:

- N2756[1] - Non-static data member initializers for C++11,
- P0683[2] - Default Bit Field Initializer for C++20,
- P0386[3] - Inline Variables C++17,
- P0329[4] - Designated Initializers C++20,
- P0960[5] and P1975[6] - Aggregate initialization from a parenthesized list for C++20.

Valuable resources for C++:

- C++ Standard Draft[7] - N4868 (October 2020 pre-virtual-plenary working draft/C++20 plus editorial changes),
- C++ compiler support - C++Reference[8] - a list of features and their compiler support since C++11,
- C++ Core Guidelines[9] - a community-edited and open guideline for C++ style, lead by Bjarne Stroustrup and Herb Sutter.

Books:

- "Embracing Modern C++ Safely"[10] by J. Lakos, V. Romeo , R. Khlebnikov, A. Meredith, a wonderful and very detailed book about latest C++ features, from C++11 till C++14 in the 1st edition.
- "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14"[11] by Scott Meyers

[1] https://wg21.link/N2756
[2] https://wg21.link/P0683
[3] https://wg21.link/P0386
[4] https://wg21.link/P0329
[5] https://wg21.link/p0960
[6] https://wg21.link/p1975
[7] https://timsong-cpp.github.io/cppwp/n4868/
[8] https://en.cppreference.com/w/cpp/compiler_support
[9] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
[10] https://amzn.to/3PywHTg
[11] https://amzn.to/3t5tmS4

Presentations:

- Core C++ 2019: Initialisation in modern C++[12] by Timur Doumler,
- CppCon 2018: "The Nightmare of Initialization in C"[13] by Nicolai Josuttis,
- CppCon 2021: Back To Basics: The Special Member Functions[14] by Klaus Iglberger,
- ACCU 2022: What Classes We Design and How[15] - by Peter Sommerlad,
- CppCon 2018 "The Bits Between the Bits: How We Get to main()"[16] - by Matt Godbolt.

Articles and other links:

- Non-Static Data Members Initialization - C++ Stories[17] - initial source for the book,
- What happens to your static variables at the start of the program? - C++ Stories[18],
- Always Almost Auto Style[19] by Herb Sutter,
- C++ Core Guidelines - C51[20] and C52[21] - about delegating and inheriting constructors,
- Modern C++ Features - Inherited and Delegating Constructors[22] by Arne Mertz,
- Trivial, standard-layout, POD, and literal types[23] at Microsoft Docs,
- Modern C++ Features - Uniform Initialization and initializer_list[24] by Arne Mertz,
- The cost of `std::initializer_list`[25] by Andrzej Krzemieński,
- Objects, their lifetimes and pointers[26] by Dawid Pilarski,
- Tutorial: When to Write Which Special Member[27] by Jonathan Müller,
- The implication of const or reference member variables in C++[28] by Lesley Lai,
- Brace initialization of user-defined types[29] by Glennan Carnie[30].

[12] https://www.youtube.com/watch?v=v0jM4wm1zYA
[13] https://www.youtube.com/watch?v=7DTlWPgX6zs
[14] https://www.youtube.com/watch?v=9BM5LAvNtus
[15] https://www.youtube.com/watch?v=fzsBZicBe88
[16] https://www.youtube.com/watch?v=dOfucXtyEsU
[17] https://www.cppstories.com/2015/02/non-static-data-members-initialization/
[18] https://www.cppstories.com/2018/02/staticvars/
[19] https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/
[20] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c51-use-delegating-constructors-to-represent-common-actions-for-all-constructors-of-a-class
[21] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c52-use-inheriting-constructors-to-import-constructors-into-a-derived-class-that-does-not-need-further-explicit-initialization
[22] https://arne-mertz.de/2015/08/new-c-features-inherited-and-delegating-constructors/
[23] https://docs.microsoft.com/en-us/cpp/cpp/trivial-standard-layout-and-pod-types?view=msvc-170
[24] https://arne-mertz.de/2015/07/new-c-features-uniform-initialization-and-initializer_list/
[25] https://akrzemi1.wordpress.com/2016/07/07/the-cost-of-stdinitializer_list/
[26] https://blog.panicsoftware.com/objects-their-lifetimes-and-pointers/
[27] https://www.foonathan.net/2019/02/special-member-functions/
[28] https://lesleylai.info/en/const-and-reference-member-variables/
[29] https://blog.feabhas.com/2019/04/brace-initialization-of-user-defined-types/
[30] https://blog.feabhas.com/author/glennan/