

```
inline; class; struct; {}; default; delete; {}; (); int; private;
public; protected; =delete; && const; inline; class; struct; {};
default; delete; {}; (); int; private; public; protected; =delete; &&
const; inline; class; struct; {}; default; delete; {}; (); int; private;
public; protected; =delete; && const; inline; class; struct; {}
default; delete; {}; (); int; private; public; protected; =delete; &&
const; inline; class; struct; {}; default; delete; {}; (); int; private;
public; protected; =delete; && const & {}; default; delete; {}; ();
int; private; public; protected; =delete; && const; inline; class;
struct; {}; default; delete; {}; (); int; private; public; protected;
=delete; && const; inline; class; struct; {}; default; delete; {}; ();
int; private; public; protected; =delete; && const & {}; default;
delete; {}; (); int; private; public; protected; =delete; && const; {};
```

C++ Initialization Story

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

```
inline; class; struct; {}; default; delete; {}; (); int; private;
public; protected; =delete; && const; inline; class; struct; {}
default; delete; {}; (); int; private; public; protected; =delete; &&
const; inline; class; struct; {}; default; delete; {}; (); int; private;
public; protected; =delete; && const & {}; default; delete; {}; ();
int; private; public; protected; =delete; && const; inline; class;
struct; {}; default; delete; {}; (); int; private; public; protected;
=delete; && const; inline; class; struct; {}; default; delete; {}; ();
int; private; public; protected; =delete; && const & {}; default;
delete; {}; (); int; private; public; protected; =delete; && const; {};
```

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 2022-09-14



Leanpub

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.

© 2021 - 2022 Bartłomiej Filipek

Contents

About the Book	i
Why should you read this book?	i
Learning objectives	i
The structure of the book	ii
Who is this book for?	iii
Prerequisites	iii
Reader feedback & errata	iii
Example code	iv
Code license	iv
Formatting	iv
Special sections	vi
About the Author	vii
Acknowledgements	viii
Revision History	ix
1. Local Variables and Simple Types	1
Starting with simple types	2
Setting values to zero	5
Initialization with aggregates	6
Default data member initialization	8
Summary	10
2. Classes and Initialization With Constructors	11
A simple class type	11
Basics of constructors	14
Body of a constructor	24
Adding constructors to <code>DataPacket</code>	26

CONTENTS

Compiler-generated default constructors	28
Explicit constructors and conversions{#sectionexplicitctor}	31
Difference between direct and copy initialization	34
Implicit conversion and converting constructors	36
Constructor summary	37
3. Copy and Move Operations	38
Copy constructor	38
Move constructor	46
Distinguishing from assignment	54
Adding debug logging to constructors	57
Trivial classes and user-provided default constructors	61
4. Delegating and Inheriting Constructors	66
Delegating constructors	66
Limitations	69
Inheritance	70
Inheriting constructors	74
5. Destructors	76
Basics	76
Objects allocated on the heap	79
Destructors and data members	81
Virtual destructors and polymorphism	83
Partially created objects	87
A compiler-generated destructor	90
Summary and use cases	91
6. Initialization and Type Deduction	93
7. Quiz on Special Member Functions	94
8. Non-Static Data Member Initialization	96
How it works	97
Investigation	98
Experiments	98
Other forms of NSDMI	101
Copy constructor and NSDMI	104
Move constructor and NSDMI	106
C++14 changes	108

CONTENTS

C++20 changes	109
Limitations of NSDMI	110
NSDMI: Advantages and Disadvantages	114
NSDMI summary	115
NSDMI: Exercises	116
9. Containers as Data Members	121
The basics	121
Using <code>std::initializer list</code>	124
Example implementation	128
More options (advanced)	133
Summary	134
10. Non-regular Data Members	135
Constant non-static data members	135
References as data members	137
Pointers as data members	137
Moveable-only data members	137
Summary	137
11. Non-local objects	138
Storage duration and linkage	138
Initialization of non-local static objects	143
Static variables in a function scope	145
<code>constexpr</code> in C++20	145
About static data members	146
Motivation for inline variables	148
Exercise for inline variables	155
Global inline variables	156
<code>constexpr</code> and <code>inline</code> variables	157
Summary	158
12. Aggregates and Designated Initializers in C++20	159
Aggregates in C++20	159
The basics of Designated Initializers	162
Rules	163
Advantages of designated initialization	164
Examples	165
Summary	167

CONTENTS

13. Techniques and Use Cases	168
Using <code>explicit</code> for strong types	168
Best way to initialize <code>string</code> data members	172
The copy and swap idiom	176
CRTP class counter	178
Several initialization types in one class	181
Vector like RAII object	184
Factory with self-registering types	184
Summary	184
14. The Final Quiz	185
Appendix A - Rules for Special Member Function Generation	188
The diagram	188
Rules	191
Appendix B - Quiz and Exercises Answers	194
The quiz after Inheriting Constructors	194
The final quiz	194
Solution to the first coding problem	195
Solution to the second coding problem	195
Solution to the third coding problem	196
References	197

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 the 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 make some mistakes 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 is a short quiz on constructors. You can check your knowledge from the first “part” of the book.
- Chapter 7 (in progress): Type deduction.
- 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, `inline` variables from C++17 and `constexpr` 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 how to work with static and non-static class data members 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.

Here's the errata with the list of fixes:

<https://www.cppstories.com/p/cppinitbook/>

Your feedback matters! Writing an honest review can help with the book promotion and the quality of my further work.

What's more, the book has a dedicated page at GoodReads. Please share your feedback at:

[C++ Initialization Story by Bartłomiej Filipek @Goodreads¹](#)

¹<https://www.goodreads.com/book/show/61327713-c-initialization-story>

Example code

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

the link will appear later

You can browse individual files or download the whole branch:

the link will appear later

Code license

The code for the book is available under the Creative Commons License.

Formatting

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

For longer examples:

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);
}
```

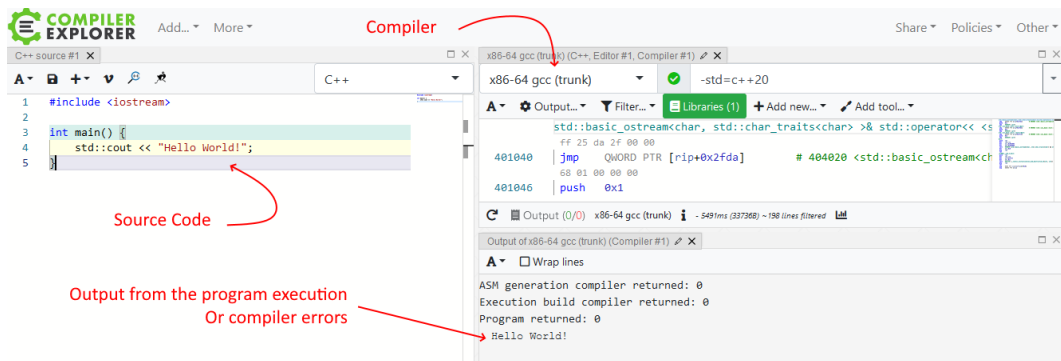
When available, you'll also see a link to online compilers where you can play with the code. For example:

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 were usually shortened to present only the core mechanics.

Syntax highlighting limitations

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

- `if constexpr` - Link to Pygments issue: [C++ if constexpr not recognized \(C++17\) · Issue #1136²](#).
- The first method of a class is not highlighted - [First method of class not highlighted in C++ · Issue #791³](#).

²<https://github.com/pygments/pygments/issues/1136>

³<https://github.com/pygments/pygments/issues/791>

- Template method is not highlighted [C++ lexer doesn't recognize function if return type is templated · Issue #1138](#)⁴.
- Modern C++ attributes are sometimes not recognized properly.

Other issues for C++ and Pygments: [C++ Issues · github/pygments/pygments](#)⁵.

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.

⁴<https://github.com/pygments/pygments/issues/1138>

⁵<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 a beautiful city 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](#)⁶, 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](#)⁷ (started as [bfilipek.com](#)⁸). 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](#)⁹. You can hear Bartek in one [@CppCast episode](#)¹⁰ 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](#)¹¹ and [C++ Lambda Story](#)¹²

⁶<http://www.xara.com/>

⁷<https://www.cppstories.com/>

⁸<https://www.bfilipek.com>

⁹<https://www.meetup.com/C-User-Group-Cracow/>

¹⁰<http://cppcast.com/2018/04/bartlomiej-filipek/>

¹¹<https://leanpub.com/cpp17indetail>

¹²<https://leanpub.com/cpluspluslambda>

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](#)¹³),
- Konrad Jaśkowiec (see his profile at [LinkedIn](#)¹⁴),
- Professor Boguslaw Cyganek (see his profile at [AGH university page](#)¹⁵),
- Dawid Pilarski (see his blog at [panicsoftware.com](#)¹⁶)
- Javier Estrada (see his blog at [javierestrada.blog](#)¹⁷),
- Jonathan Boccara (from [fluentcpp.com](#)/¹⁸),
- Andreas Fertig (see his blog at [andreasfertig](#)¹⁹),
- Peter Sommerlad (see his website and training info at [sommerlad.ch](#)/²⁰),
- Timur Doumler (see his website at [timur.audio](#)/²¹ and his [Twitter](#)²²).

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 feedback and comments from the blog readers, Patreon Discord Server (See [@C++Stories](#) [@Patreon](#)²³), and discussions at [C++ Polska](#)²⁴. Thank you all!

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

¹³<https://www.linkedin.com/in/florin-ioan-chertes-41b6845/>

¹⁴<https://pl.linkedin.com/in/konrad-ja%C5%9Bkowiec-84585159>

¹⁵<https://home.agh.edu.pl/~cyganek/>

¹⁶<https://blog.panicsoftware.com/>

¹⁷<https://javierestrada.blog/>

¹⁸<https://www.fluentcpp.com/>

¹⁹<https://andreasfertig.blog/>

²⁰<https://sommerlad.ch/>

²¹<https://timur.audio/>

²²https://twitter.com/timur_audio

²³<https://www.patreon.com/cppstories>

²⁴<https://cpp-polska.pl/>

Revision History

- 20th June 2022 - The first public version! The book is almost done. 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](#).

1. Local Variables and Simple Types

Let's start simple and ask, "what is initialization?" When we go to the definition from [C++Reference¹](https://en.cppreference.com/w/cpp/language/initialization), 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.

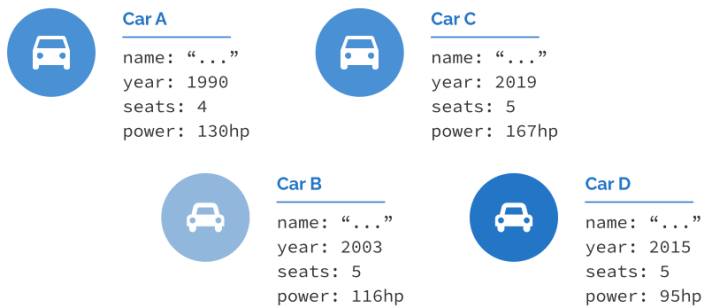
While we can explain most cases on integers and other numerical types, it’s best to work on something more practical. The book starts with some elementary custom types, then considers various issues we might have with their early implementations. Later the types will expand, giving us more context and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

²In 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 longLine {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 [@Wikipedia¹](https://en.wikipedia.org/wiki/Checksum)).

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(), 0uz);
}

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>

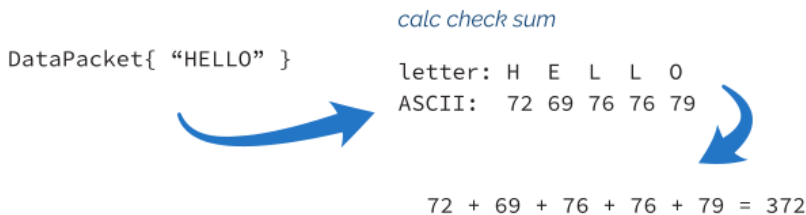
```

    void setData(const std::string& data) {
        data_ = data;
        checksum_ = calcChecksum(data);
    }
    size_t getChecksum() const { return checksum_; }
    size_t getServerId() const { return serverId_; }
};

```

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 Guide²](https://google.github.io/styleguide/cppguide.html#Variable_Names).

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 and adds numerical values of letters from the input `std::string`. Since C++23, we can use the `0UZ` integer literal to represent the `size_t` so that it matches with the return type for the function; alternatively, we could use `static_cast<size_t>` or `UL/ULL` for 32/64-bit systems respectively. 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

²https://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 the `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 outside⁵. In C++, you can also add `protected` to your class declaration, which means that member functions or fields are not accessible outside, but they are accessible to all inherited classes (assuming `public` inheritance, see more on that in the [inheritance section](#) further in the book).

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

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

³<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#con2-by-default-make-member-functions-const>

⁴https://en.cppreference.com/w/cpp/language/noexcept_spec

⁵Unless accessed by `friend` functions or classes.



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](#)⁶, 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](#)⁷.

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.

Basics of constructors

A constructor is a special member function that does not have a name, but we declare/define it using the enclosing class name⁸. 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:

⁶https://google.github.io/styleguide/cppguide.html#Structs_vs._Classes

⁷<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c8-use-class-rather-than-struct-if-any-member-is-non-public>

⁸See the full definition at <https://timsong-cpp.github.io/cppwp/n4868/class.ctor.general>

```

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`, and `constexpr` 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](https://en.cppreference.com/w/cpp/language/constructor)⁹.

For illustrative purposes, you can find a simple class type with two data members below. The `Product` class will serve as a toy example, and then we'll apply the knowledge to the `DataPacket` class I plan to update. Let's have a look at one snippet:

⁹<https://en.cppreference.com/w/cpp/language/constructor>

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

The above example shows a class 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 in the [explicit constructors section](#).

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 and *wrong* alternative:

```
class Product {  
public:  
    Product() { id_ = 0; name_ = "none"; } // bad code, only for illustration  
  
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. Please don't write such code and aim for a member initializer list to initialize your data members efficiently.

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 Explorer¹¹](#)):

```

<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
    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

The syntax with curly braces "{}" is, in fact, a powerful feature of C++11 called *list initialization*, also called "uniform" or "brace" initialization. The primary motivation was to create a uniform way to initialize data and avoid some issues.

For example, because of the C++ language grammar rules, the following line won't compile:

Ex 2.5. The Most Vexing Parse Rule. Run @Compiler Explorer

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

struct Box { };

struct Product {
    Product(): name{"default product"} { }
    Product(const Box& b) : name{"box"}{ }
    std::string name;
};

int main() {
    Product p();           // << 1.
    std::cout << p.name;
    Product p2(Box());    // << 2.
    std::cout << p2.name;
}
```

The line `Product p();` looks innocent, and one could expect a default constructor to be called. Unfortunately, the compiler recognizes it as a declaration of a function! There's a C++ rule which says that anything that can be parsed as a declaration must be interpreted as one. In our context, the line might mean a local function of a name `p` returning `Product` and taking no arguments. Similarly, the line with `p2` also causes compiler errors, and this time the compiler thinks we declare a local function `p2` returning a `Product` and taking `Box` as an argument.

But fortunately, we have at least two ways of fixing it:

```
Product p{};
Product p1;
Product p2{Box()};
Product p3{Box{}};
```

It works as expected, and the list initialization syntax seems to be the most consistent option. Try to modify the example @Compiler Explorer¹² and fix the code. Here's a good article if

¹²<https://godbolt.org/z/c48K7c9vq>

you want to know more about this rule: [The Most Vexing Parse: How to Spot It and Fix It Quickly - Fluent C++¹³](https://www.fluentcpp.com/2018/01/30/most-vexing-parse/).

List initialization also handles multiple arguments and can be used inside *initializer lists*:

Ex 2.6. Multiple arguments and braces. Run [@Compiler Explorer](#)

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

struct Product {
    Product()
        : name{"default product"}
        , value{}
    { }
    Product(char a, char b, char c, double v)
        : name{a, b, c}
        , value{v}
    { }

    std::string name;
    double value;
};

int main() {
    Product def{};
    std::cout << def.name << ", " << def.value << '\n';
    Product p{'x', 'y', 'z', 100.0};
    std::cout << p.name << ", " << p.value;
}
```

In the above example, we not only used list initialization to call the `Product` constructor with four arguments, but we also used it to initialize the `name` and `value` data members.

What's more, the curly list initialization has the following advantages:

- the syntax is similar to aggregate initialization,
- adds a way to initialize containers with a list of the objects. For example
`std::vector<int> v { 1, 2, 3, 4 },`

¹³<https://www.fluentcpp.com/2018/01/30/most-vexing-parse/>

- allowing for a safer way of initialization that checks for narrowing. For example, `int v{10.3}` won't compile and reports a narrowing error, while `int v(10.3)` works and might produce an unwanted result.

There are some annoyances with list initialization through. For example:

```
std::vector<int> vec1 { 1, 2 }; // holds two values, 1 and 2
std::vector<int> vec2 ( 1, 2 ); // holds one value, 2!
```

Above you can see a very similar declaration of vectors, but when used with list initialization, you end up with a different vector than when using direct initialization. The list version calls a special constructor taking `std::initializer_list<int>`, while the second calls a constructor taking `(size_type count, const int& value = int())`.

Additionally, for auto type deduction in C++14:

```
auto i = 42; // i is an int with value 42
auto j(42); // j is an int with value 42
auto k{42}; // k is a std::initializer_list<int> until C++17!
```

Fortunately, this “inconsistency” was fixed in C++17, and now `auto k{42}` deduces an `int`. See more [C++17 in details: fixes and deprecation @C++ Stories¹⁴](#).

C++ Core Guidelines suggest sticking to this way of initialization, as its benefits outweigh the negative sides. See [ES.23¹⁵](#)

ES.23: Prefer the {}-initializer syntax

Reason: Prefer {}. The rules for {} initialization are simpler, more general, less ambiguous, and safer than for other initialization forms. Use = only when you are sure there can be no narrowing conversions. For built-in arithmetic types, use = only with auto. Avoid () initialization, which allows parsing ambiguities.

The guideline also mentions some exceptions:

¹⁴<https://www.cppstories.com/2017/05/cpp17-details-fixes-deprecation/#new-auto-rules-for-direct-list-initialization>

¹⁵<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es23-prefer-the--initializer-syntax>

Exception: For containers, there is a tradition for using {...} for a list of elements and (...) for sizes: `vector<int> v(10);` // 10 elements with the default value 0

`vector<int> v2{10};` // vector of 1 element with the value 10

In this book, I'll follow the pattern of using {} for variable initialization and mention exceptions if needed.

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:
    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 (in C++20, `std::string` might be used in the `constexpr` context but not for variables available at runtime); 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](https://en.cppreference.com/w/cpp/language/constexpr)¹⁶.

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

¹⁶<https://en.cppreference.com/w/cpp/language/constexpr>


```

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}
    { }

    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](#)¹⁷.

¹⁷https://en.cppreference.com/w/cpp/language/default_constructor#Deleted_implicitly-declared_default_constructor



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 and conversions{#sectionexplicitctor}

Before we move on, it's essential to tackle one important case: the `explicit` keyword, which can be applied before a constructor declaration.

Why is it important? And what does this keyword mean?

In short, it prevents implicit conversions and might sometimes make code easier to read.

As an experiment, if we start with the following code:

```
struct Product {
    Product()
        : name{"default product"}
        , value{}
    { }
    Product(int v)
        : name{"basic"}
        , value{v}
    { }
    Product(const std::string& n, int v)
        : name{n}
        , value{v}
    { }

    std::string name;
    int value;
};
```

The code looks fine, but now you can create `Product` objects in a bit unusual way:

```
Product numbers = 100.2;    // copy initialization
Product box = {"a box", 1}; // copy list-initialization
```

We can read that those two lines create products, but what values do the data members get? It's not clear! The case in the first line is especially interesting, as I passed a double value of 100.2, and the compiler tried to convert it into the `int` type (a narrowing conversion) and then passed it to the constructor.

What's more, it's even more problematic with implicit conversions for function calls:

Ex 2.12. Implicit Conversions. Run [@CompilerExplorer](#)

```
void printProduct(const Product& prod) {
    std::cout << prod.name << ", " << prod.value << '\n';
}

int main() {
    double someRandomNumber = 100.1;
    printProduct(someRandomNumber);
    printProduct({"a box", 2});
}
```

The output:

```
basic, 100
a box, 2
```

The key idea is to understand: when you pass arguments into a function call, then the compiler performs copy initialization on the arguments.

As you can see, the main issue is with constructors that take only one argument (or have other arguments set to some default value). But even with several arguments, the conversion can happen when you pass an initialization list.

To prevent such unwanted and unexpected conversions, it's good to apply the `explicit` keyword.

When we apply it:

```

explicit Product(int v)
    : name{"basic"}
    , value{v}
{ }

explicit Product(const std::string& n, int v)
    : name{n}
    , value{v}
{ }

```

The compiler will report the following errors:

```

In function 'int main()':
error: invalid initialization of reference of type 'const Product&'
from expression of type 'double'
    28 |         printProduct(someRandomNumber);
        |                        ^~~~~~
error: converting to 'const Product' from initializer list would use
explicit constructor 'Product::Product(const std::string&, int)'
    29 |         printProduct({"a box", 2});
        |         ~~~~~^~~~~~

```

See the complete example [@Compiler Explorer¹⁸](#).

To fix the code, you need to tell the compiler to create a type explicitly:

```

int someRandomNumber = 100;
printProduct(Product{someRandomNumber});

```

From a practical point of view, the case with a multi-parameter constructor and initializer list is not an issue. The compiler will call the proper constructor and won't perform any narrowing conversions. That's why usually, there's no sense in marking multi-parameter constructors with `explicit`. For example:

```

Product toy = {"a toy", 1.5};
printProduct({"a box", 2.0});

```

¹⁸<https://godbolt.org/z/3KT5MfnT8>

The code generates errors about narrowing conversion ... from double to int. See the code [@Compiler Explorer¹⁹](#).



Constructors not declared with the `explicit` keyword, also called *converting constructors*. They take part in the implicit conversion sequence. In C++03, those constructors must also be callable with a single argument, but that limitation was lifted in C++11. More on the implicit conversion in a separate section in this chapter.

Difference between direct and copy initialization

Finally, after addressing several examples and explicit constructors, we can try to answer a question about the differences between direct vs. copy initialization.

We have two primary forms of initialization: copy and direct:

```
// copy:
int x = 42;           // a form of a copy initialization

void foo(int param) { }
foo(x);              // copy initialization is performed on the argument

int anotherFoo() { return 42; } // a copy initialization is done
                               // on the return value

struct Point { int x; int y; };
Point pt { 0, 1 };      // aggregate initialization
Point p2 = { 10, 11 };  // uses copy initialization for each element
```

And here's the basic syntax for the direct initialization:

¹⁹<https://godbolt.org/z/7Kab4eT5T>

```
// direct
int y {42};           // a form of a direct initialization
double z (42.2);      // direct with parens
```

In summary:

- Direct initialization behaves like a function call to an overloaded function: The functions, in this case, are the constructors of the type (including explicit ones). Overload resolution will find the best matching constructor and, when needed, will do any implicit conversion required.
- Copy initialization constructs an implicit conversion sequence: It tries to convert arguments to an object of the given type. Explicit constructors are not considered for copy initialization.

For example, since aggregate initialization uses copy initialization to init subobjects, then this code won't work:

```
struct Point {
    explicit Point(int a, int b):
        x{a}, y{b}
    { }

    int x;
    int y;
};

struct Aggregate {
    int a;
    Point p;
};

int main() {
    Aggregate ag { 0, {0, 1}};    // <<
    Aggregate ag2 = { 0, {0, 1}}; // <<
}
```

GCC reports the following error:

```
error: converting to 'Point' from initializer list would use explicit construct\
or 'Point::Point(int, int)'
```

```
19 |     Aggregate ag { 0, {0, 1}};
```

To fix this, you need to mention the type name explicitly:

```
int main() {
    Aggregate ag { 0, Point{0, 1}};
    Aggregate ag2 = { 0, Point{0, 1}};
}
```

See the working code [@Compiler Explorer²⁰](#).

Even more

The `explicit` keyword is so important that it has its own rule in C++ Core guidelines: [C++ Core Guidelines - C.46²¹](#).

C.46. By default, declare single-argument constructors `explicit`

Reason: To avoid unintended conversions.



Additionally, in C++20, we have an extended syntax `explicit(bool)` to mark explicit constructors conditionally. This is a bit advanced feature, so we won't address this in this book. You can read more [@C++Reference²²](#).

Implicit conversion and converting constructors

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

²⁰<https://godbolt.org/z/d8cronMM1>

²¹<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-explicit>

²²<https://en.cppreference.com/w/cpp/language/explicit>

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.

3. Copy and Move Operations

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. Additionally, you can provide custom assignment operators that the compiler calls when you assign new values to existing objects. 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 a `const` reference.

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

Technically it might have other parameters, but they all have to have default values assigned, and in practice, it's very uncommon.

It's used and called when you initialize an object using a variable of the same type (through copy initialization or direct initialization), and there's no better match (like a move constructor).

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

// various forms of initialization, where a copy constructor is called
Product other { base }; // direct list initialization
Product another(base); // direct initialization with parens
Product oneMore = base; // copy initialization
std::array<Product, 3> arr = { base, other, oneMore }; // copy initialization

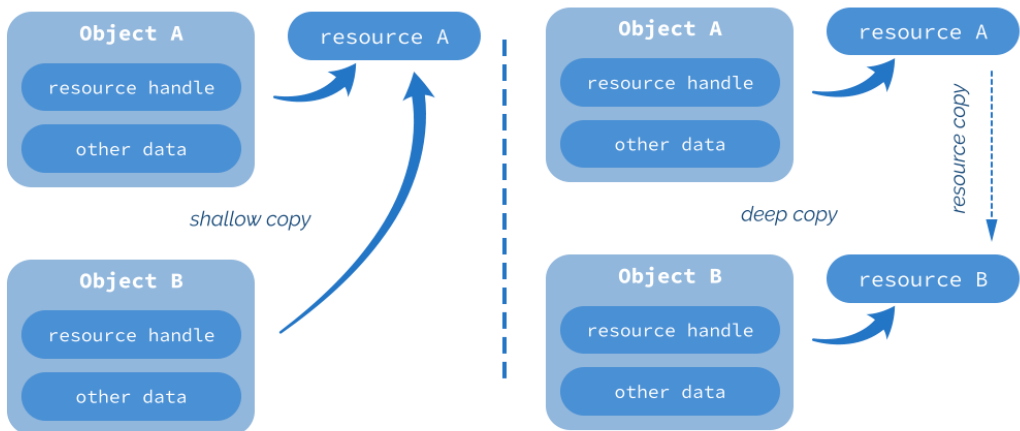
Product foo(Product p) {
    Product temp{"from foo", p.id};
    return temp; // copy initialization
}
```

```
}  
Product x;  
foo(x); // copy initialization
```

We'll discuss all of the above forms in the further section.

As a mental model, we can assume that most classes have their default, compiler-generated copy constructor. Such a default copy constructor performs a bitwise copy of each data member. Thanks to this approach, you can initialize objects, pass them as arguments or return from functions without any custom implementation in a given type. However, implementing a copy constructor might be necessary when your class has data members that shouldn't be shallow copied, like owned resource handles (files, pointers to memory blocks, etc.). For example, suppose a class contains a pointer to a memory block. When a default copy constructor copies such a pointer, the resulting pointer copy will point to the exact memory location. Similarly, if your class uses a handle to a file, then after the bitwise copy, the handle will be copied, and two objects will relate to the same file. Depending on the program requirements, this might not be what you want. In a case with pointers, it's usually better to allocate a new memory block and copy the data of that block.

The situation for a shallow copy is illustrated by the following diagram:



Shallow vs. Deep Copy

On the diagram, you can see a comparison of a shallow (bitwise copy) vs. the deep copy approach. If you copy the resource handles, the resulting object will point to the same

resource. For the full version, you need to copy the resource and then correctly assign the resource handle to that new copy.

Standard library containers like `std::vector` or `std::string` internally contain pointers to memory buffers to store the elements. They all have support for full data copy, so when you copy one vector to another, the memory buffers will also be copied. You don't have to think about those internal mechanisms when using those types in your code.

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:
    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 syntax 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 make code harder to read and understand. It might be better to use move semantics and move constructors when you want to “steal” the guts of some other object.



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

Here’s another example where basic logging is enabled. Such console output is helpful to see how and what constructors are called:

Ex 3.1. An example of a logging copy constructor. Run [@Compiler Explorer](#)

```
class Product {
public:
    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_;
};
```

And here’s the code that creates `Product` objects:

Ex 3.1. An example of a logging copy constructor, demo. Run [@Compiler Explorer](#)

```
1 void foo(Product p) {
2     std::cout << "inside foo()\n";
3 }
4
5 int main() {
6     Product base { 42, "base product" }; // an initial object
7     std::cout << base.Name() << " created\n";
8     std::cout << "Product other { base }; \n";
9     Product other { base };
10    std::cout << "Product another(base); \n";
11    Product another(base);
12    std::cout << "Product oneMore = base; \n";
13    Product oneMore = base;
14    std::cout << "std::array<Product, 2> = { base, other }; \n";
15    std::array<Product, 2> arr = { base, other };
16
17    std::cout << "calling foo()\n";
18    foo(arr[0]);
19 }
```

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
std::array<Product, 2> = { base, other };
Product(copy): 42, base product
Product(copy): 42, base product
calling foo()
Product(copy): 42, base product
inside foo()
```

In line 10, we construct base `product`, and then use it to copy-construct all other instances: `other`, `another`, and `oneMore`. Each time a copy constructor is called. The same happens for the array `std::array<Product, 2>`.

Later we call a function `foo()`, and when you pass an argument as a value, a copy has to be created using a copy constructor call.

Copy elision

Now, let's consider the following code:

Ex3.2.NamedCopyElision.Run@CompilerExplorer

```
Product createProduct() {
    Product temp{101, "from createProduct()"};
    return temp;
}

int main() {
    std::cout << "calling createProduct()\n";
    Product created = createProduct();
}
```

The output is:

```
calling createProduct()
Product(): 101, from createProduct()
```

This result contradicts what I wrote before: a copy constructor should be called for return statements. Technically it should, but the output shows a regular constructor only.

This feature is a compiler optimization that allows it to “elide” such extra object copies. To be precise, it's called Named Return Value Optimization (NRVO¹), as there's a named variable that we “reuse”. The compiler can see through the initialization, deduce that the `temp` object is used only to initialize `created` and can “compress” the creation steps. The GCC compiler has a switch to turn off such optimization: `-fno-elide-constructors`.

If you compile with that flag, you should be able to see:

¹https://en.cppreference.com/w/cpp/language/copy_elision

```
calling createProduct()
Product(): 101, from createProduct()
Product(copy): 101, from createProduct()
```

Have a look [@Compiler Explorer²](#)

But there's more!

Starting from C++17, there's a mandatory copy elision, also called “deferred temporary materialization”. While the previous example was an optional compiler optimization, we have language rules that clearly express the new behavior this time. Not going into details, it will elide additional copies when there's an unnamed temporary object from which we initialize a new entity: For example:

Ex3.3.CopyElision

```
Product createProduct() {
    return {101, "from createProduct()"};
}

int main() {
    std::cout << "calling createProduct()\n";
    Product created = createProduct();
}
```

This time the compiler will always generate the following output:

```
calling createProduct()
Product(): 101, from createProduct()
```

In other words, the temporary from `createProduct` is skipped and used to initialize the created object directly. This feature is helpful for optimization and efficiently working with non-copyable types that previously couldn't be returned from factory functions.

If you want to know more about this feature, have a look at my book: [C++17 in Detail³](#) or see this blog post: [Guaranteed Copy Elision Does Not Elide Copies⁴](#) @VisualC++ Team Blog.

²<https://godbolt.org/z/49M1GaxK6>

³<https://leanpub.com/cpp17indetail>

⁴<https://devblogs.microsoft.com/cppblog/guaranteed-copy-elision-does-not-elide-copies/>

A compiler-generated copy constructor

The compiler will generate an implicit copy constructor for you if your class complies with the following key rules:

- Your class has non-static data members that can be copied (their copy constructors are accessible, not delete);
- Your class has a direct or virtual base class that can be copied;
- Your class doesn't have any data members of rvalue reference type;
- Your class doesn't have a user-defined move constructor or move assignment operator.

You can find all the rules in this handy list [@C++Reference](#)⁵.

As an example, let's have a look at the following code:

Ex 3.4. A non-default copy constructor. Run [@Compiler Explorer](#)

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

struct Name {
    explicit Name(const std::string& str): name_{str} { }
    Name(const Name&) = delete;
    std::string name_;
};

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

private:
    int id_;
    Name name_;
};
```

⁵https://en.cppreference.com/w/cpp/language/copy_constructor#Deleted_implicitly-declared_copy_constructor

```
int main() {
    Product first{10, "basic"};
    Product second { first };
}
```

Please look at the line where we are trying to call the copy constructor. It won't compile:

```
<source>:10:7: note: 'Product::Product(const Product&)' is implicitly
deleted because the default definition would be ill-formed:
 10 | class Product {
    |           ^~~~~~
```

The compiler tells us it cannot create a copy constructor because the name data member cannot be easily copied, as it has deleted the copy constructor.

We can also observe this by looking at the output from C++Insights [@see this link](#)⁶:

```
public:
// inline Product(const Product &) = delete;
// inline Product(Product &&) = delete;
// inline ~Product() noexcept = default;
```

Move constructor

Move constructors take rvalue references of the same type. Usually, such a constructor has the following form:

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

Let's try to decipher the full syntax.

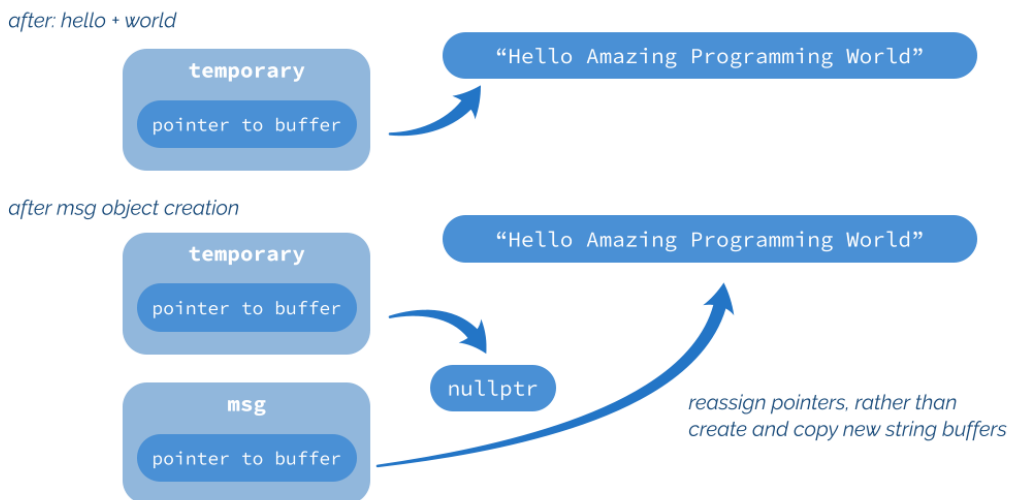
In short, rvalue references are temporary objects, usually appearing on the right-hand side of an expression and which value is about to expire.

For example:

⁶<https://cppinsights.io/s/15ea2cb1>

```
std::string hello { "Hello Amazing"}; // lvalue, a regular object
std::string world { " Programming 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`⁷), 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. The situation is illustrated with the following diagram:



Idea of a move constructor

The diagram illustrates the state after computing `hello + world` and later when `msg` is initialized. The compiler creates a temporary object with the long string, and that string is stored in a buffer allocated outside the string. The string object has a pointer to that buffer. Later, `msg` is created from that temporary object. We know that the object will expire so that we can reassign the pointers to the memory buffers. `msg` gets a pointer to the long string, and the temporary object gets `nullptr` (conceptually, as the internal implementation might differ).

Move constructors are a way to support the case with initialization from temporary objects.

⁷The 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>.

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 3.5. Move Constructor. Run [@Compiler Explorer](#)

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

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

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

    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) noexcept  
    : id_(other.id_), name_(std::move(other.name_))
```

The above implementation is similar to a copy constructor, 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. 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

While I mentioned that `noexcept` wouldn't be covered in this book, I need to make one exception to this rule. The fundamental principle for `noexcept` on a function declaration is to guarantee that the function won't return any exceptions (won't throw from the function scope). If it does, the compiler can call `std::terminate()` instead of regular exception handling. Having a `noexcept` function allows the compiler and the libraries to optimize the code.

For example, when you have a `std::vector` of `T`, then if `T` has move operations marked with `noexcept`, then the vector is allowed to perform resize operations with move rather than copy (to guarantee safety). To illustrate this behavior, I modified the `Product` class and added a copy constructor:

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

Notice that there's no `noexcept` in the move constructor. Now, if we run the following demo code:

Ex 3.6. Copy on resize for `std::vector`. Run [@Compiler Explorer](#)

```
int main() {
    std::vector<Product> prods;
    prods.emplace_back(101, "car");
    prods.emplace_back(202, "box");
    prods.emplace_back(303, "toy");
    prods.emplace_back(404, "mug");
    prods.emplace_back(505, "pencil");
}
```

We'll see the following output:

```

Product(): 101, car
Product(): 202, box
Product(copy): 101, car
Product(): 303, toy
Product(copy): 101, car
Product(copy): 202, box
Product(): 404, mug
Product(): 505, pencil
Product(copy): 101, car
Product(copy): 202, box
Product(copy): 303, toy
Product(copy): 404, mug

```

Let's try to decipher the output.

The `emplace_back` function (available since C++11) creates a new element at the end of the container using the arguments you pass. Alternatively, we could use `push_back`, but this requires an additional copy of the `Product` object. When we add the first element, you can see that a regular constructor is called. Now, with the second element, the vector must grow its internal buffer and copy existing elements to a new buffer. That's why you can see a regular constructor for the "box" object and then a copy constructor for "car". Similarly, when I add the third element, its constructor is called, and then copies of "car" and "box" must be invoked. Later the process continues as we add more elements and the container grows. It's implementation-specific, but usually, `std::vector` might grow 1.5x or 2x each time it has to resize. For example, it starts with one element and capacity of one, then two elements and capacity of 2, 3 elements and capacity of 4, 5 elements and capacity of 6 or 8, and so on. This helps to amortize the cost of adding new values.

Now, let's modify the move constructor and make it `noexcept`:

Ex 3.7. Move on resize for `std::vector`. Run [@Compiler Explorer](#)

```

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

```

When we run the code, you'll see the following log:

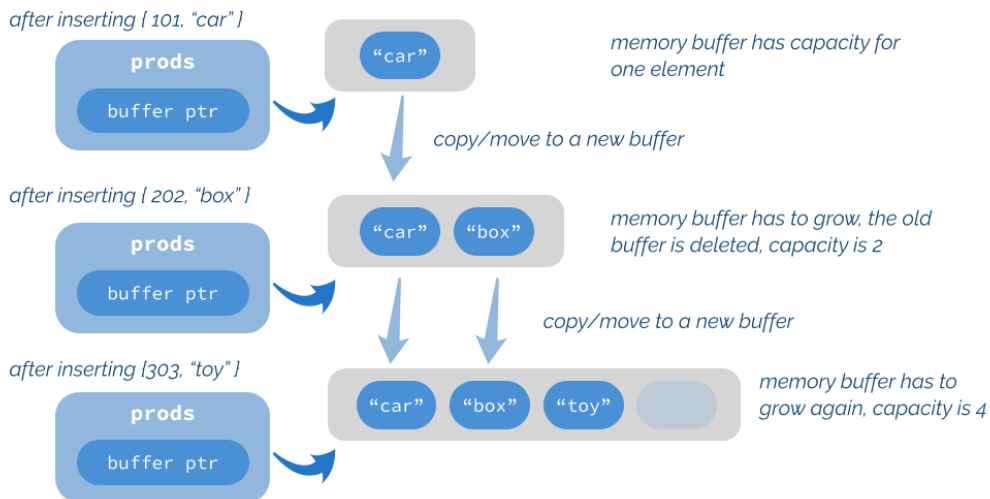
```

Product(): 101, car
Product(): 202, box
Product(move): 101, car
Product(): 303, toy
Product(move): 101, car
Product(move): 202, box
Product(): 404, mug
Product(): 505, pencil
Product(move): 101, car
Product(move): 202, box
Product(move): 303, toy
Product(move): 404, mug

```

Now, the compiler calls a move constructor rather than a copy! In many cases, this can be much faster than copying data, as we can just copy pointers rather than copying the entire content of a string. It's implementation-dependent if the library uses that optimization technique, but MSVC, GCC, and Clang library implementations stick to this rule.

Below you can find a basic illustration of this “growth” process:



Vector resize process for three elements

On the diagram, on the left-hand side, you can see the `prods` vector that has a pointer to a memory buffer with all elements. The vector class usually contains other data members, like size and capacity, but we'll stick to the simple model. After inserting the first element, the

buffer has a capacity for only one object and then has to grow if we add more values. In the third line, you can see a new buffer with three elements but a “transparent” spot for a fourth one. Each time a buffer is recreated, it must copy/move existing elements.

The reason for this technique is that when the move constructor is not marked with `noexcept` then the container has to be prepared for a case where it tries to copy elements to a new buffer, and at some point, one operation throws. The only way to revert to a save situation is to abandon the copy. When move `noexcept` is available, then the vector implementation can assume that there’s no exception happening, and moving will be “safe” for all elements. Other algorithms from the Standard Library, like `std::sort`, might also benefit from having `noexcept` guarantees on move operations.

You can read more about this approach in the following C++ Core Guideline: [C.66: Make move operations `noexcept`](#)⁸. And also in this detailed article by Andrzej Krzemiński: [Using `noexcept` @Andrzej’s C++ blog](#)⁹.

A compiler-generated move constructor

Don’t worry! If your class uses primitive types or types from the Standard Library, there’s no need to write custom move constructors. In most cases, the compiler creates a default implementation assuming your class complies with the following rules:

- There are no user-declared copy constructors.
- There are no user-declared copy assignment operators.
- There are no user-declared move assignment operators.
- There is no user-declared destructor.
- Non-static data members all have accessible move constructors.
- Your class has a direct or virtual base class that can be moved.
- Your class has a virtual base class or a non-static data member without a deleted or inaccessible destructor.

The rules are logical. For example, if you declare a custom copy constructor, there’s a high chance your class is unique, and thus move operations should also be implemented by you rather than the compiler.

You can find all the rules in this handy list [@C++Reference](#)¹⁰.

⁸<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c66-make-move-operations-noexcept>

⁹<https://akrzemi1.wordpress.com/2011/06/10/using-noexcept/>

¹⁰https://en.cppreference.com/w/cpp/language/move_constructor#Implicitly-declared_move_constructor

Distinguishing from assignment

It's crucial that you recognize a case where the compiler invokes a copy or move constructor versus a case with the assignment operator. The code might look similar, but they behave differently.

C++, as a powerful capability, allows us to implement operators for user-defined types. Such operators make it easier and more natural for operations related to math, string manipulation, relations, and stream output or input, among others. For example `std::string` implements `operator+` for string concatenation. Similarly, you can define `operator-` for classes representing 3D Vectors in space. You can see the full list of operators on [operator overloading @C++Reference](#)¹¹.

If you don't provide a custom declaration, then the compiler attempts to define an implicit version of it. While most operators won't work for user-defined types and simply don't exist until you implement them, there's a special `operator=`, called the assignment operator. By default, the implicit version calls the assignment operator for sub-objects of the given type. For example:

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

Point pt { 10, 10 };
Point another { 100, 100 };
another = pt;           // implicit assignment operator called!
```

Even though I don't provide any implementation for `operator=` the compiler defines it, and I can write the assignment: `another = pt`. By default, the operator copies data members from `pt` into `another`. Since the `Point` type is trivial and uses built-in types for its data members, we'll get a bitwise copy of `pt`.

In a basic form, we can implement the assignment operator in at least two forms:

```
// copy assignment operator
Product& operator=(const Product& other) { /* ... */}
```

Or

¹¹<https://en.cppreference.com/w/cpp/language/operators>

```
// move assignment operator
Product& operator=(Product&& other) { /* ... */}
```

The copy assignment operator will be called when there's an lvalue on the right-hand side of the assignment expression. The move assignment is called when there's an rvalue reference. See the following code, where I implemented a copy assignment for the Product class:

Ex 3.8. Copy assignment for **Product**. Run [@Compiler Explorer](#)

```
class Product {
public:
    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';
    }

    Product& operator=(const Product& other) {
        if (this == &other)
            return *this;

        id_ = other.id_;
        name_ = other.name_;
        std::cout << "operator=(copy): " << id_ << ", " << name_ << '\n';
        return *this;
    }

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

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

And here's the demo code:

```
Product base { 42, "base" };
Product first { base }; // copy ctor called!
Product second = first; // copy ctor called!
Product third { 100, "third" };
third = second; // assignment operator called!
```

The compiler calls a copy constructor when you initialize an object. When an entity already exists, the program runs an assignment operation.

And here's the corresponding version of the move assignment operator:

Ex 3.9. Move assignment for **Product**. Run [@Compiler Explorer](#)

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

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

    Product& operator=(Product&& other) noexcept {
        id_ = other.id_;
        name_ = std::move(other.name_);
        std::cout << "operator=(move): " << id_ << ", " << name_ << '\n';
        return *this;
    }

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

private:
    int id_;
```

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

```
Product base { 42, "base" };
Product first { std::move(base) }; // move ctor called!
std::cout << "base.name() after move: " << base.name() << '\n';
Product second = std::move(first); // move ctor called!
std::cout << "first.name() after move: " << first.name() << '\n';
Product third { 100, "third" };
third = std::move(second); // assignment operator called!
std::cout << "second.name() after move: " << second.name() << '\n';
```

Above, we can explicitly ask to call a move constructor when we use `std::move` and mark an object as expiring. When an object exists and we assign a new value, the assignment operator will be called. If we run the code above, you'll get the following output:

```
Product(): 42, base
Product(move): 42, base
base.name() after move:
Product(move): 42, base
first.name() after move:
Product(): 100, third
operator=(move): 42, base
second.name() after move:
```

As you can see, after we move from an object, it's left in an unspecified but valid state. For strings, it means an empty string.

Adding debug logging to constructors

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

Ex 3.10. Logging in the `DataPacket` class. Run [@Compiler Explorer](#)

```

1  class DataPacket {
2      std::string data_;
3      size_t checksum_;
4      size_t serverId_;
5
6  public:
7      DataPacket()
8          : data_{}
9          , checksum_{0}
10         , serverId_{0}
11         { }
12
13     DataPacket(const std::string& data, size_t serverId)
14         : data_{data}
15         , checksum_{calcChecksum(data)}
16         , serverId_{serverId}
17         {
18             std::cout << "Ctor for \"" << data_ << "\"\n";
19         }
20
21     DataPacket(const DataPacket& other)
22         : data_{other.data_}
23         , checksum_{other.checksum_}
24         , serverId_{other.serverId_}
25         {
26             std::cout << "Copy ctor for \"" << data_ << "\"\n";
27         }
28
29     DataPacket(DataPacket&& other) noexcept
30         : data_{std::move(other.data_)} // move string member...
31         , checksum_{other.checksum_}    // no need to move built-in types...
32         , serverId_{other.serverId_}
33         {
34             other.checksum_ = 0; // leave this in a proper state
35             std::cout << "Move ctor for \"" << data_ << "\"\n";
36         }
37

```

```

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) noexcept {
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 3.11. 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.



The logging part in the example is a bit crude, as the class directly calls a global stream object. In some cases, this might complicate unit testing or using the class in general. It would be better to rely on some configurable “tracing/logging” object that could be passed as a parameter to the constructor. We’ll tackle that approach in the section about [references as static data members](#).

Trivial classes and user-provided default constructors

We covered the basics of default, copy and move constructors. And now we can try to answer a question that might appear several times before: what's the difference between `= default`, declaring a special member function `empty {}`, or not declaring a thing at all?

Let's have a look at the example; assuming we have several similar classes `MyTypeX`:

```
struct MyType0 {
    int x;
};

struct MyType1 {
    MyType1() { }
    int x;
};

struct MyType2 {
    MyType2() = default;
    int x;
};

struct MyType3 {
    MyType3();
    int x;
};

MyType3::MyType3() = default;
```

As you can see, there are four ways you can end up with an “empty” constructor and implicit default copy constructors. The core difference between them is that in `MyType0` and `MyType2` default constructors are considered **not user-provided**. Such a term has some consequences in the C++ Standard. For example both `MyType0` and `MyType2` are considered *trivial*.

First of all: what does it mean “user-provided”. From the Standard [dcl.fct.def.default#5](https://ericniebler.com/2017/01/01/default-constructors/)¹²:

¹²<https://ericniebler.com/2017/01/01/default-constructors/>

A function is user-provided if it is user-declared and not explicitly defaulted or deleted on its first declaration.

```
struct X {  
    X() = default;           // 1  
    X(const X&) { /*...*/ }  // 2  
};
```

Above, the first declaration `// 1` is not user-provided. It's user-declared (since we declare it), but since we explicitly `=default` it, it's not considered user-provided. On the other hand, the copy constructor is user-provided since we provide a custom implementation of it. The same happens for `MyType0`, where we don't provide any special member function, or for `MyType2`, where we declare a default constructor, but we explicitly make it `default`, and thus it's not user-provided.

According to [the C++ Standard](#)¹³:

A trivial class is a class that is trivially copyable and has one or more eligible default constructors, all of which are trivial.

And a *trivially copyable* class is:

A trivially copyable class is a class:

- that has at least one eligible copy constructor, move constructor, copy assignment operator, or move assignment operator,
- where each eligible copy constructor, move constructor, copy assignment operator, and move assignment operator is trivial, and
- that has a trivial, non-deleted destructor.

¹³<https://timsong-cpp.github.io/cppwp/n4868/class.prop#1>

Now, we have to understand *a trivial special member function*:

For default constructors, see [this section in the C++ Standard - class.default.ctor#3](#)¹⁴:

A default constructor is trivial if it is not user-provided and if:

- its class has no virtual functions and no virtual base classes, and
- no non-static data member of its class has a default member initializer, and
- all the direct base classes of its class have trivial default constructors, and
- for all the non-static data members of its class that are of class type (or array thereof), each such class has a trivial default constructor.

Otherwise, the default constructor is non-trivial.

For copy/move constructors, see [this section in the C++ Standard - class.copy.ctor#11](#)¹⁵:

A copy/move constructor for class X is trivial if it is not user-provided and if:

- class X has no virtual functions and no virtual base classes, and
- the constructor selected to copy/move each direct base class subobject is trivial, and
- for each non-static data member of X that is of class type (or array thereof), the constructor selected to copy/move that member is trivial;

otherwise the copy/move constructor is non-trivial.

As I mentioned, `MyType0` and `MyType2` are *trivial* because they have trivial default constructors and don't violate any of the above rules. `MyType1` and `MyType3` have empty constructors, but they are *user-provided*, so they cannot be *trivial* types.

We have some definitions, but what are the implications of those slight differences?

- Trivial types occupy a contiguous memory area (including padding).
- They are “*mem-copyable*”, so it means you can convert them into a byte array and read it back.

¹⁴<https://timsong-cpp.github.io/cppwp/n4868/class.default.ctor#3>

¹⁵<https://timsong-cpp.github.io/cppwp/n4868/class.copy.ctor#11>

- Trivial types cannot be declared `const` without an initializer.
- When a trivial type is [zero-initialized](#)¹⁶ (for example, through value initialization `{}`), its data members will also be zero-initialized¹⁷

Let's try some code:

```
// zero initialization
MyType0 t0{};
std::cout << t0.x << '\n';
MyType1 t1{};
std::cout << t1.x << '\n';
MyType2 t2{};
std::cout << t2.x << '\n';
MyType3 t3{};
std::cout << t3.x << '\n';
```

When you run the code [@Compiler Explorer](#)¹⁸, you get the following output:

```
0
408939456
0
0
```

As you can see, zero initialization kicks in, but not for `MyType1`. In that case, the compiler calls a default constructor but won't initialize the data member to 0.

Similarly, for `const` variables:

```
// const MyType0 ct0; // error!
const MyType1 ct1; // fine, empty ctor called
// const MyType2 ct2; // error!
const MyType3 ct3; // fine
```

¹⁶https://en.cppreference.com/w/cpp/language/zero_initialization

¹⁷According to C++ Reference: The standard specifies that zero-initialization is not performed when the class has a user-provided or deleted default constructor, which implies that whether said default constructor is selected by overload resolution is not considered. All known compilers perform additional zero-initialization if a non-deleted defaulted default constructor is selected.

¹⁸<https://godbolt.org/z/7bKnP6qea>



Additionally, the class type is of *standard layout*, which means briefly that their memory layout is well defined and thus can be consumed by a C program. When a class is also trivial, sharing across multiplatform code or communicating with the C language modules is easy. Read more at [Trivial, standard-layout, POD, and literal types | Microsoft Docs](#)¹⁹.



If you like to read more about trivial types, layout, and more, I highly recommend reading the book “Embracing Modern C++ Safely”, chapter 2, page 401, “Generalized PODs”.

¹⁹<https://docs.microsoft.com/en-us/cpp/cpp/trivial-standard-layout-and-pod-types?view=msvc-170>

4. Delegating and Inheriting Constructors

In this chapter, we'll have a look at improvements from C++11 that relate to inheritance and the ability to call constructors from other constructors.

Delegating constructors

Sometimes, when your class contains many data members and several constructors, it might be convenient to reuse their initialization code. Fortunately, since C++11, you can use **delegating constructors**. Let's look at an example:

Ex 4.1. Delegating constructors. Run [@Compiler Explorer](#)

```
class Product {
public:
    Product(int id, unsigned quantity, const std::string& name)
        : id_{id}, quantity_{quantity}, name_{name}
    {
        verifyData();
    }

    Product(const std::string& name)
        : Product{0, 0, name}
    { }

    void verifyData() {
        if (quantity_ > MaxQuantity)
            throw std::invalid_argument("quantity is too large!");
    }

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

private:
```

```

    int id_;
    unsigned quantity_;
    std::string name_;

    static constexpr unsigned MaxQuantity = 100;
};

```

In the above example, we declare two constructors. The first one performs the core job. The second calls the “primary” one. Inside this main constructor, we not only initialize data members but also call other code. In our case, it’s a form of basic data validation. Please notice that I also used a default parameter (`id = 0`) for the constructor, so that’s another alternative when you want to offer various options for calling it.

And here’s the demo code:

Ex 4.1. Delegating constructors, demo. Run [@Compiler Explorer](#)

```

int main() {
    try {
        Product box{"a box"};
        std::cout << "product: " << box.getName() << " created... \n";

        Product toy{101, 200, "a box"};
        std::cout << "product: " << toy.getName() << " created... \n";
    }
    catch (const std::exception& e) {
        std::cout << "cannot create: " << e.what() << '\n';
    }
}

```

We can run it and get:

```

product: a box created...
cannot create: quantity is too large!

```

Without having delegating constructors, we’d have to duplicate the code:

```

Product(int id, unsigned quantity, const std::string& name)
    : id_{id}, quantity_{quantity}, name_{name}
{
    verifyData();
}

Product(const std::string& name, int id = 0)
    : id_{id}, quantity_{0}, name_{name}
{
    verifyData(); // code duplication
}

```

As you can see, the code with the delegating constructor is much more compact and allows full code reuse. This saves typing and might eliminate various “copy&paste” bugs in your code.

What’s more, the syntax doesn’t limit us to regular constructors only, as you can call a constructor from a copy or move constructor:

```

// copy:
Product(const Product& other) : Product{other.id_, other.quantity_, other.name_}
{ }
// move, potentially a bad idea (just for illustration)
Product(Product&& other) : Product{other.id_, other.quantity_, other.name_}
{ }

```

In a case of a copy constructor, such code might reuse the validation parts. But, **be warned** about the move constructor, as the above code won’t make any “moves” and will simply copy the data, which fails its primary purpose.



Be careful about the syntax!

```
explicit PropertyInfo(double price) { PropertyInfo(...); }
```

The above line will create a local object rather than calling the other constructor!

The call to a constructor has to appear before the constructor body.

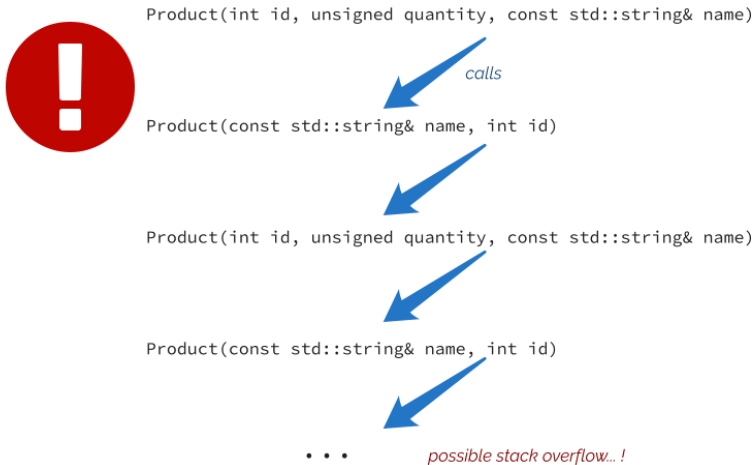
Limitations

Writing too many constructors might lead to some mistakes and recursive calls. Take a look at the following code:

```
class Product {  
public:  
    Product(int id, unsigned quantity, const std::string& name)  
        : Product {name, id}  
    { }  
  
    Product(const std::string& name, int id = 0)  
        : Product{id, 0, name}  
    { }  
  
    // ...  
};  
  
Product recursion{"a single recursion"};
```

What happens when the info object calls its constructor?

You might get a segmentation fault and stack overflow! This is a recursive call, and the compiler cannot detect it until the code is executed at runtime.



Another “restriction” is that you cannot mix member initialization with calling other constructors.

The following code won’t compile:

```
Product(int id, unsigned quantity, const std::string& name)
    : Product {name, id}, quantity_{quantity}
{ }
```

For example, GCC reports the following error:

```
error: mem-initializer for 'Product::quantity_' follows constructor delegation
7 |         : Product {name, id}, quantity_{quantity}
```

To sum up, if you want to use delegating constructors, you cannot initialize other data members.

Let’s go to another section on constructors where you’ll learn one more modern C++ trick.

Inheritance

Let’s look at situations where your class inherits from other classes. What happens with constructors? When does the compiler call them? This discussion will provide a background for a new feature from C++11 called *Inheriting Constructors*.

For debugging, let's introduce a derived class from `DataPacket` called `DebugData`, with special printing capabilities:

```
class DebugDataPacket : public DataPacket {
public:
    DebugDataPacket(const std::string& name, size_t serverId)
        : DataPacket{name, serverId}
    { }

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

As you can see, the code declares a new class and uses `: public DataPacket` to indicate public inheritance. The example also defines a single constructor that invokes base class constructors.

C++ offers three options to specify the way we inherit from a base class:

- Public inheritance means that `public` members of the base class become `public` members of the derived class, and `protected` members are `protected` in the derived type.
- Protected inheritance makes all `public` and `protected` members of the base class accessible as `protected` members of the derived class.
- Private inheritance makes all `public` and `protected` base class members accessible as `private` members of the derived class.
- In all three cases, `private` members of the base class are not accessible by derived classes unless explicitly made `friend`.

We can use it like:

Ex 4.2. Inheritance, simple demo code. Run [@Compiler Explorer](#)

```
int main() {
    DebugDataPacket hello{"hello!", 404};
    hello.DebugPrint(std::cout);
}
```

In the example, base class constructors are called explicitly, but in general, each constructor will also call the default constructor of a base class implicitly. This is illustrated by the following code:

Ex 4.3. Base class construction order. Run [@Compiler Explorer](#)

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

class Product {
public:
    Product() : id_{0} { std::cout << "Product() default\n"; }
    explicit Product(int id, const std::string& name)
        : id_{id}, name_{name}
    {
        std::cout << "Product(): " << id_ << ", " << name_ << '\n';
    }

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

class ExProduct : public Product {
public:
    ExProduct() { std::cout << "ExProduct() default\n"; }
    explicit ExProduct(int id) {
        id_ = id;
        std::cout << "ExProduct(id)\n";
    }
};

int main() {
```

```

    ExProduct p;
    ExProduct withId{42};
}

```

If we run the program, we'll get:

```

Product() default
ExProduct() default
Product() default
ExProduct(id)

```

As you can see, even though we haven't called any base constructor inside our `ExProduct` constructor, the compiler invoked it anyway. What's more, inside a constructor of a derived class, you cannot use base classes' data members in the initialization list, for example:

```

ExProduct(): id_(10) { // << err! we don't have access!
    std::cout << "ExProduct() default\n";
}

```

You can only access it in the body of the constructor:

```

ExProduct(int id) {
    id_ = id;
    std::cout << "ExProduct(id)\n";
}

```

This behavior is essential to keep the integrity of the object.



Additionally, it's best not to call `virtual` functions in constructors as they might behave differently than expected. In short, a call to a virtual function in a base class constructor results in the call to the base implementation, as the inherited class and the implementation is not yet set up. You can read more about this behaviour in [the C++ FAQ¹](https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors) or at [C++ Core Guideline C.82²](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor-virtual).

After the introduction to the inheritance topic, we can discuss one improvement we got with Modern C++.

¹<https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors>

²<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor-virtual>

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 which 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](#)

```
1  class DebugDataPacket : public DataPacket {
2  public:
3      using DataPacket::DataPacket;
4
5      void DebugPrint(std::ostream& os) {
6          os << getData() << ", " << getChecksum() << '\n';
7      }
8  };
9
10 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. See the example:

Ex 4.5. Inheriting constructors and protected section. Run [@Compiler Explorer](#)

```

struct Base {
    int x{};
    int y{};

    Base(int a, int b): x{a}, y{b} { }
protected:
    Base() = default;
    Base(int a): x{a} { }
};

struct Derived : public Base {
    using Base::Base;
};

int main() {
    // Derived d{0}; // error: 'Base::Base(int)' is protected
    Derived d2{0, 1}; // fine
}

```

If you want to limit the access to constructors, you must explicitly write constructors for Derived:

```
Derived(int a) : Base{a} { }
```

```
Derived d{0}; // fine now, as Derived::Derived(int) is public
```

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...
```

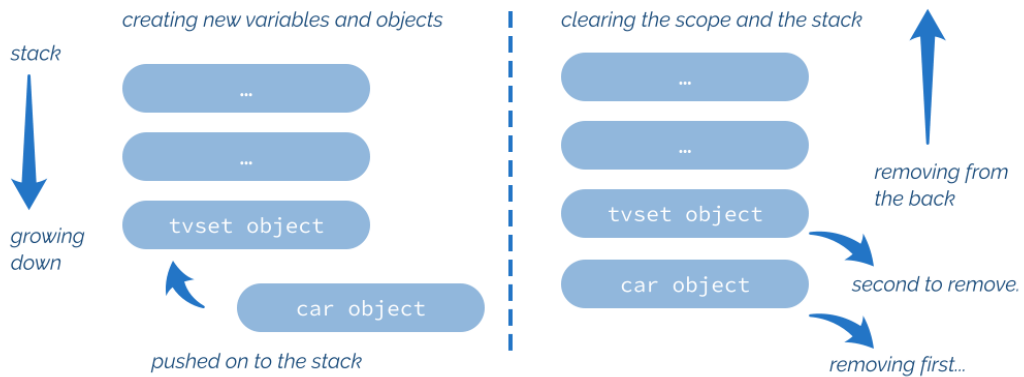
I specifically enclosed objects (created on the stack) in separate scopes so that their lifetime ends when their scope ends. On the other hand, if we have code:

```
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

In the previous examples, I used objects created on the stack. For a clearer picture of destructors, it's good to discuss a case when you have objects on the heap. In that case, a destructor will be called only when the memory is released via the `delete` operator.

Consider the following snippet:

```
{
    Product* ptr = new Product("TV Set", 123);
}
// !!
```

`ptr` is a pointer to an object allocated on the heap. But I didn't call `delete`, and thus, the destructor won't be invoked! Moreover, I generated a memory leak since the memory was also not released. After the scope ends, `ptr` goes out of the scope, but since it's a pointer, the memory is still present but not accessible.

To fix the issue we have to call `delete` (for single elements) or `delete []` (for arrays).

```
{
    Product* ptr = new Product("TV Set", 123);
    // use ptr...
    delete ptr;
}
```

And similarly, for an array:

```
{
    Product* arr = new Product[10]("TV Set", 123);
    // use...
    delete [] arr;
}
```

Since it's easy to forget about proper heap release, it's best to use smart pointers that wrap allocation with the ownership.

```
{
    std::unique_ptr<Product> ptr = std::make_unique<Product>("box", 1);
    // use ptr...
}
```

Now, when `ptr` goes out of the scope, it's "smart" and knows it also has to call `delete` on the stored pointer. In the example, I'm using a `unique_ptr` as the basic form of smart pointers in the C++ Standard Library. It wraps the pointer to an object and keeps the "unique" ownership of it. If you want to pass pointers around in the system and have multiple "owners," then you can use `shared_ptr`.

The smart pointer will also work for the array version:

```
{
    // create 10 Products
    std::unique_ptr<Product[]> ptr = std::make_unique<Product[]>(10);
    // use...
}
```

And this time, the `unique_ptr` makes sure the `delete []` is called. You can play with the example [@Compiler Explorer](#)³.

For more information about smart pointers, have a look at my blog series: [6 Ways to Refactor new/delete into unique ptr - C++ Stories](#)⁴ and more articles about [smart pointers @C++Stories](#)⁵.



What's more, Modern C++ strongly suggest avoiding raw `new` and `delete`. Thanks to many library containers, wrappers and smart pointers there's almost no need to rely on those low-level memory management routines. See this C++ Core guideline: [R.11: Avoid calling new and delete explicitly](#)⁶. The code in this section with `new` can be treated only for illustrative purpose.

Destructors and data members

If you have a class, then by default its destructor calls the destructors for all data members:

Ex 5.2. Nested destructor call. Run [@Compiler Explorer](#)

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

class Product {
    // defined as in the previous example...
};

class Wrapper {
public:
```

³<https://godbolt.org/z/Ps9Ye79zc>

⁴<https://www.cppstories.com/2021/refactor-into-uniqueptr/>

⁵<https://www.cppstories.com/tags/smart-pointers/>

⁶<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#r11-avoid-calling-new-and-delete-explicitly>


```

    Wrapper() : prod_("internal", 123) {
        std::cout << "Wrapper()\n";
    }
    ~Wrapper() {
        std::cout << "~Wrapper()\n";
    }
private:
    Product prod_;
};

int main() {
    Wrapper w;
}

```

In example 5.2, the Wrapper class contains Product as a data member.

The output:

```

internal, id 123
Wrapper()
~Wrapper()
internal destructor...

```

As you notice, `internal destructor` is called along with the `~Wrapper()` invocation.

On the other hand, when your object is part of some other class as a pointer, then the pointer will go out of the scope, but not the data that it points to. So you have to pay attention to your pointer data members and call their `delete` in a proper place. See an example below:

Ex 5.3. Pointer data member and a destructor. Run [@Compiler Explorer](#)

```

class Wrapper {
public:
    Wrapper() : prod_(new Product("internal", 123)) {
        std::cout << "Wrapper()\n";
    }
    ~Wrapper() {
        delete prod_;
        std::cout << "~Wrapper()\n";
    }
}

```

```
private:
    Product *prod_;
};

int main() {
    Wrapper w;
}
```

This time I had to manually call `delete prod_` to release the data member properly. It's another case where a smart pointer is handy as it will automatically destroy the underlying object.

Virtual destructors and polymorphism

There's also one feature of destructors that plays an essential part in inheritance and polymorphism.

According to [Wikipedia](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))⁷:

In programming language theory and type theory, polymorphism is the provision of a single interface to entities of different types or the use of a single symbol to represent multiple different types.

In short, in C++, the definition means that if you have a pointer or a reference to a base class, when you call a member function, the compiler invokes an implementation (if available) in the derived classes. C++ does this technique through `virtual` functions. We can demonstrate it with the following, but naïve, code in C++:

⁷[https://en.wikipedia.org/wiki/Polymorphism_\(computer_science\)](https://en.wikipedia.org/wiki/Polymorphism_(computer_science))

Ex 5.4. Virtual destructor, base class, incorrect. Run [@Compiler Explorer](#)

```
class Product {
public:
    explicit Product(const char* name)
        : name_(name)
    {
        std::cout << name << '\n';
    }

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

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

    virtual double CalculateMass() const = 0;

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

The above `Product` type declares a virtual member function. We can declare derived classes and then provide their implementation of that virtual member function. This allows the compiler to call the proper function based on the type and give this “polymorphic” semantics.

Have a look:

Ex 5.4. Virtual destructor, derived classes. Run [@Compiler Explorer](#)

```
struct BoxProduct : public Product {
    using Product::Product; // inheriting ctor
    ~BoxProduct() {
        std::cout << "~BoxProduct...\n";
    }

    double CalculateMass() const override { return 10.0; }
};
```

```

struct FluidProduct : public Product {
    using Product::Product; // inheriting ctor
    ~FluidProduct() {
        std::cout << "~FluidProduct...\n";
    }

    double CalculateMass() const override { return 100.0; }
};

```

The CalculateMass function has two separate, and trivial, implementations⁸. The function signature also uses the override keyword, which is a C++11 addition. It tells the compiler that a given member function is about to be overridden, so the compiler can check if there's a corresponding declaration in a base class. Read more about the keyword in my article: [Modern C++: Safety and Expressiveness with override and final - C++ Stories](#)⁹.

We can now write code that uses both of those products:

Ex 5.4. Virtual destructor, main. Run [@Compiler Explorer](#)

```

void CallCalculate(const Product& prod) {
    std::cout << "calculating: " << prod.CalculateMass() << '\n';
}

int main() {
    using std::unique_ptr;
    using std::make_unique;
    unique_ptr<Product> box = make_unique<BoxProduct>("box");
    unique_ptr<Product> water = make_unique<FluidProduct>("water");
    CallCalculate(*box.get());
    CallCalculate(*water.get());
}

```

The demo use case is simple, it creates two smart pointers that are pointers to a base class, but they are assigned with pointers to derived classes. When we run the code [@Compiler Explorer](#)¹⁰, you'll see the following output:

⁸Let's assume that in the actual production code, those functions would have some more advanced calculations based on the properties of a particular type.

⁹<https://www.cppstories.com/2021/override-final/>

¹⁰<https://godbolt.org/z/cb9Pfhhqe>

```
box
water
calculating: 10
calculating: 100
water destructor...
box destructor...
```

As you can see, the `CallCalculate(Product& prod)` function takes a reference to a base class, and then it can call its functions. If a function is virtual, the compiler will call in polymorphically, based on the real type.

But... do you see an error here?

Take a moment and think...

It looks like the destructor of the derived class is not called!

This is because we used pointers to hold our objects. And when the smart pointer goes out of the scope, it will call `delete` on the pointer to a base class. Since the destructor is not marked as `virtual`, the polymorphism doesn't kick in, and only the `~Product()` destructor is called.

To fix this, each class that has virtual functions should also have a virtual destructor:

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

This fixes our output:

```
box
water
calculating: 10
calculating: 100
~FluidProduct...
water destructor...
~BoxProduct...
box destructor...
```

You can play with the correct example [@Compiler Explorer](https://godbolt.org/z/dfa49sjc6)¹¹.

¹¹<https://godbolt.org/z/dfa49sjc6>

There's also a specific C++ Core Guideline related to this critical aspect. See [C.35: A base class destructor should be either public and virtual, or protected and non-virtual¹²](#):

Reason: To prevent undefined behavior. If the destructor is public, then calling code can attempt to destroy a derived class object through a base class pointer, and the result is undefined if the base class's destructor is non-virtual.

Partially created objects

The compiler calls a destructor only for objects that are fully created. Consider the modified version of a constructor that checks the `id` parameter and throws an exception:

```
explicit Product(const char* name, unsigned id)
: name_(name)
, id_(id)
{
    std::cout << name << ", id " << id << '\n';
    if (id < 100)
        throw std::runtime_error{"bad id..."};
}
```

Ex 5.5. Destructors and partial object creation. Run [@Compiler Explorer](#)

```
int main() {
    try {
        Product tvset("TV Set", 123);
        Product car("Mustang", 9);
    }
    catch (const std::exception& ex) {
        std::cout << "exception: " << ex.what() << '\n';
    }
}
```

When we run the example, we'll get the output:

¹²<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c35-a-base-class-destructor-should-be-either-public-and-virtual-or-protected-and-non-virtual>

```
TV Set, id 123
Mustang, id 9
TV Set destructor...
exception: bad id...
```

This time the example creates two objects: `TV set` and `Mustang`. In the output, we can notice that both objects call their constructors, but there's only one destructor invocation (for `TV set`). Since `Mustang` threw an exception in the constructor, the destructor won't be executed.



Since destructors might be called when the compiler performs stack unwinding, they shouldn't throw exceptions, as this might result in calling `std::terminate()`. Read this C++ Core Guideline suggestion for more information: [E.16: Destructors, deallocation, and swap must never fail](#)¹³.

Another important aspect is to manage resources allocated before the exception occurs properly. For example, if you allocate some memory dynamically using a raw pointer, you might get a memory leak. See the following sample:

Ex 5.6. A memory leak in partially created objects. Run [@Compiler Explorer](#)

```
class Product {
public:
    explicit Product(int id)
        : res_(new Resource())
    {
        std::cout << "Product: id " << id << '\n';
        if (id < MIN_ID)
            throw std::runtime_error{"bad id..."};
    }

    ~Product() {
        delete res_;
        std::cout << "~Product...\n";
    }

private:
    Resource* res_;
};
```

¹³<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-never-fail>

```

int main() {
    try {
        Product invalid(MIN_ID - 1);
    }
    catch (const std::exception& ex) {
        std::cout << "exception: " << ex.what() << '\n';
    }
}

```

The output:

```

Product: id 99
exception: bad id...

```

As you can see, the compiler didn't call the destructor for `Resource`, and the memory wasn't released (since we called `res_(new Resource())` in the constructor). We could potentially fix this leak by deleting `res_` before we throw. Still, manual resource management is fragile, and it's best to look for a better solution.

The key mechanism to fix such leaking resources is to rely on variables and data members with automatic storage duration, like regular value types. In that case, the stack unwinding destroys them properly and calls their destructors. That's why a destructor for smart pointers can be safely called:

Ex 5.7. Fixing memory leaks in partially created objects. Run [@Compiler Explorer](#)

```

class Product {
public:
    explicit Product(int id)
        : res_(std::make_unique<Resource>())
    {
        std::cout << "Product: id " << id << '\n';
        if (id < MIN_ID)
            throw std::runtime_error{"bad id..."};
    }

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

```



```
    }

private:
    std::unique_ptr<Resource> res_; // << smart pointer now!
};

int main() {
    try {
        Product invalid(MIN_ID - 1);
    }
    catch (const std::exception& ex) {
        std::cout << "exception: " << ex.what() << '\n';
    }
}
```

When we run the code, we'll see the following output:

```
Product: id 99
~Resource
exception: bad id...
```

The compiler didn't call the destructor for `Product`, but the stack unwinding correctly called the destructor for all data members with automatic storage duration.



You can read more information about stack unwinding and handling resources on the following sites: [throw expression - @C++Reference](#)¹⁴ and [Exceptions and Error Handling, C++ FAQ](#)¹⁵.

A compiler-generated destructor

As with other special member functions, the compiler creates an implicit default destructor for your classes if you don't provide your implementation. The basic rule is that each data member, and also the base classes must have an accessible destructors (they are not deleted, not inaccessible nor ambiguous). For example:

¹⁴https://en.cppreference.com/w/cpp/language/throw#Stack_unwinding

¹⁵<https://isocpp.org/wiki/faq/exceptions#selfcleaning-members>

Ex 5.8. Compiler generated destructor. Run [@C++Insights](#)

```
#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';
    }

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

int main() {
    Product first{10, "basic"};
}
```

At C++Insights, we can see the output from the compiler and how it “sees” the code. As you can notice, the compiler created the following destructor for us:

```
inline ~Product() noexcept = default;
```

You can read more about compiler-generated destructors in the “Implicitly-declared destructor” section [@C++Reference](#)¹⁶.

Summary and use cases

This chapter covered destructors, a special member function invoked when an object ends its lifetime. In most cases, we can use this capability to properly clean up the object and deallocate any resources that we used and have not yet released.

¹⁶<https://en.cppreference.com/w/cpp/language/destructor>

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.

Destructors are one of the best features of the C++ language as they provide a clear and well-defined point at where they are called. This is opposed to dynamic garbage collection that can work in the background, potentially slowing down the program and with less control over the process. Destructors are also the critical element to a popular term in C++ called RAII (Resource Acquisition Is Initialization), coined by B. Stroustrup, the author of the C++ language. It states that holding a resource is a class invariant, and is tied to object lifetime. Read more at [Wikipedia](https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization)¹⁷.

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.

¹⁷https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

6. Initialization and Type Deduction

Since C++11, we can write shorter code thanks to automatic type deduction. Rather than specifying the full type for a new object, we can ask the compiler to deduce its type. In this chapter, we'll discuss how that "type deduction" affects initialization.

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

7. Quiz on Special Member Functions

Congratulations!

You've just completed the section on the basics and special member functions. Here's a quick quiz. Try answering the following questions, and then we will continue our journey :)

1. In a class that doesn't inherit from other types, can you declare a constructor using 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`.

3. Can a constructor return a value using the `return` statement?

1. Yes, it's like a regular function with a return type of a class name.
2. No, a constructor doesn't have any return type specified.
3. Yes, though a special data member called `self_return`.

4. Can you mix delegating constructors with data member initialization, like in the constructor `Type(int a, int b)`?

For example:

```
struct Type {  
    explicit Type(int a) : a_(a) { }  
    explicit Type(int a, int b) : Type(a), b_(b) { }  
    int a_;  
    int b_;  
};
```

1. Yes.
2. Sometimes, depending on if the data members come first.
3. No, the compiler reports an error in this case.

5. Is the following code ok?

```
Product* arr = new Product[10];  
// use...  
delete arr;
```

Select the true statement:

1. Yes. The code is fine and properly destroys `arr`.
2. This code generates a memory leak as not all elements from `arr` are deleted. The code should use `delete [] arr;`
3. The code uses `delete arr`, which is not necessary as the compiler will properly release all Products.

6. Select the true statements

1. Copy initialization considers `explicit` constructors and will use them if there's a matching one.
2. When you pass an argument to a function by value, then a copy initialization is used to initialize the function parameter.
3. Aggregate initialization copy-initializes each sub-objects or an array element for which an initializer is provided.

Please write down your answers and check them in Appendix B.

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 [@Compiler Explorer](#)

```
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 data members have their default values set at the point of declaration. There’s no need to assign default values inside constructors. Such a feature is 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;  
};
```

Here’s the full working example:

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 and see the changes in the output.

¹Technically, 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 plug in our function and write some additional constructors:

```

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

    SimpleType() { std::cout << "SimpleType()\n"; }
    SimpleType(int x) : a(x) { std::cout << "SimpleType(int)\n"; }
};

```

Here's the test code scenario:

Ex 8.3. Calling `initA` and `initB` functions. Run @Compiler Explorer

```

#include <iostream>
#include <string>

int initA() { /* as above */ }
std::string initB() { /* as above */ }
struct SimpleType { /* as in the snippet above */ };

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 t0
initA() called
initB() called
SimpleType()
SimpleType t1(10)
initB() called
SimpleType(int)

```

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. Later, the body of the default constructor is called.

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) { }  
};
```

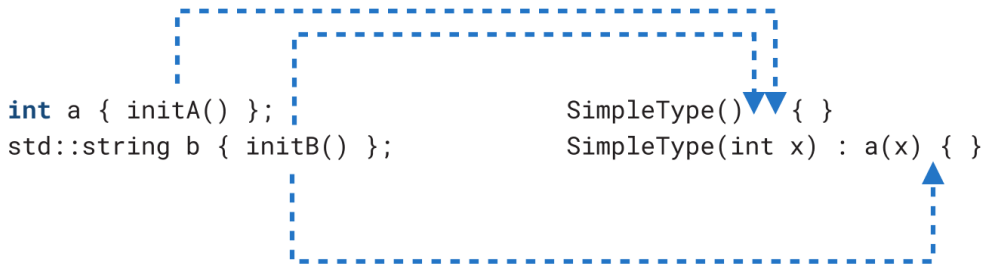
Into:

```
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:

data members

constructors



Other forms of NSDMI

Let's try some other examples and see all options that we can initialize a data member using NSDMI:

Ex 8.4. Various syntax for NSDMI. Run [@Compiler Explorer](#)

```

1  struct S {
2      int zero {};           // fine, value initialization
3      int a = 10;           // fine, copy initialization
4      double b { 10.5 };    // fine, direct list initialization
5      // short c ( 100 );    // err, direct initialization with parens
6      int d { zero + a };    // dependency, risky, but fine
7      // double e { *mem * 2.0 }; // undefined!
8      int* mem = new int(d);
9      long arr[4] = { 0, 1, 2, 3 };
10     std::array<int, 4> moreNumbers { 10, 20, 30, 40 };
11     // long arr2[] = { 1, 2 }; // cannot deduce
12     // auto f = 1;           // err, type deduction doesn't work
13     double g { compute() };
14
15     ~S() { delete mem; }
16     double compute() { return a*b; }
17 };

```

Here's the summary:

- zero uses *value* initialization, and thus, it will get the value of 0,
- a uses *copy* initialization,
- b uses direct list initialization,
- c would generate an error as *direct* initialization with parens is not allowed for NSDMI,
- d initializes by reading zero and a, but since d appears later in the list of data members, it's okay, and the order is well defined,
- e, on the other hand, would have to read from a data member mem, which might not be initialized yet (since it's further in the declaration order), and thus this behavior is undefined,
- mem uses a memory allocation which is also acceptable,
- arr[4] declares and initializes an array, but you need to provide the number of elements as the compiler cannot deduce it (as in arr2),
- similarly we can use `std::array<type, count>` for moreNumbers but we need to provide count and the type of the array elements,
- f would also generate an error, as auto type deduction won't work,
- g calls a member function to compute the value. The code is valid only when that function calls reads from already initialized data members.

And here's a simple "demo" to test the S structure:

Ex 8.4. Various syntax for NSDM - Demo. Run [@Compiler Explorer](#)

```

void showProperties(std::string_view text, const S& s) {
    std::cout << text << '\n';
    std::cout << ".zero: " << s.zero << '\n';
    std::cout << ".a:    " << s.a << '\n';
    std::cout << ".b:    " << s.b << '\n';
    std::cout << ".d:    " << s.d << '\n';
    std::cout << "/*.mem: " << *s.mem << '\n';
    std::cout << ".arr[0]: " << s.arr[0] << '\n';
    std::cout << "g:      " << s.g << '\n';
}

int main() {
    S s;    // default initialization

```

```
    showProperties("s", s);

    S y { 1 }; // aggregate initialization
    showProperties("y", y);
}
```

The first object `s` uses default initialization, and it will assign default values to all data members. For the second object `y`, I used aggregate initialization with only the first argument, so it will only set the `S::zero` data member.

When we run the code we can see the following output:

```
s
.zero: 0
.a:    10
.b:    10.5
.d:    10
*.mem: 10
.arr[0]: 0
g:     105
y
.zero: 1
.a:    10
.b:    10.5
.d:    11
*.mem: 11
.arr[0]: 0
g:     105
```

Using the knowledge from this section, in our `DataPacket` class we can be more “creative” and write the following initializers. Be warned that this is probably not the best code and we’ll discuss this later.

Ex 8.5. Dependency in initializers, potentially risky. Run [@Compiler Explorer](#)

```
class DataPacket {
    std::string data_ {"empty"};
    size_t checksum_ { calcChecksum(data_) };
    size_t serverId_ { 404 };

    /* rest of the code*/
}
```

Since `checksum_` is after `data_`, we know the order of initialization, and thus we can safely use `data_` and pass it into `calcChecksum`.

While the code works and the order of initialization is well defined in our case, such a technique might be problematic to maintain. If you introduce a new data member and reorder class parts, you might encounter some new bugs and complications. For some people, such an approach might also be harder to read and understand. I mentioned a similar problematic case with a regular initializer list in constructors.

That's why it's best to avoid such dependency and write:

```
inline constexpr auto defaultData {"empty"};
class DataPacket {
    std::string data_ {defaultData};
    size_t checksum_ { calcChecksum(defaultData) };
}
```

Now, it's clear what's the default value, and there's no dependency in the initialization sequence. Here's the corrected version [@Compiler Explorer²](#). And we'll look at `inline` variables in a [separate chapter](#).

Copy constructor and NSDMI

The compiler initializes the fields in all the constructors, including the copy and move constructors. However, when a copy or move constructor is the default, there's no need to perform that extra initialization.

Now, let's update our previous examples with copy constructors:

²<https://godbolt.org/z/9vezbWfbs>

Ex 8.6. Copy constructor and NSDMI. Run [@Compiler Explorer](#)

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

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

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

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

    SimpleType() { }
    explicit SimpleType(std::string s) : b(std::move(s)) { }

    SimpleType(const SimpleType& other) {
        std::cout << "copy ctor\n";

        a = other.a;
        b = other.b;
    };
};

int main() {
    SimpleType t1;
    std::cout << "SimpleType t2 = t1:\n";
    SimpleType t2 = t1;
}
```

After running it, we can see the following output:


```
initA() called
initB() called
SimpleType t2 = t1:
initA() called
initB() called
copy ctor
```

The compiler initialized the fields with their default values in the above example. We can see that `initA()` and `initB()` are called just before the `copy ctor` message.

This is why it's better to use the initializer list inside a copy constructor:

```
SimpleType(const SimpleType& other) : a(other.a), b(other.b) {
    std::cout << "copy ctor\n";
};
```

Now we'll get the following output:

```
SimpleType t1:
initA() called
initB() called
SimpleType t2 = t1:
copy ctor
```

The same happens if you rely on the default copy constructor generated by the compiler (of course, this time, you won't get the output).

```
SimpleType(const SimpleType& other) = default;
```

See the live code [@Compiler Explorer](https://godbolt.org/z/jM8863Wo3)³.

Move constructor and NSDMI

We can observe a similar effect with a move constructor:

³<https://godbolt.org/z/jM8863Wo3>

Ex 8.7. NSDMI and move constructor. Run [@Compiler Explorer](#)

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

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

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

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

    SimpleType() { }
    explicit SimpleType(std::string s) : b(std::move(s)) { }

    SimpleType(const SimpleType& other) = default;

    SimpleType(SimpleType&& other) // only for illustration
    {
        std::cout << "move ctor\n";
        a = std::move(other.a);
        b = std::move(other.b);
    };
};

int main() {
    std::cout << "SimpleType t1:\n";
    SimpleType t1;
    std::cout << "SimpleType t2 = t1:\n";
    SimpleType t2 = std::move(t1);
}
```

When you run the code, you can see that `initA()` and `initB()` are also called only at the start of the move constructor:

```
SimpleType t1:
initA() called
initB() called
SimpleType t2 = t1:
initA() called
initB() called
move ctor
```

This can be fixed by writing a default move constructor:

```
SimpleType(SimpleType&&) = default;
```

or:

```
SimpleType(SimpleType&& other) noexcept
    : a(std::move(other.a)), b(std::move(other.b)) { }
```

You can now experiment with the code example above and see if changing the move constructor reduces the number of invocations of `initA()` and `initB()`.

C++14 changes

Originally, in C++11, if you used default member initialization, your class would lose the “aggregate” status:

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

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

Fortunately, in C++14 the limitation was lifted and the above line compiles. The aggregate status of the `Point` struct is preserved. You can see and play with the full code below:

Ex 8.8. 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

Since C++11, the code only considered “regular” fields... but how about bit fields in a class? For example:

```
class Type {
    int value : 4;
};
```

Unfortunately, in C++11, it wasn’t possible to default-initialize the value bit field. However, with a compiler that conforms to C++20, you can write:

Ex 8.9. Bit fields and NSDMI in C++20. Run [@Compiler Explorer](#)

```
#include <iostream>

struct Type {
    int value : 4 = 1;
    int second : 4 { 2 };
};

int main() {
    Type t;
    std::cout << t.value << '\n';
    std::cout << t.second << '\n';
}
```

As you can see above, C++20 offers improved syntax where you can specify the default value after the bit size: `var : bit_count { default_value }.`

Limitations of NSDMI

In this section, we'll discuss the current (as of C++20) limitations of non-static data member initialization.

The case with `auto` type deduction

Since we can declare and initialize a variable inside a class, can we also/still use `auto`? It seems natural and follows the AAA (Almost Always Auto) Rule.



Almost Always Auto Rule: this term was coined by Herb Sutter. It recommends using `auto`-type deduction rather than writing explicit types. See the blog post [GotW #94 Solution: AAA Style \(Almost Always Auto\)](https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/)⁴

You can use `auto` for static variables:

```
class Type {  
    static inline auto theMeaningOfLife = 42; // int deduced  
};
```

However, you cannot use it as a class non-static member:

```
class Type {  
    auto myField { 0 }; // error  
    auto param { 10.5f }; // error  
};
```

The alternative syntax also fails:

⁴<https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/>

```
class Type {  
    auto myField = int { 10 };  
};
```

Unfortunately, auto is not supported. For example, in GCC, I get:

```
error: non-static data member declared with placeholder 'auto'
```

It's easy for the compiler to deduce the type of a static data member as the initialization happens at the place you declare it. However, it's not possible for regular data members because the initializer might come from the default member init or the constructor (when you override a default value).

```
class Type {  
    auto myField = int { 10 };  
};
```

cannot deduce!



The case with Class Template Argument Deduction (CTAD)

As with auto, non-static member variables and Class Template Argument Deduction (CTAD) also have limitations.

CTAD has been available since C++17, and it allows you to define a class template object without specifying the template arguments. For example:

```
std::pair<double, int> myPair(10.5, 42);  
std::vector<float> numbers { 1.1f, 2.2f, 3.3f };  
std::array<double, 3> doubles { 1.1, 2.2, 3.3 };
```

In C++17, we can write:

```
std::pair myPair(10.5, 42);
std::vector numbers { 1.1f, 2.2f, 3.3f };
std::array doubles { 1.1, 2.2, 3.3 };
```

The compiler deduces the correct template arguments for `std::pair`, `std::vector` and `std::array`.

This new functionality works fine for static data members of a class:

```
class Type {
    static inline std::vector ints { 1, 2, 3, 4, 5 }; // deduced vector<int>
};
```

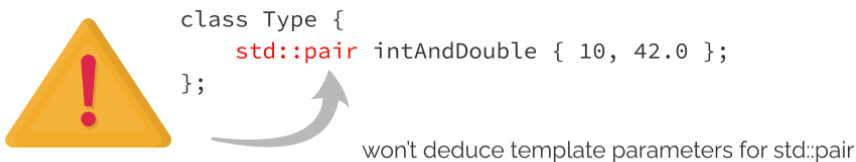
However, it does not work as a non-static member:

```
class Type {
    std::vector ints { 1, 2, 3, 4, 5 }; // error!
};
```

On GCC 10.0, I get:

error: 'vector' does not name a type

Hopefully, both issues presented here are not big blockers, but it's good to be aware of them.



The case with direct initialization and parens⁵

I applied NSDMI to the `DataPacket` class and initialized `data_` to `{"empty"}`.

⁵Thanks to Nicolai Josuttis for a discussion and clarification on this topic.

```
class DataPacket {
    std::string data_ {"empty"};
    // .. the rest...
```

What if I want `data_` to be initialized with 40 stars `*`? I can write the long string or use one of the `std::string` constructors taking a count and a character. Yet, because of a constructor with the `std::initializer_list` in `std::string` which takes precedence, you need to use direct initialization with parens to call the correct version::

Ex 8.10. Direct initialization with parens and `std::string`. Run [@Compiler Explorer](#)

```
#include <iostream>

int main() {
    std::string stars(40, '*');    // parens
    std::string moreStars{40, '*'}; // <<
    std::cout << stars << '\n';
    std::cout << moreStars << '\n';
}
```

If you run the code, you'll see:

```
*****
(*)
```

It's because `{40, '*'}` converts 40 into a character ((using its) ASCII code), and passes those two characters through `std::initializer_list` to create a string with two characters only. The problem is that direct initialization with parens won't work inside a class member declaration:

```
class DataPacket {
    std::string data_ (40, '*'); // syntax error!
    size_t checkSum_ { calcCheckSum(data_) };
    size_t serverId_ { 404 };

    /* rest of the code*/
```

The code doesn't compile and to fix this you can rely on copy initialization:


```
class DataPacket {  
    std::string data_ = std::string(40, '*'); // fine  
    size_t checksum_ { calcChecksum(data_) };  
    size_t serverId_ { 404 };  
  
    /* rest of the code*/  
};
```

This limitation might be related to the fact that the syntax parens might quickly run into the most vexing parse/parsing issues, which might be even worse for class members.

There's a [separate section on `std::initializer_list`](#) in the book that shares more information about the pros and cons of this helper library type.

How about other constructor types? We'll cover those in the next section.

NSDMI: Advantages and Disadvantages

Let's summarize non-static data member initialization.

Advantages of NSDMI

It looks like using NSDMI is a clear winner for Modern C++. Here are the main reasons why it is so helpful:

- It's easy to write.
- You can be sure that each member is initialized correctly.
- The declaration and the default value are in the same place, so it's easier to maintain.
- It's much easier to conform to the rule that every variable should be initialized.
- It is beneficial when we have several constructors. Previously, we would have to duplicate the initialization code for members or write a custom method, like `InitMembers()`, that would be called in the constructors. Now, you can do a default initialization, and the constructors will only do their specific jobs.

Any negative sides of NSDMI?

On the other hand, the feature has some limitations and inconveniences:

- Using NSDMI makes a class not trivial, as the default constructor (compiler-generated) has to perform some work to initialize data members.
- Performance: When you have performance-critical data structures (for example, a `Vector3D` class), you may want to have an “empty” initialization code. You risk having uninitialized data members, but you might save several CPU instructions.
- (Only until C++14) NSDMI makes a class non-aggregate in C++11. See the section about C++14 changes.
- They have limitations in a case of `auto` type deduction and CTAD, so you need to explicitly provide the type of the data member.
- You cannot use direct initialization with parens, to fix it you need list initialization or copy initialization syntax for data members.
- Since the default values are in a header file, any change can require recompiling dependent compilation units. This is not the case if the values are set only in an implementation file.
- Might be hard to read if you rely on calling member functions or depend on other data members.

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. Such an approach makes it harder to leave data members in an uninitialized state. In many cases it also reduces the need to write any user defined constructors that would only set default values.

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.48⁶](#):

⁶<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c48-prefer-in-class-initializers-to-member-initializers-in-constructors-for-constant-initializers>

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](#)⁷:

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

You have all the knowledge to complete this first simple exercise.

Below, you have the `Point` class declaration with two data members.

⁷<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c45-dont-define-a-default-constructor-that-only-initializes-data-members-use-in-class-member-initializers-instead>

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

Update this class so that it uses NSDMI and initializes `Point::x` to 1.0 and `Point::y` to 2.0.

Here's the code for test cases:

```
TEST(PointTest, X) {  
    Point p;  
    EXPECT_DOUBLE_EQ(1.0, p.x);  
}
```

```
TEST(PointTest, Y) {  
    Point p;  
    EXPECT_DOUBLE_EQ(2.0, p.y);  
}
```

You can practice with the following Compiler Explorer solution: [Point tests @Compiler Explorer](https://godbolt.org/z/Gor1Y8qx7)⁸.

When you run the code, you'll see that the test fail:

⁸<https://godbolt.org/z/Gor1Y8qx7>

```

1 #include <gtest/gtest.h>
2
3 struct Point {
4     double x;
5     double y;
6 };
7
8 TEST(PointTest, X) { ...
9     Point p;
10    EXPECT_DOUBLE_EQ(1.0, p.x);
11 }
12
13 TEST(PointTest, Y) { ...
14     Point p;
15    EXPECT_DOUBLE_EQ(2.0, p.y);
16 }
17
18 int main(int argc, char **argv) {
19     ::testing::InitGoogleTest(&argc, argv);
20     return RUN_ALL_TESTS();
21 }

```

```

Output (0/0) x86-64 gcc (trunk) - 1339ms (1071720B) ~81357 lines filtered
Output of x86-64 gcc (trunk) (Compiler #1)
A Filter... Libraries (1) + Add new... + Add tool...
Wrap lines
2.0
Which is: 2
p.y
Which is: 8.8502137240549163e-317
[ FAILED ] PointTest.Y (0 ms)
[-----] 2 tests from PointTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (0 ms total)
[ PASSED ] 0 tests.
[ FAILED ] 2 tests, listed below:
[ FAILED ] PointTest.X
[ FAILED ] PointTest.Y

2 FAILED TESTS

```

Your task is to improve the code so that tests pass:

```

/
8 TEST(PointTest, X) {
9     Point p;
10    EXPECT_DOUBLE_EQ(1.0, p.x);
11 }
12
13 TEST(PointTest, Y) {
14     Point p;
15    EXPECT_DOUBLE_EQ(2.0, p.y);
16 }
17
18 int main(int argc, char **argv) {
19     ::testing::InitGoogleTest(&argc, argv);
20     return RUN_ALL_TESTS();
21 }

```

```

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from PointTest
[ RUN ] PointTest.X
[ OK ] PointTest.X (0 ms)
[ RUN ] PointTest.Y
[ OK ] PointTest.Y (0 ms)
[-----] 2 tests from PointTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (0 ms total)
[ PASSED ] 2 tests.

```

The second exercise

Let's try another use case. Below, there's a structure called `SalesRecord`.

```
#include <string>

constexpr unsigned int DEFAULT_CATEGORY = 4;
constexpr unsigned int DEFAULT_FLAGS = 0x0a;

struct SalesRecord {
    std::string name_;
    double price_;
    unsigned int category_ : 4;
    unsigned int flags_ : 4;
};
```

Use NSDMI to initialize the data members to the following values:

- `name_` should be "empty"
- `price_` should be 1.0
- `category_` should be `DEFAULT_CATEGORY`
- `flags_` should be `DEFAULT_FLAGS`

Here's the code for the test to solve:

```
TEST(SalesRecord, Name) {
    SalesRecord s;
    EXPECT_EQ("empty", s.name_);
}

TEST(SalesRecord, Price) {
    SalesRecord s;
    EXPECT_DOUBLE_EQ(1.0, s.price_);
}

TEST(SalesRecord, Category) {
    SalesRecord s;
    EXPECT_EQ(DEFAULT_CATEGORY, s.category_);
}
```

```
}  
  
TEST(SalesRecord, Flags) {  
    SalesRecord s;  
    EXPECT_EQ(DEFAULT_FLAGS, s.flags_);  
}
```

You can practice with the following Compiler Explorer solution: [Point tests @Compiler Explorer](https://godbolt.org/z/Y19jMs4Gb)⁹.

⁹<https://godbolt.org/z/Y19jMs4Gb>

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. First, we'll try to understand the syntax and ways of initializing them, and then you'll learn about `std::initializer_list`.

The basics

If you have a simple structure with various containers, here are some basic ways you can initialize them in a default constructor:

Ex 9.1. Basic syntax for containers as data members, initialization in a constructor. Run [@Compiler Explorer](#)

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

struct S {
    S()
        : numbers { 1, 2, 3, 4}
        //, nums { 1, 2, 3}
        , doubles { 0.1, 1.1, 2.1 }
        , ints { 100, 101, 102}
        , moreInts( 10, 1) // 10 1's, not 10 and 1
        , names ( 10, "hello" ) // 10 "hello" strings
        , mapping { {"one", 1}, {"two", 2} }
    { }

    int numbers[4];
    // int nums[]; // need to provide the size!
    std::array<double, 3> doubles;
```



```

std::vector<int> ints;
std::vector<int> moreInts;
std::vector<std::string> names;
std::map<std::string, int> mapping;
};

int main() {
    S s;
    std::cout << "s.numbers[0]: " << s.numbers[0] << '\n';
    std::cout << "s.double[0]: " << s.doubles[0] << '\n';
    std::cout << "s.ints[0]: " << s.ints[0] << '\n';
    std::cout << "s.moreInts[9]: " << s.moreInts[9] << '\n';
    std::cout << "s.names[9]: " << s.names[9] << '\n';
    std::cout << "s.mapping[\"one\"]: " << s.mapping["one"] << '\n';
}

```

Here are the options from the example:

- `int numbers[4];` - is a regular C-style array; we can use aggregate initialization to put the values.
- The syntax with `//`, `nums { 1, 2, 3}` is not an option, as we cannot declare an array without the size and then initialize it later.
- `doubles` is `std::array<double, 3>` which is a C++11-style array, it's also an aggregate type.
- `ints` is a `std::vector` of integers, and we can use list initialization to set elements. Note that there's no need to pass the size/count of those elements.
- `moreInts` is another vector, but this time I used parens `()` to call the `vector(size_type count, const T& value = T())` constructor. In this case, braces `{}` would call the wrong constructor and create a vector with two elements 1 and 10. It's because the value type stored in the container is convertible to the size type (`size_t`). Additionally, according to core guidelines [ES.23](https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es23)¹, for constructors with sizes, it's clearer to use parens.
- `names` is a vector of `std::string`, and I also used parens `()` to call the "size" constructor. This time, the braces `{}` would also call the "size" constructor as it would be clear to the compiler that `{10, "hello"}` is not a pair of two elements of the same type.

¹<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es23>-prefer-the--initializer-syntax

- mapping is `std::map`, and we can also use a handy constructor to pass all pairs of key values at once.

A huge benefit of using standard containers (as regular data members, not pointers) is that there's no need to implement additional special member functions. Copy, move, or the assignment operator works out of the box. Not to mention that there's no need for a custom destructor.

From previous chapters you know that for default initialization we can also rely on NSDMI and set the values in the moment we declare a data member:

Ex 9.2. Containers as data members and NSDMI. Run [@Compiler Explorer](#)

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

struct S {
    int numbers[4] { 1, 2, 3, 4};
    // int nums[] { 0, 1, 2 }; // need to provide the size!
    std::array<double, 3> doubles { 0.1, 1.1, 2.1 };
    std::vector<int> ints { 100, 101, 102};
    std::vector<int> moreInts = std::vector<int>(10, 1);
    std::vector<std::string> names = std::vector<std::string>(10, "hello");
    std::map<std::string, int> mapping { {"one", 1}, {"two", 2} };
};

int main() {
    S s;
    std::cout << "s.numbers[0]: " << s.numbers[0] << '\n';
    std::cout << "s.double[0]: " << s.doubles[0] << '\n';
    std::cout << "s.ints[0]: " << s.ints[0] << '\n';
    std::cout << "s.moreInts[9]: " << s.moreInts[9] << '\n';
    std::cout << "s.names[9]: " << s.names[9] << '\n';
    std::cout << "s.mapping[\"one\"]: " << s.mapping["one"] << '\n';
}
```

In most cases, the NSDMI syntax is convenient and allows us to initialize all container-like members where we declare them. As we discussed in the section on [NSDMI and direct](#)

[initialization](#), we have to use, for example, a copy initialization to call the vector's constructor using parens (). There's no need to duplicate the code in a constructor, and the S structure can preserve its aggregate status (thus, we can leverage aggregate initialization).

Since C++11, all standard containers have a way to make a list of values into a constructor. For example, before C++11, for `std::vector`, you'd had to use `push_back` calls to populate a container with different values. How does the new Standard achieve this? See in the next section.

Using `std::initializer list`

With the idea of list initialization there also came support to pass such a list not only to aggregate types. Since C++11, you can use `std::initializer_list<T>`, a lightweight proxy object that provides access to an array of objects of type `const T`.

The Standard shows the following example [decl.init.list](#)²:

```
struct X {
    X(std::initializer_list<double> v);
};
X x{ 1,2,3 };
```

The initialization will be implemented in a way roughly equivalent to this:

```
const double __a[3] = {double{1}, double{2}, double{3}};
X x(std::initializer_list<double>(__a, __a+3));
```

In other words, the compiler creates a `const` array of and then passes you a proxy object that looks like a regular C++ container with iterators, `begin()`, `end()` and even the `size()` function. Here's a basic example that illustrates usage of this type:

²<https://timsong-cpp.github.io/cppwp/n4868/dcl.init.list#5>

Ex 9.3. A function taking `initializer_list`. Run [@Compiler Explorer](#)

```
#include <iostream>
#include <initializer_list>

void foo(std::initializer_list<int> list) {
    if (!std::empty(list)) {
        for (auto& x : list)
            std::cout << x << ", ";
        std::cout << "(" << list.size() << " elements)\n";
    }
    else
        std::cout << "empty list\n";
}

int main() {
    foo({});
    foo({1, 2, 3});
    foo({1, 2, 3, 4, 5});
}
```

In the example, there's a function taking a `std::initializer_list` of integers. Since it looks like a regular container, we can use non-member functions like `std::empty`, use it in a range-based for loop, and check its `size()`. Please notice that there's no need to pass `const initializer_list<int>&` (a `const` reference) as the initializer list is a lightweight object, so passing by value doesn't copy the referenced elements in the "hidden" array.

Note that we cannot do the same with `std::array` as the parameter to a function would have to have a fixed size. `initializer_list` has a variable length; the compiler takes care of that. Moreover, the "internal" array is created on the stack, so it doesn't require any additional memory allocation (like if you used `std::vector`). The list also takes homogenous values, and the initialization disallows narrowing conversions. For example:

```
// foo({1, 2, 3, 4, 5.5}); // error, narrowing
foo({1, 'x', '0', 10}); // fine, char converted to int
```

There's also a handy use case, where you can use range-based for loop directly with the `initializer_list`:

```
#include <iostream>

int main() {
    for (auto x : {"hello", "coding", "world"})
        std::cout << x << ", ";
}
```

The temporary `initializer_list` has extended lifetime and is visible in the scope of the loop. To see the underlying mechanism for this code, you can have a look at this [C++ Insights example](#)³.

I also have to point out that since `initializer_list` refers to some internal local array, then you should not return it:

```
std::initializer_list<int> wrong() { // for illustration only!
    return { 1, 2, 3, 4 };
}

int main() {
    std::initializer_list<int> x = wrong();
}
```

The above code is equivalent to:

```
std::initializer_list<int> wrong() {
    const int arr[] { 1, 2, 3, 4 }
    return std::initializer_list<int>{arr, arr+4};
}

int main() {
    std::initializer_list<int> x = wrong();
}
```

The example only illustrates this mistake, so you know how this type works. The function returns pointers/iterators to a local object, and that will cause undefined behavior. The compiler should warn about such usage. See a demo [@Compiler Explorer](#)⁴.

³<https://cppinsights.io/s/67363d91>

⁴<https://godbolt.org/z/bonveWf4a>

All in all, we can make the following conclusion:



`std::initializer_list` is a “view” kind of type, it references some implementation-dependent and local array of `const` values. Use it mostly for passing into functions when you need a variable number of arguments of the same type. If you try to return such lists and pass them around, then you risk lifetime issues. Use with care.

Constructors taking `std::initializer_list`

As mentioned, all containers from the Standard Library have constructors supporting `initializer_list`. For instance:

```
// the vector class:
constexpr vector( std::initializer_list<T> init,
                  const Allocator& alloc = Allocator() );

// map:
map( std::initializer_list<value_type> init,
     const Compare& comp = Compare(),
     const Allocator& alloc = Allocator() );
```

How does it work? Let’s see the basic class type with a user-declared constructor taking the list:

Ex 9.4. Test constructor with `initializer_list`. Run [@Compiler Explorer](#)

```
#include <iostream>
#include <initializer_list>

struct X {
    X(std::initializer_list<int> list)
        : count{list.size()} { puts("X(init_list)"); }
    X(size_t cnt) : count{cnt} { puts("X(cnt)"); }
    X() { puts("X()"); }
    size_t count {};
};
```

```

int main() {
    X x;
    std::cout << "x.count = " << x.count << '\n';
    X y { 1 };
    std::cout << "y.count = " << y.count << '\n';
    X z { 1, 2, 3, 4 };
    std::cout << "z.count = " << z.count << '\n';
    X w ( 3 );
    std::cout << "w.count = " << w.count << '\n';
}

```

The `X` class defines three constructors, and one of them takes `initializer_list`. If we run the program, you'll see the following output:

```

X()
x.count = 0
X(init_list)
y.count = 1
X(init_list)
z.count = 4
X(cnt)
w.count = 3

```

As you can see, writing `X x;` invokes a default constructor. Similarly, if you write `X x{}`, the compiler won't call a constructor with the empty initializer list. But in other cases, the list constructor is “greedy” and will take precedence over the regular constructor taking one argument. To call the exact constructor, you need to use direct initialization with parens `()`.

Example implementation

Let's go further with containers and have some more realistic examples. I want to show you the `Package` class that holds several `Product` objects. As an additional complexity, inside this `Package` class, let's count the total value of the package, as well as count products by name. Here's the `Product` class (similar to our previous declarations):

```

struct Product {
    Product() = default;
    Product(std::string s, double v)
        : name{std::move(s)}, value{v}
    { }

    std::string name;
    double value{};
};

```

And the Package class:

```

class Package {
public:
    void addProduct(const Product& p) {
        ++counts_[p.name];
        prods_.push_back(p);
        totalValue_ += p.value;
    }
    void printContents() const {
        for (auto& [key, val] : counts_)
            std::cout << key << ", count: " << val << '\n';
        std::cout << "total value: " << totalValue_ << '\n';
    }
private:
    std::vector<Product> prods_; // all products
    std::map<std::string, unsigned> counts_;
    double totalValue_ { };
};

```

The Package class holds all objects that we pass through the `AddProduct()` member function and also performs some internal changes: it counts the total sum of values and also adds the product to the `counts_` dictionary. We can run this code with the following client code:

Ex 9.5. The Package class demo. Run [@Compiler Explorer](#)

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

struct Product { /*as before*/ };
class Package { /*as above*/ };

int main() {
    Package pack;
    pack.addProduct({"crayons", 3.0});
    pack.addProduct({"pen", 2.0});
    pack.addProduct({"bricks", 11.0});
    pack.addProduct({"bricks", 12.0});
    pack.addProduct({"pen", 12.0});
    pack.addProduct({"pencil", 12.0});
    pack.printContents();
}
```

And not surprisingly, we'll get the following output:

```
bricks, count: 2
crayons, count: 1
pen, count: 2
pencil, count: 1
total value: 52
```

While the code looks correct, this approach has at least one inconvenience. The client must use `addProduct` several times to populate the internal containers. This can be improved by creating a constructor (as well as some function overloads) that would take whole containers:

```
Package(const std::vector<Product>& items) {
    for (const auto& elem : items)
        addProduct(elem);
}
```

Having a function taking more elements at once might be handy if you process a bulk of data. For example when you load products from a file or get a network packet.

On the other hand, for unit tests or test code, you might want to initialize your objects with a list of objects. In a case of taking `std::vector` you have to write.

```
Package pack {
    std::vector<Product>{{"pen", 1.0}, {"pencil", 2.0}}
};
```

Notice the additional pair of braces. The first pair opens a call to a constructor, and the next creates a temporary vector which is later passed to the constructor. On the other hand, can also use `std::initializer_list` and get a simpler syntax:

Ex 10.6. The Package class with `initializer_list`. Run [@Compiler Explorer](#)

```
struct Product { /*as before*/ };

class Package {
public:
    Package() = default;
    Package(std::initializer_list<Product> items) {
        for (auto& elem : items)
            addProduct(elem);
    }

    void addProduct(const Product& p) {
        ++counts_[p.name];
        prods_.push_back(p);
        totalValue_ += p.value;
    }

    void printContents() const {
        for (auto& [key, val] : counts_)
            std::cout << key << ", count: " << val << '\n';
        std::cout << "total value: " << totalValue_ << '\n';
    }
};
```

```
    }  
private:  
    std::vector<Product> prods_; // all products  
    std::map<std::string, unsigned> counts_  
    double totalValue_ { };  
};  
  
int main() {  
    Package pack {  
        {"pen", 1.0}, {"pencil", 2.0}  
    };  
    pack.addProduct({"crayons", 3.0});  
    pack.addProduct({"pen", 2.0});  
    pack.addProduct({"bricks", 11.0});  
    pack.addProduct({"bricks", 12.0});  
    pack.printContents();  
}
```

In the example above, there are two constructors: one default and another with `std::initializer_list`. I had to specify a default constructor, so it's possible to create `Package emptyPackage`;, as `initializer_list` doesn't allow us to pass empty lists `{}` or use default initialization.

What's an advantage over passing `std::vector`? Apart from nested braces, `std::vector` requires a dynamic allocation for a memory block to store its elements. For `std::initializer_list` the compiler deduces the size and creates a C-style array underneath for elements, so there's no extra allocation here.

We're also not limited to only constructors, as we can use `initializer_list` in regular functions:

```

Package(std::initializer_list<Product> items) {
    addProducts(items);
}

void addProduct(const Product& p) { /* as before */ }

void addProducts(std::initializer_list<Product> items) {
    for (auto& elem : items)
        addProduct(elem);
}

```

The example above shows just a part of implementing the `Package` class. I created a new member function, `addProducts`, which takes `initializer_list` and calls `addProduct` to perform the main job. The constructor is also updated to call the new function and doesn't duplicate code.

More options (advanced)

Is `std::initializer_list` the best way to pass a list of homogenous values? It has some uses and might be good enough for classes that look like containers and you want to provide a handy way of a passing a list of values at once. But, we can also leverage some template techniques and use variadic function templates.

Ex 9.7. Variadic function template. Run [@Compiler Explorer](#)

```

template<typename... Ts>
requires (std::same_as<Ts, Product> && ...)
void addProducts(const Product & first, const Ts&... args) {
    addProduct(first);
    (addProduct(args), ...);
}

// and later:
pack.addProducts({"pencil", 12.0}, Product{"pen", 10});
//pack.addProducts({"pencil", 12.0}, 10); // error, 10 is not a Product

```

I don't want to go into full details, as it's outside the scope of the book, but here are the core features that enable such code:

- Variadic templates allow us to pass any number of arguments into a function and process it with argument pack syntax.
- Concepts from C++20 add a way to require all input types to be `Product`. For example in the last line I tried to pass `10` as the second argument to the function, and compiler will generate an error that the integer `10` doesn't match with the concept requirements.
- `(addProduct(args), ...)`; is a fold expression over a comma operator that nicely expands the argument pack at compile time. Fold expressions are available since C++17.
- The code might also be updated with rvalue references (forming a universal reference) which would be forwarded to the internal function.

For more information about those techniques have a look at articles at the C++Stories blog: [C++20 Concepts - a Quick Introduction⁵](#), [C++ Templates: How to Iterate through `std::tuple`: the Basics⁶](#).



`std::initializer_list` has a bit of a bad reputation in C++. You can see this Jason Turner's talk from C++Now 2018, "[Initializer Lists are Broken — Let's Fix Them.](#)"⁷ and understand different solutions to passing lists to a function. And look at this article by Andrzej Krzemiński about [The cost of `std::initializer_list`](#)⁸.

Summary

In this chapter, we discussed having various containers as data members. If we use containers from the Standard Library, then they handle all memory management and allocations. We can use those standard containers as regular types without worrying about custom implementation for special member functions. Thanks to the NSDMI feature, we can safely initialize them with a convenient syntax. In the second part of the chapter, you learned about `initializer_list`, which is an option to pass multiple values at once with a handy API. The `initializer_list` type is only a view of an internal array of `const` objects, so please use it carefully. Additionally, if you have a constructor taking the list, then it will be "greedy" and takes priority over other non-default constructors.

⁵<https://www.cppstories.com/2021/concepts-intro/>

⁶<https://www.cppstories.com/2022/tuple-iteration-basics/>

⁷<https://www.youtube.com/watch?v=sSImmZMFsXQ>

⁸https://akrzemi1.wordpress.com/2016/07/07/the-cost-of-stdinitializer_list/

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++Reference¹](#).

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

Consider the following code with a `const` data member `id_`:

Ex 10.1. A constant data member. Run [@Compiler Explorer](#)

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

class Product {
public:
    explicit Product(const char* name, unsigned id)
        : name_(name)
        , id_(id)
    { }

    const std::string& name() const { return name_; }
    void name(const std::string& name) { name_ = name; }
    unsigned id() const { return id_; }

private:
```

¹<https://en.cppreference.com/w/cpp/concepts/regular>

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

int main() {
    Product tvset("TV Set", 123);
    std::cout << tvset.name() << ", id: "
               << tvset.id() << '\n';
}
```

In the example above, the class `Product` has a constant data member `id_`. This member is set during construction; we can initialize it at the point where the object is created.

While the code seems fine, you might be surprised when you try to change the value of `tvset`:

```
tvset = Product("TV Set 2022", 456);
```

The line generates a compilation error! It's because the compiler tries to invoke an assignment operator, but since the data member is constant, it cannot change it. When a data member is `const`, the compiler won't generate a default assignment operator for us.

The only way to fix the compiler error is to write an assignment operator. We can work only with mutable data members (copy them) in one approach and leave constant members. The problem is that it's pretty complicated. Moreover, it might be misleading to the reader, as it's not easy to reason what is changing.

Having a `const` data member with a custom assignment operator might also cause troubles in standard containers. Consider the following code:

```
std::vector<Product> prods;
prods.push_back(Product("box", 234));
prods.push_back(Product("car", 567));
prods.insert(prods.begin(), Product("ball", 987));
```

While the `push_back` calls work fine (as the compiler can successfully create objects inside the container), there's an issue with the `insert()`, and you'll get a compiler error.

GCC might report the following:

```
error: use of deleted function 'Product& Product::operator=(Product&&)'  
430 |         *__position = std::forward<_Arg>(__arg);
```

You can experiment with this example [@Compiler Explorer²](#).

If you want your object to be constant, make it const as a “whole” rather than just some of its parts.



You can read more about this semantic problem with constant data members in a good overview by Arthur O'Dwyer, see at [const all the things?³](#).

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.

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.

²<https://godbolt.org/z/PchPnn84E>

³<https://quuxplusone.github.io/blog/2022/01/23/dont-const-all-the-things/>

11. Non-local objects

Thus far, we considered variables that appeared in some local function scope or as sub-objects of a class type. However, this is not the only option, and C++ allows us to declare various forms of non-local objects: they usually live throughout the execution of the whole program. In this chapter, we'll look at global variables, static data members, and thread-local objects. We'll also consider new features for safe initialization from C++17 and C++20.

Storage duration and linkage

To start, we need to understand two key properties of an object in C++: *storage* and *linkage*. Let's begin with the definition of *storage*, from [basic.stc#general¹]:

The storage duration is the property of an object that defines the minimum potential lifetime of the storage containing the object. The storage duration is determined by the construct used to create the object.

An object in C++ has one of the following storage duration options:

Storage duration	Explanation
<i>automatic</i>	Automatic means that the storage is allocated at the start of the scope. Most local variables have automatic storage duration (except those declared as <code>static</code> , <code>extern</code> , or <code>thread_local</code>).
<i>static</i>	The storage for an object is allocated when the program begins (usually before the <code>main()</code> function starts) and deallocated when the program ends. There's only one instance of such an object in the whole program.
<i>thread</i>	The storage for an object is tied to a thread: it's started when a thread begins and is deallocated when the thread ends. Each thread has its own "copy" of that object.
<i>dynamic</i>	The storage for an object is allocated and deallocated using explicit dynamic memory allocation functions. For example, by the call to <code>new/delete</code> .

¹<https://timsong-cpp.github.io/cppwp/n4868/basic.stc#general>

And the definition for the second property: *linkage*, extracted from [basic.link²]:

A name is said to have linkage when it can denote the same object, reference, function, type, template, namespace, or value as a name introduced by a declaration in another scope.

We have several linkage types:

Linkage	Explanation
<i>external linkage</i>	External means that the name can be referred to from the scopes in the same or other translation units. Non-const global variables have external linkage by default.
<i>module linkage</i>	Available since C++20. A name can be referred in scopes of the same module or module units.
<i>internal linkage</i>	A name can be referred to from the scopes in the same translation units. For example, a <code>static</code> , <code>const</code> , <code>constexpr</code> global variables have internal linkage.
<i>no linkage</i>	Cannot be referred from other scopes.
<i>language linkage</i>	Allows interoperability between different programming languages, usually with C. For example, by declaring <code>extern "C"</code>

If we work with regular variables declared in a function's scope, the storage is automatic, and there's no linkage, but those properties matter for objects in a global or thread scope. In the following sections, we'll try some experiments with global objects to understand the meaning of those definitions.

Static duration and external linkage

Consider the following code:

²<https://timsong-cpp.github.io/cppwp/n4868/basic.link#2>

Ex 11.1. Static and automatic objects. Run [@Compiler Explorer](#)

```
#include <iostream>

struct Value {
    Value(int x) : v(x) { std::cout << "Value(" << v << ")\n"; }
    ~Value() noexcept { std::cout << "~Value(" << v << ")\n"; }

    int v {0};
};

Value v{42};

int main() {
    puts("main starts...");
    Value x { 100 };
    puts("main ends...");
}
```

If we run the example, you'll see the following output:

```
Value(42)
main starts...
Value(100)
main ends...
~Value(100)
~Value(42)
```

In the example, there's a structure called `Value`, and I declare and define a global variable `v`. As you can see from the output, the object is initialized **before** the `main()` function starts and is destroyed after the `main()` ends.

The global variable `v` has a static storage duration and external linkage. On the other hand, the second variable, `x`, has no linkage and automatic storage duration (as it's a local variable).

If we have two translation units: `main.cpp` and `other.cpp`, we can point to the same global variable by declaring and defining an object in one place and then using the `extern` keyword to provide the declaration in the other translation unit. This is illustrated by the following example:

Ex 11.2. Static and **extern**. Run @Wandbox

```
// main.cpp
#include <iostream>
#include "value.h"

Value v{42};
void foo();

int main() {
    std::cout << "in main(): " << &v << '\n';
    foo();
    std::cout << "main ends...\n";
}

// other.cpp
#include "value.h"

extern Value v; // declaration only!

void foo() {
    std::cout << "in foo(): " << &v << '\n';
}
```

If we run the code, you'll see that the address of `v` is the same in both lines. For instance:

```
Value(42)
in main(): 0x404194
in foo(): 0x404194
main ends...
~Value(42)
```

Internal linkage

If you want two global variables visible as separate objects in each translation unit, you need to define them as `static`. This will change their linkage from external to internal.

Ex 11.3. Static and internal linkage. Run [@Wandbox](#)

```
// main.cpp
#include <iostream>
#include "value.h"

static Value v{42};
void foo();

int main() {
    std::cout << "in main(): " << &v << '\n';
    foo();
    std::cout << "main ends...\n";
}

// other.cpp
#include "value.h"

static Value v { 100 };

void foo() {
    std::cout << "in foo(): " << &v << '\n';
}
```

Now, you have two different objects which live in the static storage (outside `main()`):

```
Value(42)
Value(100)
in main(): 0x404198
in foo(): 0x4041a0
main ends...
~Value(100)
~Value(42)
```

You can also achieve this by wrapping objects in an anonymous namespace:

```
namespace {  
    Value v{42};  
}
```

Additionally, if you declare `const Value v{42};` in one translation unit, then `const` implies an internal linkage. If you want to have a `const` object with the external linkage, you need to add the `extern` keyword:

```
// main.cpp:  
extern const Value v { 42 }; // declaration and definition!  
  
// other.cpp:  
extern const Value v; // declaration
```



While constant global variables might be useful, try to avoid mutable global objects. They complicate the program's state and may introduce subtle bugs or data races, especially in multithreaded programs. In this chapter, we cover all global variables so that you can understand how they work, but use them carefully. See this C++ Core Guideline: [I.2: Avoid non-const global variables](https://ericniebler.com/2017/02/avoid-non-const-global-variables/)³.

Thread local storage duration

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

Dynamic storage duration

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

Initialization of non-local static objects

All non-local objects are initialized before `main()` starts and before their first “use”. But there's more to that.

Consider the following code:

³<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#i2-avoid-non-const-global-variables>

Ex 11.4. Static initialization. Run [@CompilerExplorer](#)

```
#include <iostream>

struct Value { /*as before*/ };

double z = 100.0;
int x;
Value v{42};

int main() {
    puts("main starts...");
    std::cout << x << '\n';
    puts("main ends...");
}
```

All global objects `z`, `x`, and `v` are initialized during the program startup and before the `main()` starts. We can divide the initialization into two distinct types: *static initialization* and *dynamic initialization*.

The static initialization occurs in two forms:

- **constant initialization** - this happens for the `z` variable, which is value initialized from a constant expression.
- The `x` object looks uninitialized, but for non-local static objects, the compiler performs **zero initialization**, which means they will take the value of zero (and then it's converted to the appropriate type). Pointers are set to `nullptr`, arrays, trivial structs, and unions have their members initialized to a zero value.



Don't rely on zero initialization for static objects. Always try to assign some value to be sure of the outcome. In the book, I only showed it so you can see the whole picture.

Now, `v` global objects are initialized during so-called **dynamic initialization** of non-local variables". It happens for objects that cannot be constant initialized or zero-initialized during static initialization at the program startup.



In a single translation unit, the order of dynamic initialization of global variables (including static data members) is well defined. If you have multiple compilation units, then the order is unspecified. When a global object A defined in one compilation unit depends on another global object B defined in a different translation unit, you'll have undefined behavior. Such a problem is called the “static initialization order fiasco”; read more [C++ Super FAQ](#)⁴.

In short, each static non-local object has to be initialized at the program startup. However, the compiler tries to optimize this process and, if possible, do as much work at compile time. For example, for built-in types initialized from constant expressions, the value of the variable might be stored as a part of the binary and then only loaded during the program startup. If it's not possible, then a dynamic initialization must happen, meaning that the value is computed once before the `main()` starts. Additionally, the compiler might even defer the dynamic initialization until the first use of the variable but must guarantee the program's correctness. Since C++11, we can try to move dynamic initialization to the compile-time stage thanks to `constexpr` (allowing us to write custom types). Since C++20, we can use `constinit` to guarantee constant initialization.



For more information, have a look at this good blog post for more information: [C++ - Initialization of Static Variables by Pablo Arias](#)⁵ and also a presentation by Matt Godbolt: [CppCon 2018 “The Bits Between the Bits: How We Get to main\(\)”](#)⁶.

Static variables in a function scope

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

`constinit` in C++20

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

⁴<https://isocpp.org/wiki/faq/ctors#static-init-order>

⁵<https://pabloariasal.github.io/2020/01/02/static-variable-initialization/>

⁶<https://www.youtube.com/watch?v=dOfucXtyEsU>

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 example 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 11.5. Simple **static** Data Member. Run [@Compiler Explorer](#)

```
#include <iostream>

struct Value {
    int x;

    static int y; // declaration
};

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.

To be precise, `Value::y` has a static storage duration and external linkage.



Local classes or unnamed classes cannot have static data members.

Here's an example that illustrates the lifetime of a static data member:

Ex 11.6. **static** Data Member Lifetime. Run [@Compiler Explorer](#)

```
#include <iostream>

struct Value {
    Value(int x) : v(x) { std::cout << "Value(" << v << ")\n"; }
    ~Value() noexcept { std::cout << "~Value(" << v << ")\n"; }

    int v {0};
};

struct Test {
    Test() { puts("Test::Test()"); }
    ~Test() noexcept { puts("Test::~Test()"); }

    static Value u;
    static Value v;
    static int w;
    static double z;
};

Value Test::v { 42 };
Value Test::u { 24 };
```

```
int Test::w;  
double Test::z = 10.5f;  
  
int main() {  
    puts("main starts...");  
    Test x;  
    std::cout << Test::w << '\n';  
    std::cout << Test::z << '\n';  
    puts("main ends...");  
}
```

The code has the `Value` structure that has two instances in the form of two static data members inside the `Test` class. Additionally, we have two other data members, `w` and `z`, which are built-in types. If we run the code, you'll see the following output:

```
Value(42)  
Value(24)  
main starts...  
Test::Test()  
0  
10.5  
main ends...  
Test::~~Test()  
~Value(24)  
~Value(42)
```

As you can see, two `Value` objects are created before the `main` starts, in the order of definitions in a file (not declarations!). After the `main()` function ends, the two objects are destroyed in reverse order.

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 examples 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:

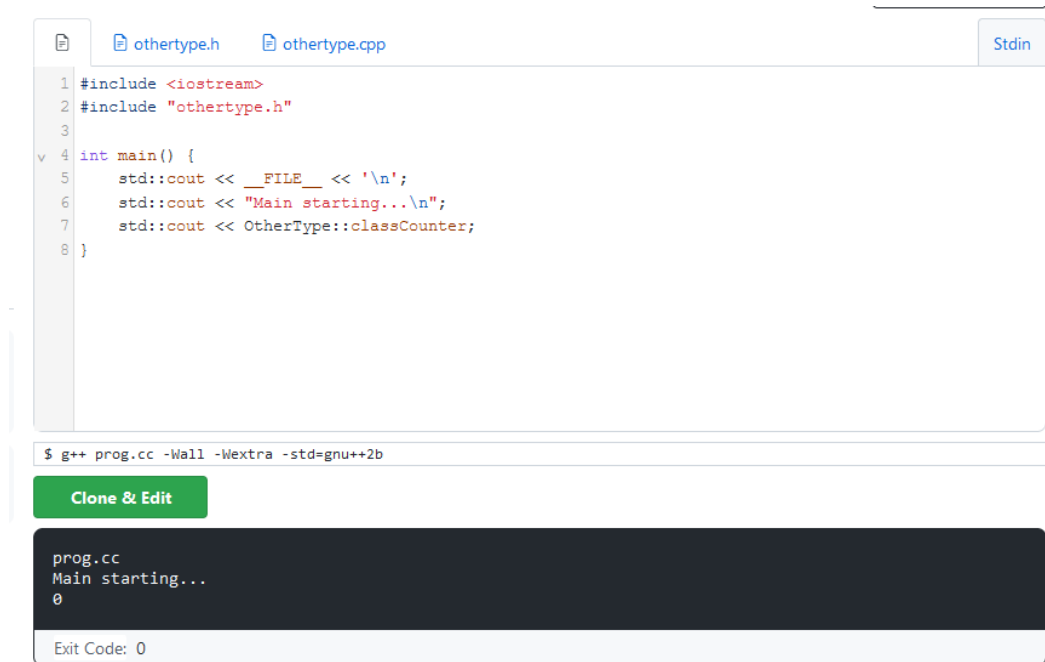
Ex 11.7. Static data member, multiple files. Run @Wandbox

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

    // ...
};

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

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



The screenshot shows the Wandbox online compiler interface. At the top, there are tabs for 'othertype.h' and 'othertype.cpp'. The 'othertype.h' tab is active, showing the following code:

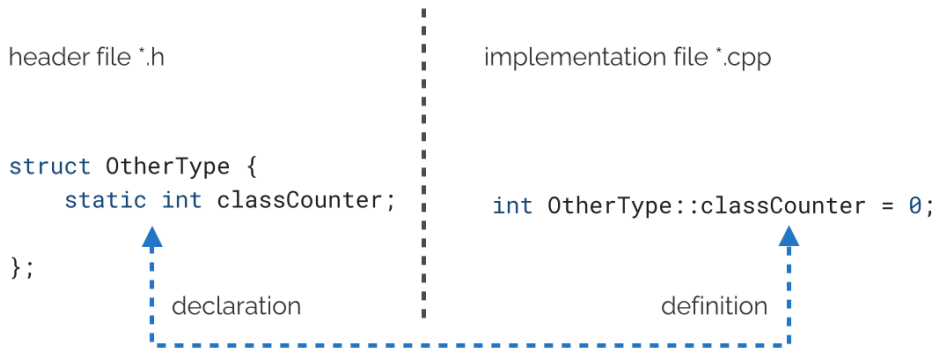
```
1 #include <iostream>
2 #include "othertype.h"
3
4 int main() {
5     std::cout << __FILE__ << '\n';
6     std::cout << "Main starting...\n";
7     std::cout << OtherType::classCounter;
8 }
```

Below the code editor, there is a command line with the command: `$ g++ prog.cc -Wall -Wextra -std=gnu++2b`. A green button labeled 'Clone & Edit' is visible. Below the command line, the output of the program is shown in a dark box:

```
prog.cc
Main starting...
0
```

At the bottom, the 'Exit Code' is 0.

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 11.8. 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 that they will be initialized before the `main()` function is called.

If we read [Dynamic Initialization @C++Reference](https://en.cppreference.com/w/cpp/language/initialization#Dynamic_initialization)⁷ and [C++ Standard: basic.start](https://timsong-cpp.github.io/cppwp/n4868/basic.start.dynamic#3)⁸ we get the following rules about the initialization order:

Partially-ordered dynamic initialization, which applies to all inline variables that are not an implicitly or explicitly instantiated specialization. If a partially-ordered V is defined before ordered or partially-ordered W in every translation unit, the initialization of V is sequenced before the initialization of W.

Based on the previous code, here's the example with the `Value` class and multiple compilation units:

Ex 11.9. Inline Variables and multiple compilation units, test.h. Run [@Wandbox](#)

```
// test.h
#include <iostream>

struct Value {
    Value(int x) : v(x) { std::cout << "Value(" << v << ")\n"; }
    ~Value() noexcept { std::cout << "~Value(" << v << ")\n"; }

    int v {0};
};

struct Test {
    Test() { puts("Test::Test()"); }
    ~Test() noexcept { puts("Test::~Test()"); }

    static inline Value u { 42 };
    static inline Value v { 24 };
};
```

⁷https://en.cppreference.com/w/cpp/language/initialization#Dynamic_initialization

⁸<https://timsong-cpp.github.io/cppwp/n4868/basic.start.dynamic#3>

Ex 11.9. Inline Variables and multiple compilation units, main. Run [@Wandbox](#)

```
// main.cpp
#include <iostream>
#include "test.h"

void foo();

static Value local{100};

int main() {
    std::cout << "Main starting...\n";
    foo();
    Test t;
}
```

Ex 11.9. Inline Variables and multiple compilation units, other and noname. Run [@Wandbox](#)

```
// other.cpp
#include "test.h"

static Value local{200};

void foo() {
    std::cout << "foo starting...\n";
    Test t;
}

// noname.cpp
#include "test.h"

static Value local{300};
```

The build command line:

```
$ g++ prog.cc -Wall -Wextra -std=c++2a noname.cpp other.cpp
```

The output:

```

Value(42)
Value(24)
Value(100)
Value(300)
Value(200)
Main starting...
foo starting...
Test::Test()
Test::~~Test()
Test::Test()
Test::~~Test()
~Value(200)
~Value(300)
~Value(100)
~Value(24)
~Value(42)

```

As you notice, `Value(42)` and `Value(24)`, inline variables, are initialized before all other `Value` global objects. Moreover, depending on the command line, `Value(200)` might be created before `Value(300)`.

The Inline variables make 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; }
};

```

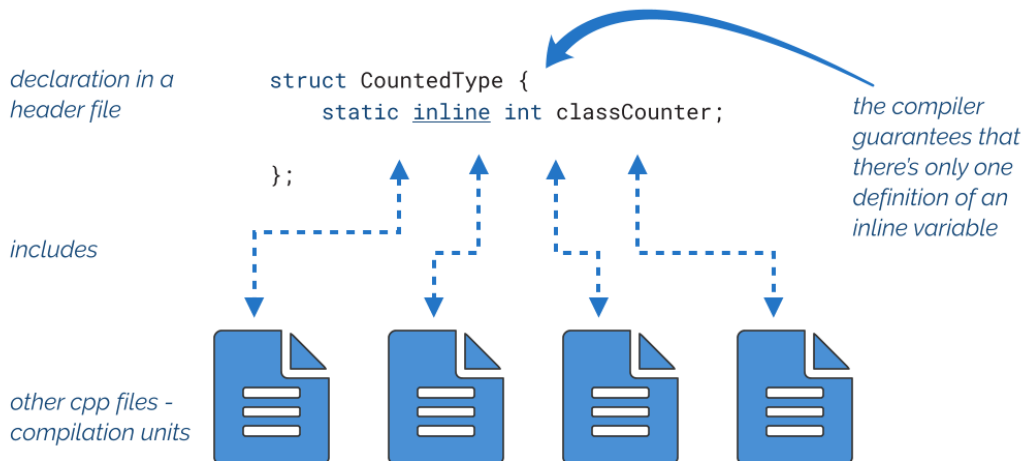
And the `main()` function:

Ex 11.10. 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.



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

We can combine our knowledge about constructors and inline variables and continue with the `CountedType`.

Please implement the support for other constructors so that the following test passes.

```
struct CountedType {
    static inline int instanceCounter = 0;
    static inline int maxInstanceCounter = 0;

    // your code here...
};
```

And here are the test:

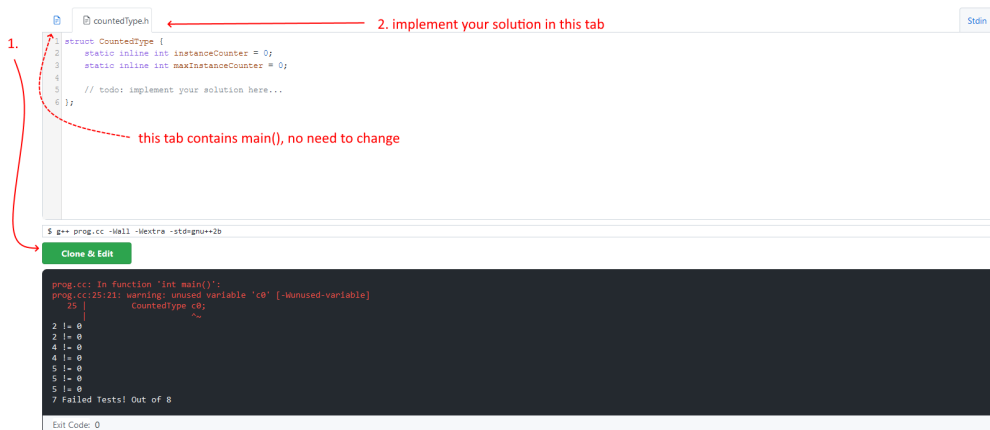
```
int main() {
    {
        CountedType c0;
        CountedType c1;
        Tests::Expect(2, CountedType::instanceCounter);
        Tests::Expect(2, CountedType::maxInstanceCounter);

        CountedType c2(c1);
        CountedType c3(c1);
        Tests::Expect(4, CountedType::instanceCounter);
        Tests::Expect(4, CountedType::maxInstanceCounter);

        CountedType c4(std::move(c1));
        Tests::Expect(5, CountedType::instanceCounter);
        Tests::Expect(5, CountedType::maxInstanceCounter);
    }
    Tests::Expect(0, CountedType::instanceCounter);
    Tests::Expect(5, CountedType::maxInstanceCounter);
}
```

As you can see, the example creates several `CountedType` instances and then checks (via `Test::Expect`) if the counters are correct.

Start from the following runnable code sample [@Wandbox⁹](https://wandbox.org/permlink/GuGzTWKF8irN2YLz), Click “Clone & Edit” to start the example and make changes.



The starting point for the exercise

Global inline variables

While we covered `inline` variables in the context of static data members for a class type, it's not the only use case. As of C++17, you can also declare `inline` variables in the global scope.

Have a look at this basic header file with the declaration and definition of two inline global variables:

```

// globals.h
#include <string>

inline constexpr int gMyGlobal { 10 };
inline const std::string gHelloText {"Hello World "};

// or better in a namespace
namespace appConstants {
    inline constexpr double scalingFactor { 1.33 };
    inline const std::string appName { "Testing app" };
}

```

⁹<https://wandbox.org/permlink/GuGzTWKF8irN2YLz>

And you can now use those variables in the `main()`:

Ex 11.11. **const** global variables. Run [@Wandbox](#)

```
// main.cpp
#include <iostream>
#include "globals.h"

int main() {
    std::cout << gMyGlobal << ", " << gHelloText << '\n';
    std::cout << appConstants::scalingFactor << ", "
               << appConstants::appName << '\n';
}
```

Before C++17, you'd had to declare such variables in a header file as `extern`, and provide their definition in one compilation unit. Thanks to C++17, such a use case is now greatly simplified.

Read more about details in this great post from Fluent C++: [What Every C++ Developer Should Know to \(Correctly\) Define Global Constants¹⁰](#).

constexpr and inline variables

Throughout the book, we briefly spoke about `constexpr` variables. They are convenient for built-in types like integers and other trivial types. When you apply this keyword to a variable, the compiler might compute its value at compile time and thus save time at runtime.

The compiler automatically implies `inline` when you have a `static constexpr` data member in your class type. See the following example:

```
struct Value {
    static constexpr int basic { 10 };
    static constexpr auto name { "Hello World" };
};
```

When we run this code through C++Insights (run [@this link¹¹](#)), we'll see that the compiler applied `inline` to both of those variables:

¹⁰<https://www.fluentcpp.com/2019/07/23/how-to-define-a-global-constant-in-cpp/>

¹¹<https://cppinsights.io/s/a76b8133>

```
struct Value {  
    inline static constexpr const int basic = {10};  
    inline static constexpr const char *const name = {"Hello World"};  
};
```

Please remember that implicit `inline` applies only to static class data members. The compiler won't do anything when you declare a global `static constexpr` variable.

Summary

Here are the essential items to remember from this chapter:

- Non-local objects have two fundamental properties: storage duration (where they are stored), and linkage (how can we access them).
- When a non-local object is initialized with a constant expression and the object's constructor is `constexpr` (including built-in types), the initialization may happen at compile time (static initialization).
- If constant initialization (or zero initialization) is not possible, then a static object is initialized in the dynamic initialization stage.
- In a single translation unit, the order of dynamic initialization of non-local variables (including static data members) is performed in the definition order, but it's unspecified across multiple compilation units.
- A static data member is not bound to any class instances.
- A static data member has a static storage duration and external linkage.
- The compiler and the linker ensure there's only one definition of an `inline` variable; there's no need to define and declare such variables in different places.
- The compiler automatically implies `inline` when you have a `static constexpr` data member.

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](https://ericniebler.com/2020/02/05/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://ericniebler.com/2020/02/05/dcl-init-aggr/>

Ex 12.1. 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';
    std::array floats { 10.1f, 20.2f, 30.3f };
    std::cout << "floats[0] " << floats[0] << '\n';
    std::array params {Param{"val", 10}, Param{"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. Here's a short example with major use cases and its limitations:

Ex 12.2. Initialization with round parens in C++20. Run [@Compiler Explorer](#)

```

struct Point { int x; int y; };
struct PointExt { Point pt; int z; };

int main() {
    // C++20 and parens:
    Point pt (1, 2);
    // Point pt = (1, 2); // doesn't work, wrong syntax
    Point pt1 = {1, 2}; // fine with braces

    //Point pt2 { 1.1, 2.2 }; // narrowing prevented
    Point pt3 ( 1.1, 2.2 ); // narrowing is fine

    PointExt pt4 { 4, 5, 6}; // brace elision works
    //PointExt pt5 ( (4, 5), 6); // nesting doesn't work
    // PointExt pt5 ( 4, 5, 6); // brace elision doesn't work
    PointExt pt5 ( Point(4, 5), 6); // need to be explicit

    double params[] (9.81, 3.14, 1.44);
    // double paramsDeduced[] = (9.81, 3.14, 1.44); // wrong syntax
    int arrX[10] (1, 2, 3, 4); // rest is 0
}

```

Here are some basic rules about the new way of initialization:

- You have to use direct initialization (`args...`) not copy style `=(args...)`. In the example `pt = (1, 2);` fails to compile, while `Point pt1 = {1, 2};` works. Similar rule works for arrays.
- On the other hand, braces `{}` prevent narrowing conversion, but parens `()` allows it. In the example `pt3` will be initialized with 1 and 2 which are truncated from 1.1 and 2.2 double values.
- When you use braces, you can skip braces for nested types. This feature is not possible with parens and you have to be explicit about sub-objects.

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 12.2. 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 our aggregate has no user-declared constructors, then such syntax generates errors. With the C++20 change, the code works fine now.

We can also try another example that compiles in C++20, but failed before:

Ex 13.3. Aggregates and parens for `emplace_`. Run [@Compiler Explorer](#)

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

int main() {
    std::vector<Point> points;
    points.emplace_back(10, 20);
}
```

The `emplace_back()` function takes arguments and creates a `Point` object at the end of the vector. This is an alternative to `push_back` which requires to pass already created object and then a copy of the object is put at the end of the container. `emplace_back` uses `()` to create an object, and before C++20 aggregates had issues to work with this function.

If you like to know more, I highly recommend reading [C++20's parenthesized aggregate initialization has some downsides – Arthur O'Dwyer²](#), which discusses pros and cons of this new initialization syntax. Plus have a look at this short lighting-talk from ACCU 2022 by Timur Doumler: [Lightning Talk - Direct Aggregate Initialisation³](#).

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.

²<https://quuxplusone.github.io/blog/2022/06/03/aggregate-parens-init-considered-kind-bad/>

³https://www.youtube.com/watch?v=1_2e8r4zXJg

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:

```
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

The following rules apply to designated initializers:

- designated initializers work only for aggregate initialization, so they only support aggregate types,
- designated initialization requires braces `{}` and doesn't support C++20 initialization with parens `()`,

- designators can only refer to non-static data members,
- designators in the initialization expression must have the same order of data members in a class declaration,
 - this is unlike the C language, where you can put designators in any order,
- not all data members must be specified in the expression,
- you cannot mix regular initialization with designators,
- there can only be one designator for a data member,
- you cannot nest designators.

Here's a simple example that illustrates the main errors with designated initializers:

```
struct Date {  
    int year;  
    int month;  
    int day;  
  
    static int mode;  
};  
  
Date d { .mode = 10 };           // error, mode is static!  
Date d { .day = 1, .year = 2010 }; // error, out of order!  
Date d { 2050, .month = 12 };    // error, mix!
```

The code above illustrates several cases where designated initializers won't work: static data member, use out-of-order initialization, or a mix. In all cases, the compiler generates an error.

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

Let's take a look at some examples:

Ex 12.4. Designated initializers demo. Run [@Compiler Explorer](#)

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

struct Product {
    std::string name_;
    bool inStock_ { false };
    double price_ = 0.0;
};

void Print(const Product& p) {
    std::cout << "name: " << p.name_ << ", in stock: "
               << std::boolalpha << p.inStock_ << ", price: "
               << p.price_ << '\n';
}

struct Time { int hour; int minute; };
struct Date { Time t; int year; int month; int day; };

int main() {
    Product p { .name_ = "box", .inStock_ {true} };
    Print(p);

    Date d { .t { .hour = 10, .minute = 35 },
             .year = 2050, .month = 5, .day = 10 };

    // pass to a function:
    Print({.name_ = "tv", .inStock_ {true}, .price_{100.0}});

    // not all members used:
    Print({.name_ = "car", .price_{2000.0}});
}
```

It's also interesting that we can use designated initialization inside another designated

initialization. For example:

```
struct Time { int hour; int min; };
struct Date { Time t; int year; int month; int day; };

Date d { .t { .hour = 10, .min = 6 }, .year = 2050, .month = 5, .day = 10 };
```

However, we can't use "nested" ones, like:

```
Date d { .t.hour = 10, .t.min = 35, .year = 2050, .month = 5, .day = 10 };
```

The syntax `.t.hour` does not work.

As another demo, we can create an almost JSON-like structure ⁴:

Ex 12.5. JSON-like structure initialization. Run [@Compiler Explorer](#)

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

struct Date { int year; int month; int day; };
struct Team { std::string name; std::string where; };
struct GameSession {
    std::string game;
    std::string localization;
    std::vector<Team> teams;
    Date date;
};

int main() {
    GameSession test {
        .game = "Pong",
        .localization = "Pacific Ocean",
        .teams = {
            Team {
                .name = "Johny Test",
                .where = "Arctica",
```

⁴Thanks to Mariusz Jaskółka for inspiration.

```
    },  
    Team {  
        .name = "Jane Doe",  
        .where = "Antarctic",  
    },  
},  
.date = {  
    .year = 2022,  
    .month = 10,  
    .day = 6  
},  
};  
}
```

Thanks to designated initializers, we can create large objects and still be able to assign values in a readable form. Such `GameSession` from the above demo might be handy for some unit test scenarios.

Summary

When you have a simple type composed of all public data members, without complex special member functions, C++ allows you to create them in a simplified manner. What's more, C++ evolves to streamline the work even more. For example, in C++20, we get Designated Initializers which usually yield a more readable way of initializing aggregate types. Additionally, C++20 also extended the use of regular parens `()` for initialization, so the “factory” function can easily be more generic and work with both aggregate and complex classes.

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

¹https://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](https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/).

Best way to initialize string data members

See the following example:

```
class UserName {
    std::string name_;
public:
    explicit UserName(const std::string& str) : name_(str) { }
};
```

As you can see, a constructor is taking `const std::string& str`.

Let's compare those alternative implementations in three cases: creating from a string literal, creating from an lvalue, and creating from an rvalue reference:

```
// creation from a string literal
UserName u1{"John With Very Long Name"};

// creation from lvalue:
std::string s2 {"Marc With Very Long Name"};
UserName u2 { s2 };
// use s2 later...

// from rvalue reference
std::string s3 {"Marc With Very Long Name"};
UserName u3 { std::move(s3) };

// third case is also similar to taking a return value:
std::string GetString() { return "some string..."; }
UserName u4 { GetString() };
```

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

Please note that allocations/creation of `s2` and `s3` are not taken into account; we're only looking at what happens for the constructor call. For `s2` we can also assume it's used later in the code.

For `const std::string&`:

- `u1` - two allocations: the first one creates a temp string and binds it to the input parameter, and then there's a copy into `name_`.
- `u2` - one allocation: we have a no-cost binding to the reference, and then there's a copy into the member variable.
- `u3` - one allocation: we have a no-cost binding to the reference, and then there's a copy into the member variable.
- You'd have to write a ctor taking rvalue reference to skip one allocation for the `u1` case, and also, that could skip one copy for the `u3` case (since we could move from rvalue reference).

However, since the introduction of move semantics in C++11, it's usually better and safer to pass `string` as a value and then move from it.

For example:

```
class UserName {
    std::string name_;

public:
    explicit UserName(std::string str) : name_(std::move(str)) { }
};
```

Now we have the following results:

For `std::string`:

- `u1` - one allocation - for the input argument and then one move into the `name_`. It's better than with `const std::string&` where we got two memory allocations in that case.
- `u2` - one allocation - we have to copy the value into the argument, and then we can move from it.
- `u3` - no allocations, only two move operations - that's better than with `const string&`!

When you pass `std::string` by value, not only is the code simpler, but there's also no need to write separate overloads for rvalue references.

The approach with passing by value is consistent with item 41 - "Consider pass by value for copyable parameters that are cheap to move and always copied" from Effective Modern C++ by Scott Meyers.

However, is `std::string` cheap to move?

Although the C++ Standard doesn't specify that, usually, strings are implemented with **Small String Optimisation (SSO)** - the string object contains extra space to fit characters without additional memory allocation³. That means that moving a string is the same as copying it. And since the string is short, the copy is also fast.

Let's reconsider our example of passing by value when the `string` is short:

```
UserName u1{"John"}; // fits in SSO buffer

std::string s2 {"Marc"}; // fits in SSO buffer
UserName u2 { s2 };

std::string s3 {"Marc"}; // fits in SSO buffer
UserName u3 { std::move(s3) };
```

Remember that each move is the same as a copy in a case of a short string.

For `const std::string&`:

- `u1` - two copies: one copy from the input string literal into a temporary string argument, then another copy into the member variable.
- `u2` - one copy: the existing string is bound to the reference argument, and then we have one copy into the member variable.
- `u3` - one copy: the rvalue reference is bound to the input parameter at no cost; later we have a copy into the member field.

For `std::string`:

³SSO is not standardized and prone to change. MSVC (VS 2013 and above)/GCC (8.1 and above) - it's a buffer of 16 bytes, and the empty string has a size of 32 bytes. It means a space for 15 characters of the `char` type, or 7 for `wchar_t`. In Clang (6.0 and above when compiled with `-stdlib=libc++`) the buffer might contain space for 22 characters of `char` but might be only a few characters for `wchar_t`. The size of an empty string is only 24 bytes. For multiplatform code, it's not a good idea to assume optimizations based on SSO. Read more at this good article: <https://shaharmike.com/cpp/std-string/>.

- u1 - two copies: the input argument is created from a string literal, and then there's a copy into `name_`.
- u2 - two copies: one copy into the argument, and then there's the second copy into the member.
- u3 - two copies: one copy into the argument (move means copy in this particular case), and then there's the second copy into the member.

As you see, short strings passing by value might be “slower” when you pass some existing string simply because you have two copies rather than one. On the other hand, the compiler might optimize the code better when it sees an object and not a reference. Moreover, short strings are cheap to copy, so the potential “slowdown” might not even be visible.

All in all, passing by value and then moving from a string argument is the preferred solution. You have simple code and better performance for larger strings.

As always, if your code needs maximum performance, then you have to measure all possible cases.



Other Types & Automation

The problem discussed in this section can also be extended to other copyable and movable types. If the move operation is cheap, passing by value might be better than by reference. You can also use automation, like Clang-Tidy, which can detect potential improvements. Clang Tidy has a separate rule for that use case, see [clang-tidy - modernize-pass-by-value⁴](https://clang.llvm.org/extra/clang-tidy/checks/modernize-pass-by-value.html).

Here's the summary of string passing and initialization of a string member:

Input Parameter	<code>const string&</code>	<code>string</code> and <code>move</code>
<code>const char*</code>	2 allocations	1 allocation + move
<code>const char*</code> SSO	2 copies	2 copies
<code>lvalue</code>	1 allocation	1 allocation + 1 move
<code>lvalue</code> SSO	1 copy	2 copies
<code>rvalue</code>	1 allocation	2 moves
<code>rvalue</code> SSO	1 copy	2 copies

This part covered only the basic approach with string references and string copies. We could

⁴<https://clang.llvm.org/extra/clang-tidy/checks/modernize-pass-by-value.html>

also extend this discussion and cover `std::string_view` added in C++17. If you want a complete comparison, see this blog post [How to Initialize a String Member - C++ Stories](https://www.cppstories.com/2018/08/init-string-member/)⁵.

The copy and swap idiom

The implementation of `DataPacket` from the second chapter contained two versions of the assignment operator:

```
DataPacket& operator=(const DataPacket& other) { }  
DataPacket& operator=(DataPacket&& other) noexcept {}
```

The code inside those functions contains a bit of code duplication, and what's more, it's not entirely safe. For example, if we change one data member, but the other change throws an exception, our object will be in an invalid state (partially assigned). To improve, we can try writing a single function:

```
DataPacket& operator=(DataPacket other) noexcept {  
    using std::swap;  
    swap(data_, other.data_);  
    swap(checkSum_, other.checkSum_);  
    swap(serverId_, other.serverId_);  
    std::cout << "Assignment for \"" << data_ << "\"\n";  
    return *this;  
}
```

The first striking thing to notice is that the operator takes `DataPacket` by value. Before the operator's body, a fully initialized `DataPacket` object must be passed. In other words, the compiler will call a copy or move constructor for that purpose. Later we can use `swap` to exchange data members. I also wrote `using std::swap` so that the overload resolution can find all related `swap` functions, including those from the `std` namespace. `swap` cannot throw exceptions (as those functions are usually marked with `noexcept`) our assignment operator is safe and won't cause "partially assigned"/invalid objects. Additionally, since we have an object created in the input argument, there's no need to check against `== this`. Let's invoke the new assignment implementation:

⁵<https://www.cppstories.com/2018/08/init-string-member/>

```
DataPacket another { ... };
DataPacket newOne { ... };
another = newOne;
```

In the above use case, the compiler creates a copy of `newOne` and passes it as the `another` argument in the assignment operator.

On the other hand, below, you can see a case where a move constructor will be called to create the other argument before it's passed to the assignment operator:

```
DataPacket another { ... };
DataPacket newOne { ... };
another = std::move(newCar);
```

You can play with this example [@Compiler Explorer⁶](#).

To fully implement the idiom, we could add a `DataPacket::swap` function that could be reused in the copy and the move constructor. For example:

```
DataPacket(DataPacket&& other) noexcept
: DataPacket() // make sure data is initialized
{
    swap(*this, other);
    std::cout << "Move ctor for \"" << data_ << "\"\n";
}

DataPacket& operator=(DataPacket other) noexcept {
    swap(*this, other);
    std::cout << "Assignment for \"" << data_ << "\"\n";
    return *this;
}

friend void swap(DataPacket& a, DataPacket& b) noexcept {
    using std::swap;
    swap(a.data_, b.data_);
    swap(a.checkSum_, b.checkSum_);
    swap(a.serverId_, b.serverId_);
}
```

⁶<https://godbolt.org/z/es0qY97cx>

See this alternative version [@Compiler Explorer](#)⁷.

The move constructor version only works if the member variables are correctly initialized when defined (that's why we have to call a default constructor or use NSDMI described in another chapter). If you use swap with a move constructor and the variables haven't been initialized, then the swap will swap with indeterminate values!

The idea for the idiom is intensely discussed in this Stack Overflow Question: [c++ - What is the copy-and-swap idiom?](#)⁸.

CRTP class counter

In the chapter about inline variables, there was an example called “Instance Counter”. It looks like a handy type that could be used to ** count instances of other types separately. For example, we could inherit from it to share the code. Unfortunately, there's an issue with such a simple approach:

Ex 13.3. The `InstanceCounter` type. Run [@Compiler Explorer](#)

```
class InstanceCounter {
    static inline size_t counter_ { 0 };

public:
    // constructors and destructor...

    static size_t GetInstanceCounter() { return counter_; }
};

struct Value : InstanceCounter {
    int val { 0 };
};

struct Wrapper : InstanceCounter {
    double val { 0.0 };
};

int main() {
    Value v;
    Wrapper w;
```

⁷<https://godbolt.org/z/Me5189Wr8>

⁸<https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom>

```
std::cout << "Values: " << Value::GetInstanceCounter() << '\n';  
std::cout << "Wrappers: " << Wrapper::GetInstanceCounter() << '\n';  
}
```

If you run this code, you'll see the following output:

```
Values: 2  
Wrappers: 2
```

The main trouble is that both classes share the single base class, and thus there's only one "copy" of the `counter_` static data member. We want to count the objects separately and therefore need to have distinct counters. To fix the problem we can use a technique called *Curiously Recurring Template Pattern* (CRTP).

The core idea is to have a class that derives from a class template using itself as a template parameter.

```
template<class Derived>  
class Base {};  
class X : public Base<X> {};
```

Now, each derived class will have a separate "copy" of the Base class, which opens at least two possibilities:

- Add common functionality to derived classes and improves their interface.
- A way to implement static polymorphism. The base class might implement a member function that accesses the Derived class and calls the Derived implementation.

We can implement our counter helper in the following way:

Ex 13.4. The `InstanceCounter` CRTP version. Run [@Compiler Explorer](#)

```
template <typename Derived>
class InstanceCounter {
    static inline size_t counter_ { 0 };

public:
    // constructors and destructor....

    static size_t GetInstanceCounter() { return counter_; }
};

struct Value : InstanceCounter<Value> {
    int val { 0 };
};

struct Wrapper : InstanceCounter<Wrapper> {
    double val { 0.0 };
};

int main() {
    Value v;
    Wrapper w;
    std::cout << "Values: " << Value::GetInstanceCounter() << '\n';
    std::cout << "Wrappers: " << Wrapper::GetInstanceCounter() << '\n';
}
```

Now the output is:

```
Values: 1
Wrappers: 1
```

As you can see, we created two different template instantiations for `InstanceCounter`. There's one for `Value` and the second for `Wrapper`. Now the counters are separate and show the expected values.



Read more about this handy technique in [Curiously Recurring Template Pattern @C++Reference](#)⁹ and also in a three-part series at the Fluent C++ blog: [The Curiously Recurring Template Pattern \(CRTP\), part 1](#)¹⁰.

⁹<https://en.cppreference.com/w/cpp/language/crtp>

¹⁰<https://www.fluentcpp.com/2017/05/12/curiously-recurring-template-pattern/>

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:

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<std::string> adj{ "regular ", "empty ", "blue ", "super " };
    const std::array<std::string> nouns{ "app", "tool", "console", "game" };
    const std::array<std::string> 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 { ads[(r + i) % ads.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 array¹¹](#).

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 RAI 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.

¹¹https://en.cppreference.com/w/cpp/container/array/deduction_guides

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 `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. What's the output of the following code:

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

Select the true statement:

1. C::a is initialized twice. The first time, it's initialized with 10 and then the second time with 0 in the constructor.
2. C::a is initialized only once with 0 in the constructor
3. The code doesn't compile because the compiler cannot decide how to initialize the C::a member.

7. What happens when you throw an exception from a constructor?

1. The object is considered "created" so it will follow the regular lifetime of an object.
2. The object is considered "partially created," and thus, the compiler won't call its destructor.
3. The compiler calls `std::terminate` as you cannot throw exceptions from constructors.

8. What happens when you compile this code?

```
class Point { int x; int y; };  
Point pt { .y = 10, .x = 11 };  
std::cout << pt.x << ", " << pt.y;
```

Select the true statement:

1. The code doesn't compile. Designators have to be in the same order as the data members in the Point class.
2. The code compiles and prints 11, 10.
3. The code compiles and prints 10, 11.

9. Will this code work in C++11?

```
struct User { std::string name = "unknown"; unsigned age { 0 }; };  
User u { "John", 101 };
```

1. Yes, the code compiles in C++11 mode
2. The code compiles starting with C++14 mode
3. The code doesn't compile even in C++20

10. Does the following struct have a compiler-generated copy constructor?

```
struct Test {  
    Test() = default;  
    Test(Test&& t) { /* some implementation*/ }  
    int val { 10 };  
};
```

1. Yes, it's a simple class type so copy constructor will be implicitly defined.
2. No, the class declares a user-defined move constructor which prevents implicit copy constructor.
3. No, the class declares a default constructor which prevents implicit copy constructor.

Write down your answers and check them in Appendix B.

Appendix A - Rules for Special Member Function Generation

In the chapters about constructors and the destructor, we discussed when a compiler implicitly generates a given special member for a class type. In this appendix, you'll see a handy summary of the rules and guidelines for most common use cases.

The diagram

A C++ expert Howard Hinnant, a few years ago created a diagram¹ with the rules:

	Default constructor	Destructor	Copy constructor	Copy assignment	Move constructor	Move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
Default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
Destructor	defaulted	user declared	defaulted*	defaulted*	not declared	not declared
Copy constructor	not declared	defaulted	user declared	defaulted*	not declared	not declared
Copy assignment	defaulted	defaulted	defaulted*	user declared	not declared	not declared
Move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
Move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

¹diagram redrawn, with permission of Howard Hinnant.

Howard is a lead designer and author of the C++11 proposal for move semantics, the `std::chrono` library, and a few other vital parts of Modern C++. The diagram, along with an informative description, can be found on this page: [C++ class declarations](#)² and also see this presentation: [Everything you need to know about move semantics - Howard Hinnant @YouTube](#)³.

How to read the diagram:

Labels:

- defaulted - compiler generates the function if possible.
- defaulted* - deprecated behavior since C++11, the compiler might warn about a function generation.
- not-declared - there's no declaration of a function so that it won't participate in the overload resolution.
- deleted - the function is `=delete`, meaning that it participates in the overload resolution, but it won't be accessible.
- user declared - a given function is declared by the user and not implicitly provided by the compiler. That includes empty implementation, `=default` or even `=delete`.

Rules:

- If a user declares no special member functions, the compiler defaults all special member functions.
- If a user declares any constructor, the compiler defaults all special member functions except for the default constructor
- If a user declares a default constructor, the compiler defaults all special member functions
- If a user declares a destructor, the compiler defaults a default constructor, copy constructor and copy assignment operations. The move constructor and move assignment are not declared. The approach is deprecated, and compilers might warn about such behavior. Usually, when you declare a destructor, there's a high chance the default copy constructor might be insufficient.
- If a user declares a copy constructor, the compiler doesn't declare the default constructor, destructor, and copy assignment is defaulted. The move constructor and move assignment are not declared.

²<https://howardhinnant.github.io/classdecl.html>

³<https://www.youtube.com/watch?v=vLinb2fgkHk>

- If a user declares a copy assignment, the compiler default constructor, destructor, and copy assignment are defaulted. The move constructor and move assignment are not declared.
- If a user declares a move constructor, the compiler doesn't declare the default constructor, and the destructor is defaulted. The move assignment is not declared. The most important part is that the copy constructor and the copy assignment operator are deleted.
- If a user declares a move assignment, the compiler defaults a default constructor and destructor. The move constructor is not declared. The copy constructor and the copy assignment operator are deleted.

More functions provided

In a row, there's only one "user-declared" function, but if your class type has more than one special member function declared, then you have to look at the intersection of the matching rows. For example, suppose you declare a default constructor and a move assignment. In that case, the compiler will provide a default implementation for the destructor but will delete copy operations and not declare the move constructor.

Inheritance

And how about base and derived classes?

- A default constructor for a class T will be defined as deleted if T has a direct or virtual base that has a deleted default constructor, or it is ambiguous or inaccessible from this constructor.
- A copy constructor for a class T will be defined as deleted if T has a direct or virtual base class that cannot be copied (has deleted, inaccessible, or ambiguous copy constructors).
- A move constructor for a class T will be defined as deleted if T has a direct or virtual base class that cannot be moved (has deleted, inaccessible, or ambiguous move constructors).

For example

```
struct Base {
    Base(Base&&) = delete;
};

struct Derived : Base { };

int main() {
    Derived d; // won't compile!
}
```

The above code doesn't compile. We delete the move constructor from the `Base` class. This means that the move constructor in the `Derived` type is also deleted. In both types, the default constructor is not declared and not accessible.

Rules

Rule of zero

In most cases, defining a class without any special member functions will just work:

```
class RuleOfZero {
public:
    // no custom special member functions...

    // member functions...

    // data members...
};
```

In the above case, the `RuleOfZero` class has all special member functions implicitly defined by the compiler.

See the following rule from the C++ Coding Guideline: [C.20: If you can avoid defining default operations, do⁴](#).

⁴<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c20-if-you-can-avoid-defining-default-operations-do>

Rule of three (deprecated!)

Before C++11, when there were no move operations, you could implement all special member functions:

```
class OldRuleOfThree {
public:
    ~OldRuleOfThree();

    OldRuleOfThree(const OldRuleOfThree& other);
    OldRuleOfThree& operator=(const OldRuleOfThree& other);
};
```

However, since C++11, you **shouldn't use this pattern**, as having a copy operation declared won't declare the move operations. The lack of move operations will “slow down” the code that uses those objects, as the compiler will be able to copy data rather than optimize with move.

Rule of 5 and 6

For C++11 and above, if you implement a class that serves as a container or a manager for a resource, then you probably need to implement all special member functions:

```
class RuleOfSix {
public:
    RuleOfSix();
    ~RuleOfSix() noexcept;

    RuleOfSix(const RuleOfSix& other);
    RuleOfSix& operator=(const container& other);

    RuleOfSix(RuleOfSix&& other) noexcept;
    RuleOfSix& operator=(container&& other) noexcept;
};
```

See the following rule from the C++ Coding Guideline: [C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all](#)⁵.

⁵<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c21-if-you-define-or-delete-any-copy-move-or-destructor-function-define-or-delete-them-all>

Moveable only types

```
class MoveableOnly {
public:
    MoveableOnly() noexcept;
    ~MoveableOnly() noexcept;

    MoveableOnly(MoveableOnly&& other) noexcept;
    MoveableOnly& operator=(MoveableOnly&& other) noexcept;
};
```

Note that in the above case, because we declare move operations, then the compiler will delete copy operations:

```
// MoveableOnly(const MoveableOnly&) = delete;
// MoveableOnly& operator=(const MoveableOnly&) = delete;
```

Example types: `std::unique_ptr`.

Polymorphic base classes

```
class BasePoly {
public:
    virtual ~BasePoly() = default;
    BasePoly& operator=(BasePoly&& other) = delete;

    virtual void foo();
}
```

By declaring move assignment, we prevent copy operations (they will be deleted); the move constructor is not declared. But we have to explicitly introduce a `virtual` destructor because, by default, the compiler creates only a non-virtual default destructor.

Appendix B - Quiz and Exercises Answers

See the correct answers

The quiz after Inheriting Constructors

1. 2, as a side note, for classes with a base class you can use inheriting constructors which uses a base class name, rather the derived class name to declare a constructor.
2. 2
3. 2
4. 3
5. 2
6. 2, 3

The final quiz

1. 2
2. 2
3. 1
4. 1
5. 3
6. 2
7. 2
8. 1
9. 2, aggregates and in-class member initialization available since C++14
10. 2, (move constructor prevents implicit copy constructor, see code [@C++ Insight](https://cppinsights.io/s/9a1daa06)⁶

⁶<https://cppinsights.io/s/9a1daa06>

Solution to the first coding problem

Solution to the first coding problem. Run [@Compiler Explorer](#)

```
struct Point {  
    double x { 1.0 };  
    double y { 2.0 };  
};
```

As you can see, the solution uses NSDMI to initialize x and y to required values.

Solution to the second coding problem

Solution to the second coding problem. Run [@Compiler Explorer](#)

```
constexpr unsigned int DEFAULT_CATEGORY = 4;  
constexpr unsigned int DEFAULT_FLAGS = 0x0a;  
  
struct SalesRecord {  
    std::string name_ {"empty"};  
    double price_ { 1.0 };  
    unsigned int category_ : 4 { DEFAULT_CATEGORY };  
    unsigned int flags_ : 4 { DEFAULT_FLAGS };  
};
```

The solution initializes data members to required values including bit fields that are supported since C++20.

Solution to the third coding problem

Solution to the Counted Type problem. Run [@Wandbox](#)

```
struct CountedType {
    static inline int instanceCounter = 0;
    static inline int maxInstanceCounter = 0;

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

    CountedType(const CountedType&) { ++instanceCounter; ++maxInstanceCounter; }
};
```

This solution implements a default constructor, a copy constructor and a destructor. Since we want to know the maximum number of instances, then this variable is not decremented in the destructor.

References

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

Proposals for C++ features:

- [N2756](#)⁷ - Non-static data member initializers for C++11,
- [P0683](#)⁸ - Default Bit Field Initializer for C++20,
- [P0386](#)⁹ - Inline Variables C++17,
- [P0329](#)¹⁰ - Designated Initializers C++20,
- [P0960](#)¹¹ and [P1975](#)¹² - Aggregate initialization from a parenthesized list for C++20.

Valuable resources for C++:

- [C++ Standard Draft](#)¹³ - N4868 (October 2020 pre-virtual-plenary working draft/C++20 plus editorial changes),
- [C++ compiler support - C++Reference](#)¹⁴ - a list of features and their compiler support since C++11,
- [C++ Core Guidelines](#)¹⁵ - a community-edited and open guideline for C++ style, lead by Bjarne Stroustrup and Herb Sutter.

Books:

- [“Embracing Modern C++ Safely”](#)¹⁶ 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”](#)¹⁷ by Scott Meyers

⁷<https://wg21.link/N2756>

⁸<https://wg21.link/P0683>

⁹<https://wg21.link/P0386>

¹⁰<https://wg21.link/P0329>

¹¹<https://wg21.link/p0960>

¹²<https://wg21.link/p1975>

¹³<https://timsong-cpp.github.io/cppwp/n4868/>

¹⁴https://en.cppreference.com/w/cpp/compiler_support

¹⁵<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

¹⁶<https://amzn.to/3PywHTg>

¹⁷<https://amzn.to/3t5tmS4>

Presentations:

- Core C++ 2019: Initialisation in modern C++¹⁸ by Timur Doumler,
- CppCon 2018: “The Nightmare of Initialization in C”¹⁹ by Nicolai Josuttis,
- CppCon 2021: Back To Basics: The Special Member Functions²⁰ by Klaus Iglberger,
- ACCU 2022: What Classes We Design and How²¹ - by Peter Sommerlad,
- CppCon 2018 “The Bits Between the Bits: How We Get to main()²² - by Matt Godbolt

Articles and other links:

- Non-Static Data Members Initialization - C++ Stories²³ - initial source for the book,
- What happens to your static variables at the start of the program? - C++ Stories²⁴,
- Always Almost Auto Style²⁵ by Herb Sutter,
- C++ Core Guidelines - C51²⁶ and C52²⁷ - about delegating and inheriting constructors,
- Modern C++ Features - Inherited and Delegating Constructors²⁸ - by Arne Mertz,
- Trivial, standard-layout, POD, and literal types²⁹ - at Microsoft Docs,
- Modern C++ Features - Uniform Initialization and `initializer_list`³⁰ - by Arne Mertz,
- The cost of `std::initializer_list`³¹ - by Andrzej Krzemiński,
- Objects, their lifetimes and pointers³² - by Dawid Pilarski,
- Tutorial: When to Write Which Special Member³³ by Jonathan Müller.

¹⁸<https://www.youtube.com/watch?v=v0jM4wm1zYA>

¹⁹<https://www.youtube.com/watch?v=7DTIWPgX6zs>

²⁰<https://www.youtube.com/watch?v=9BM5LAvNtus>

²¹<https://www.youtube.com/watch?v=fzsBZicBe88>

²²<https://www.youtube.com/watch?v=dOfucXtyEsU>

²³<https://www.cppstories.com/2015/02/non-static-data-members-initialization/>

²⁴<https://www.cppstories.com/2018/02/staticvars/>

²⁵<https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/>

²⁶<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c51-use-delegating-constructors-to-represent-common-actions-for-all-constructors-of-a-class>

²⁷<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c52-use-inheriting-constructors-to-import-constructors-into-a-derived-class-that-does-not-need-further-explicit-initialization>

²⁸<https://arne-mertz.de/2015/08/new-c-features-inherited-and-delegating-constructors/>

²⁹<https://docs.microsoft.com/en-us/cpp/cpp/trivial-standard-layout-and-pod-types?view=msvc-170>

³⁰https://arne-mertz.de/2015/07/new-c-features-uniform-initialization-and-initializer_list/

³¹https://akrzemi1.wordpress.com/2016/07/07/the-cost-of-stdinitializer_list/

³²<https://blog.panicsoftware.com/objects-their-lifetimes-and-pointers/>

³³<https://www.fooanathan.net/2019/02/special-member-functions/>