C++ STORIES

# Inicialización en C++

Una guía a través de todas las opciones de inicialización y áreas relacionadas con C++

Traducido por Javier Estrada

cppstories.com

Bartłomiej Filipek

# Inicialización en C++

## Una guía a través de todas las opciones de inicialización y áreas relacionadas con C++

Bartłomiej Filipek y Javier Estrada

Este libro está a la venta en http://leanpub.com/cppinitbook_spanish

Esta versión se publicó en 2023-06-24

# También por estos autores

## Libros por Bartłomiej Filipek

C++17 in Detail

C++ Lambda Story

Lambdas en C++

C++ Initialization Story

## Libros por Javier Estrada

Lambdas en C++

# Índice general

# Sobre el libro

¡La inicialización en C++ es un tema candente! El Internet está lleno de debates sobre las mejores prácticas, e incluso hay memes divertidos sobre ese tema. La situación no es sorprendente, ya que hay más de una docena de formas de inicializar un valor entero simple, reglas complejas para la deducción de tipos automáticos, datos miembro y matices del tiempo de vida de objetos.

Y aquí viene el libro.

A lo largo de este texto, aprenderás opciones prácticas para inicializar varias categorías de variables y datos miembro en C++ moderno. Más específicamente, este texto enseña varios tipos de inicialización, constructores, inicialización de datos miembro no estáticos, variables en línea, inicializadores designados y más. Además, verás los cambios y las nuevas técnicas de C++11 a C++20 y muchos ejemplos para completar tu comprensión.

El plan es explicar la mayoría (si no todas) las partes de la inicialización, aprender muchas técnicas excelentes de C++ y ver qué sucede debajo del capó o cofre.

## ¿Por qué debes leer este libro?

Con C++ moderno (desde C++11) tenemos muchas características nuevas para agilizar el trabajo y simplificar nuestro código. Un área de mejora es la inicialización. C++ moderno agregó nuevas reglas de inicialización, tratando de hacerlo más fácil manteniendo el comportamiento y la compatibilidad antiguos (principalmente del lenguaje C). Sin embargo, a veces las reglas pueden parecer confusas y complejas, e incluso el comité de ISO puede necesitar corregir algunas cosas en el camino. El libro te ayudará a navegar a través de esos principios y comprender mejor este tema. Además, la inicialización es solo un aspecto de este texto. Aprenderás todos los temas relacionados con las clases, los constructores, los destructores, el tiempo de vida de los objetos o incluso cómo el compilador procesa los datos al inicio.

## Objetivos de aprendizaje

El objetivo es equiparte con los siguientes conocimientos:

- Explicar las reglas sobre la inicialización de objetos, incluidas las variables regulares, los datos miembro y los objetos no locales.
- Cómo implementar funciones miembro especiales (constructores, destructores, operaciones de copia/movimiento) y cuándo son útiles.
- Cómo inicializar eficientemente datos miembro no estáticos usando funciones de C++11, tales como inicialización de datos miembro no estáticos, constructores herederos y delegadores.
- Cómo agilizar el trabajo con variables estáticas y datos miembro estáticos con variables `inline` de C++17.
- Cómo trabajar con miembros tipo contenedor, datos miembro no copiables (como datos miembro `const`) o datos miembro que solo se pueden mover, o incluso lambdas.
- Qué es un agregado y cómo crear tales objetos con inicializadores designados de C++20.

# Estructura del libro

El libro contiene 14 capítulos con la siguiente estructura:

- Los capítulos 1 a 5 crean una base para el resto del libro. Cubren las reglas básicas de inicialización, los constructores, los destructores y los conceptos básicos de los datos miembro.
- El capítulo 6 trata sobre la deducción de tipos.
- El capítulo 7 es un breve cuestionario sobre constructores. Puedes comprobar tus conocimientos desde la primera "parte" del libro.
- El capítulo 8 describe la inicialización de datos miembro no estáticos (NSDMI por sus siglas en inglés), una potente característica de C++11 que mejora la forma en que trabajamos con datos miembro. Al final del capítulo, puede resolver algunos ejercicios.
- El capítulo 9 analiza cómo inicializar datos miembro similares a contenedores.
- El capítulo 10 contiene información sobre datos miembro no regulares y cómo manejarlos en una clase. Aprenderás sobre `const`, `unique_ptr` como datos miembro y referencias.
- El capítulo 11 describe variables estáticas no locales, objetos estáticos, varias opciones de duración de almacenamiento y variables `inline` de C++17 y `constinit` de C++20.
- El capítulo 12 pasa a C++20 y describe los inicializadores designados, una función útil basada en algo similar del lenguaje C.

- El capítulo 13 muestra varias técnicas, como pasar cadenas a constructores, tipificación fuerte, un contador de clases usando el patrón de plantilla curiosamente recurrente (o CRTP por sus siglas en inglés), el modismo copiar-e-intercambiar y más.
- El capítulo 14 es el cuestionario final con preguntas de todo el libro.

Y hay dos apéndices:

- Apéndice A – una guía útil sobre las reglas para las funciones miembro especiales generadas por el compilador.
- Apéndice B – respuestas a cuestionarios y ejercicios.

# Para quién es este libro

El libro está destinado a programadores de C++ principiantes o intermedios que desean aprender varios aspectos de la inicialización en C++ moderno (de C++11 a C++20).

Debes conocer al menos algunos de los aspectos básicos de la creación y el uso de clases personalizadas.

Este texto también es útil para los programadores experimentados que conocen los estándares de C++ más antiguos y desean pasar a C++17/C++20.

# Requisitos previos

- Deberás tener conocimientos básicos de expresiones C++ y tipos primitivos.
- Deberás ser capaz de implementar una clase elemental con varios datos miembro, así como saber crear y manipular objetos de dicha clase de forma básica.

# Retroalimentación del lector y errata

Si detectas un error, un error tipográfico, un error gramatical o cualquier otra cosa (¡especialmente problemas lógicos!) que deba corregirse, envía tus comentarios a bartek@cppstories.com o somete un asunto en github.com/fenbf/cppinitbook_public/issues[1].

Aquí está la errata con la lista de correcciones:

[1] https://github.com/fenbf/cppinitbook_public/issues

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

¡Tus comentarios son importantes! Si escribes una crítica honesta, puedes ayudar con la promoción del libro y la calidad de mi trabajo posterior.

Además, el libro tiene una página dedicada en GoodReads. Por favor comparte tus comentarios en: C++ Initialization Story by Bartłomiej Filipek[3].

O escribe una reseña en Amazon si obtienes este libro en forma impresa.

# Código de ejemplo

Puedes encontrar el código fuente de todos los ejemplos en este repositorio público independiente de Github.

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

Puedes buscar archivos individuales o descargar toda la rama:

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

## Licencia del código

El código del libro está disponible bajo el modelo de la licencia MIT.

# Formateo y secciones especiales

Los ejemplos de código se presentan en una fuente monoespaciada, similar al siguiente ejemplo:

[2] https://www.cppstories.com/p/cppinitbookspanish/
[3] https://www.goodreads.com/book/show/62606823-c-initialization-story

**Título del ejemplo**

```
#include <iostream>

int main() {
    const std::string text { "Hola, mundo" }
    std::cout << text << '\n';
}
```

O fragmentos más cortos (sin título y a veces con instrucciones `include`):

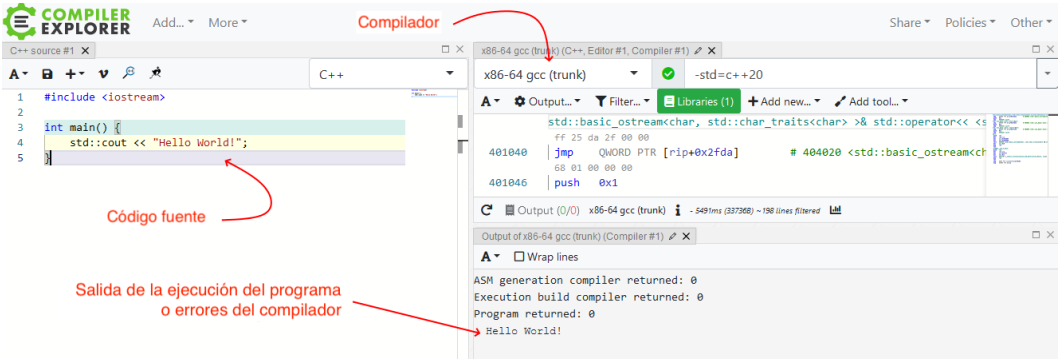
```
int foo() {
    return std::clamp(100, 1000, 1001);
}
```

Cuando esté disponible, también verás un enlace a uno de los compiladores en línea donde puedes jugar con el código. Por ejemplo:

**Título del ejemplo. Ejecutar en Compiler Explorer**

```
#include <iostream>

int main() {
    std::cout << "Hola, mundo!";
}
```

Puedes hacer clic en el enlace en el título y luego deberá abrirse el sitio web de un compilador en línea determinado (en el caso anterior es Compiler Explorer). Puedes compilar la muestra, ver el resultado y experimentar con el código directamente en tu navegador. He aquí una descripción general básica de Compiler Explorer:




**Un diseño de Compiler Explorer utilizado en el libro**

Los fragmentos de programas más largos generalmente se acortaron para presentar solo la mecánica principal. Pueden carecer de algunas declaraciones `#include` o tener una línea "comprimida". Haz clic en el enlace del compilador en línea para ver la versión completa del programa o verlos en el repositorio público.

## Recomendación para Compiler Explorer y Referencia de C++

Al ejecutar los ejemplos en Compiler Explorer[4] con un navegador en español, puedes hacer clic en el botón derecho del ratón (o equivalente) y aparecerá una opción en el menú de contexto, `Search on CppReference`. Si haces clic, te llevará a la versión en español de la Referencia de C++[5] para el término selecctionado (clase, función, contenedor u otros)[^contextmenu].

[^contextmenu] ¡Gracias a Javier Estrada por sugerir este tip genial!

## Limitaciones del resaltado de sintaxis

La versión actual del libro puede mostrar algunas limitaciones con respecto al resaltado de sintaxis.

Por ejemplo:

- El primer método de una clase no está resaltado - Primer método de clase no resaltado en C++ · Asunto #791[6].

[4] https://godbolt.org
[5] https://es.cppreference.com
[6] https://github.com/pygments/pygments/issues/791

- El método de plantilla no está resaltado C++ analizador léxico no reconoce la función si el tipo de retorno tiene una plantilla · Asunto #1138[7].
- Los atributos de C++ moderno a veces no se reconocen correctamente.

Otros asuntos de C++ y Pygments: Asuntos de C++ · github/pygments/pygments[8].

## Secciones especiales

A lo largo del libro también puedes ver las siguientes secciones:

Este es un cuadro de información, con notas adicionales relacionadas con la sección actual.

Este es un cuadro de advertencia con riesgos y amenazas potenciales relacionados con un tema determinado.

> Este es un cuadro de citas. A menudo se usa en el libro para citar el estándar de C++.

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

# Sobre el autor

**Bartłomiej (Bartek) Filipek** es un desarrollador de software C++ de la hermosa ciudad de Cracovia, en el sur de Polonia. Comenzó su carrera profesional en 2007 y en 2010 se graduó de la Universidad Jagiellonian con una Maestría en Ciencias de la Computación.

Bartek trabaja actualmente en Xara[9], donde desarrolla funciones para editores avanzados de documentos. También tiene experiencia con aplicaciones de gráficos de escritorio, desarrollo de juegos, sistemas a gran escala para aviación, escritura de controladores de gráficos e incluso biorretroalimentación. En el pasado, Bartek también ha enseñado programación (principalmente cursos de programación de juegos y gráficos) en las universidades locales de Cracovia.

Desde 2011, Bartek publica regularmente en su blog cppstories.com[10] (comenzó como bfilipek.com[11]). El blog se centra en las características centrales de C++ y en actualizarse con los estándares de C++. También es coorganizador del Grupo de usuarios de C++ en Cracovia[12]. Puedes escuchar a Bartek en un episodio de CppCast[13] donde habla sobre C++17, blogs y procesamiento de texto.

Desde octubre de 2018, Bartek es experto en C++ para el organismo nacional polaco, que trabaja directamente con ISO/IEC JTC 1/SC 22 (el comité de estandarización de C++).

Bartek recibió su primer título de MVP de Microsoft para los años 2019/2020.

En su tiempo libre, le encanta coleccionar y ensamblar modelos de Lego con su hijo.

Bartek es el autor de C++17 In Detail[14] y C++ Lambda Story[15]

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

# Sobre el traductor

**Javier Estrada** es un desarrollador de software de C++ en Silicon Valley en el norte de California. Inició su carrera profesional en 1988 y se graduó del Instituto Tecnológico de Chihuahua[16] con una Ingeniería Industrial en Electrónica.

Javier trabaja actualmente como Ingeniero Principal de Software en Motorola Solutions[17], donde desarrolla software para seguridad pública (9-1-1 emergency) en C++ y Java. Javier también trabajó para VMware[18] en el equipo que produce vRealize Aria Suite (anteriormente vRealize Suite), y para Samsung Semiconductor USA (SSI)[19] en un grupo de desarollo e investigación de almacenamiento de discos de estado sólido (SSD) y su aplicación en sistemas operativos, bases de datos, y aprendizaje de máquinas. En el pasado Javier ha impartido cursos de programación en Python y Java para equipos de robótica de escuelas preparatorias regionales en el sur de California.

Javier publica en su blog Se Habla C++[20], donde trata temas generales y reseñas de presentaciones en CppCon[21] por distintos autores. Puedes ver su perfil profesional en LinkedIn[22].

Javier es el traductor de C++17 - La guía completa[23] y Lambdas en C++[24] y es uno de los editores principales de la Referencia de C++[25]. Puedes escuchar a Javier en pláticas relámpago en CppCon: A Conversion Story: Improving from_chars and to_chars in C++17[26], If You Build It, Will They Come?[27] y C++ en tu idioma[28].

En su tiempo libre, Javier disfruta discutir C++ con su hija, una buena partida de ajedrez, y leer un buen libro.

[16] https://itchihuahua.mx
[17] https://motorolasolutions.com
[18] https://vmware.com
[19] https://semiconductor.samsung.com/us/
[20] https://javierestrada.wordpress.com
[21] https://cppcon.org/
[22] https://linkedin.com/in/ljestrada
[23] https://leanpub.com/cpp17es
[24] https://leanpub.com/cpplambdaspanish
[25] https://es.cppreference.com
[26] https://www.youtube.com/watch?v=7HB4AejLHZs
[27] https://www.youtube.com/watch?v=I8lVKve_bEk
[28] https://youtu.be/n5Uq4J3o7AI

# Agradecimientos

Este libro no sería posible sin el valioso aporte de muchos amigos y expertos en C++.

Me gustaría agradecer especialmente a las siguientes personas:

- JFT (John Taylor),
- Mariusz Jaskółka,
- Florin Chertes (véase su perfil profesional en LinkedIn[29]),
- Konrad Jaśkowiec (véase su perfil profesional en LinkedIn[30]),
- Professor Boguslaw Cyganek (véase su perfil profesional en AGH university page[31]),
- Dawid Pilarski (véase su blog en panicsoftware.com[32]),
- Javier Estrada (véase su perfil profesional en LinkedIn[33] blog en Se Habla C++[34]) y su cuenta Twitter[35],
- Jonathan Boccara (de fluentcpp.com[36]),
- Andreas Fertig (véase su blog en andreasfertig.blog[37]),
- Peter Sommerlad (véase su sitio web e información de capacitación en sommerlad.ch[38]),
- Timur Doumler (véase su sitio web timur.audio[39] y su cuenta Twitter[40]),
- Michael Goldshteyn, Arquitecto de Software.

Pasaron mucho tiempo en encontrar incluso pequeñas cosas que podrían mejorarse y ampliarse.

[29] https://www.linkedin.com/in/florin-ioan-chertes-41b6845/
[30] https://pl.linkedin.com/in/konrad-ja%C5%9Bkowiec-84585159
[31] https://home.agh.edu.pl/~cyganek/
[32] https://blog.panicsoftware.com/
[33] https://www.linkedin.com/in/ljestrada/
[34] https://javierestrada.wordpress.com
[35] https://twitter.com/jestrada
[36] https://www.fluentcpp.com/
[37] https://andreasfertig.blog/
[38] https://sommerlad.ch/
[39] https://timur.audio/
[40] https://twitter.com/timur_audio

Por último, pero no menos importante, recibí mucha retroalimentación valiosa y comentarios de los lectores del blog, Patreon Discord Server (véase C++ Stories en Patreon[41]), y debates en C++ Polska[42]. ¡Gracias a todos!

¡Con toda la ayuda de estas amables personas, la calidad del libro mejoró cada vez más!

[41] https://www.patreon.com/cppstories
[42] https://cpp-polska.pl/

# Historial de revisiones

- 10 de enero de 2023 - ¡La primera versión pública!
- 3 de febrero de 2023 - Capítulo de Inicialización de datos miembro no estáticos (NSDMI), diagramas en español.
- 4 de febrero de 2023 - Capítulo de Contenedores como datos miembro.
- 5 de febrero de 2023 - Referencias al final del libro, corrección de erratas.
- 10 de febrero de 2023 - Capítulo de datos miembro no regulares.
- 14 de febrero de 2023 - Capítulo de objetos no locales.
- 24 de junio de 2023 - Todos los capítulos.

# 1. Local Variables and Simple Types

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

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

We can translate this definition to the following example:

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

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

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

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

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

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

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

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

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

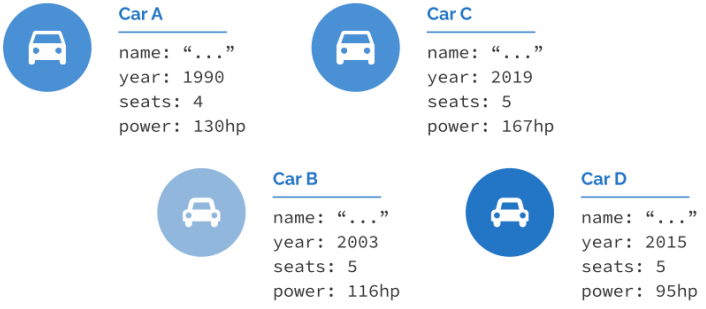
While we can explain most cases on integers and other numerical types, it's best to work on something more practical. The book starts with some elementary custom types, then considers various issues we might have with their early implementations. Later the types will expand, giving us more context and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

# Starting with simple types

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

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

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




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

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

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

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

int main() {
    CarInfo firstCar;
    firstCar.name = "Renault Megane";
    firstCar.year = 2003;
    firstCar.seats = 5;
    firstCar.power = 116;
```

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

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

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

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

```
CarInfo firstCar;
```

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

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

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

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

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

### What is "Automatic Storage Duration" ?

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

## Setting values to zero

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

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

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

The output:

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

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

[3] chapterinlinevars

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

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

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

# Initialization with aggregates

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

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

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

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

In summary, for the above code:

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

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

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

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

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

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

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

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

This will output:

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

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

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

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

## Default data member initialization

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

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

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

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

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

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

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

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

This will print:

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

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

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

# Summary

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

Things to keep in mind:

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

**What's next?**

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

# 2. Classes and Initialization With Constructors

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

## A simple class type

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

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

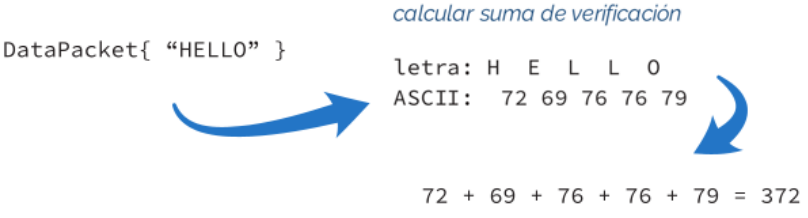
```
#include <iostream>
#include <numeric>

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

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

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

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

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

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

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




**Calculating simple checksum for a string**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This will generate the following output:

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

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

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

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

# Basics of constructors

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

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

A constructor has the following parts:

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

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

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

Let's have a look at one snippet:

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

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

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

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

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

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

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

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

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

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

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

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

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

When I write:

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

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

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

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

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

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

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

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

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

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

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

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

```
Product none;
```

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

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

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

And the last example:

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

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

Here's the complete example:

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

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

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

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

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

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

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

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

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

## More on uniform initialization

*Content available in the full version of the book.*

# Body of a constructor

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

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

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

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

constexpr int LOWEST_ID_VALUE = -100;

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

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

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

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

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

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

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

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

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

## Adding constructors to `DataPacket`

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

**Ex 2.8. Adding constructors. Run @Compiler Explorer**

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

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

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

```
    { }

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

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

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

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

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

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

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

The output:

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

In the above example, we used two constructors:

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

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

# Compiler-generated default constructors

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

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

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

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

Then the compiler will create an implicit empty constructor:

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

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

Have a look:

**Ex 2.10. Implicit default constructor. Run @Compiler Explorer**

```
struct Value {
    int x;
};

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

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

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

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

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

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

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

    int x;
};

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

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

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

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

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

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

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

# Explicit constructors

*Content available in the full version of the book.*

# Difference between direct and copy initialization

*Content available in the full version of the book.*

## Even more

*Content available in the full version of the book.*

# Constructor summary

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

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

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

# 3. Copy and Move Constructors

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

## Copy constructor

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

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

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

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

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

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

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

### A canonical implementation of a copy constructor

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

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

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

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

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

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

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

Here's another example where logging is enabled:

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

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

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

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

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

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

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

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

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

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

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

## A compiler-generated copy constructor

*Content available in the full version of the book.*

# Move constructor

Move constructors take rvalue references of the same type.

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

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

For example:

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

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

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

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

Have a look at this example:

**Ex 3.3. Move Constructor. Run @Compiler Explorer**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

### `noexcept` and move constructors

*Content available in the full version of the book.*

### A compiler-generated move constructor

*Content available in the full version of the book.*

## Distinguishing from assignment

*Content available in the full version of the book.*

## Adding logging to constructors

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

*Content available in the full version of the book.*

# 4. Delegating and Inheriting Constructors

*Content available in the full version of the book.*

## Limitations

*Content available in the full version of the book.*

## Inheritance

*Content available in the full version of the book.*

## Inheriting constructors

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

**Ex 4.4. Inheriting constructors. Run @Compiler Explorer**

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

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

int main() {
```

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

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

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

# 5. Destructors

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

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

## Basics

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

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

And continuing:

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

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

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

**Ex 5.1. A logging destructor. Run @Compiler Explorer**

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

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

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

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

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

The example contains the following special member function:

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

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

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

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

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

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

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

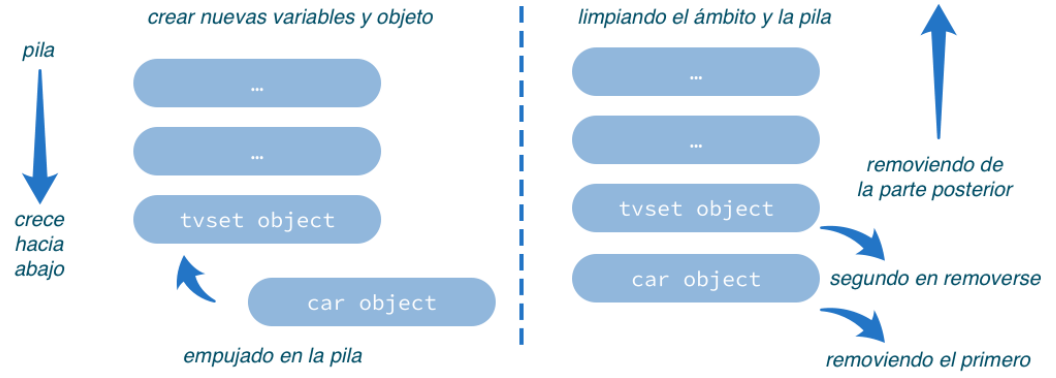
I specifically enclosed objects (created on the stack) in separate scopes so that their lifetime ends when their scope ends. On the other hand, if we have code:

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

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

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

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




**Adding and removing objects from the stack.**

# Objects allocated on the heap

*Content available in the full version of the book.*

# Destructors and data members

*Content available in the full version of the book.*

# Virtual destructors and polymorphism

*Content available in the full version of the book.*

## Partially created objects

*Content available in the full version of the book.*

## Use Cases

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

## A compiler-generated destructor

*Content available in the full version of the book.*

# 6. Initialization and Type Deduction

*Content available in the full version of the book.*

# 7. Quiz on Constructors

Congratulations!

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

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

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

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

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

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

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

*More questions available in the full version of the book.*

# Apéndice A – Cuestionario y respuestas a los ejercicios

*Contenido disponible en la versión completa del libro.*

# Referencias

Materiales y enlaces relacionados sobre la inicialización de datos miembro en C++:

Propuestas para características de C++:

- N2756[1] - Inicializadores de datos miembro no estáticos para C++11,
- P0683[2] - Inicializador por defecto de campo de bits para C++20,
- P0386[3] - Variables en línea para C++17,
- P0329[4] - Inicializadores designados para C++20,
- P0960[5] y P1975[6] - Inicialización de agregados a partir de una lista con paréntesis para C++20.

Recursos valiosos para C++:

- Borrador del estándar de C++[7] - N4868 (octubre de 2020 - borrador de trabajo previo a la plenaria virtual/C++20 más cambios editoriales),
- Apoyo de compiladores de C++ - Referencia de C++[8] - una lista de características y su compatibilidad con el compilador desde C++11,
- Guías Básicas de C++[9] - una guía abierta y editada por la comunidad para el estilo C++, dirigida por Bjarne Stroustrup y Herb Sutter.

Libros:

- "Embracing Modern C++ Safely"[10] por J. Lakos, V. Romeo , R. Khlebnikov, A. Meredith, un libro maravilloso y muy detallado sobre las últimas características de C++, desde C++11 hasta C++14 en la primera edición.

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

- "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14"[11] por Scott Meyers

Presentaciones:

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

Artículos y otros enlaces:

- Non-Static Data Members Initialization - C++ Stories[17] - fuente inicial del libro,
- What happens to your static variables at the start of the program? - C++ Stories[18],
- Always Almost Auto Style[19] por Herb Sutter,
- Guías Básicas de C++ - C51[20] and C52[21] - sobre constructores delegadores y herederos,
- Modern C++ Features - Inherited and Delegating Constructors[22] por Arne Mertz,
- Trivial, standard-layout, POD, and literal types[23] en Microsoft Docs,
- Modern C++ Features - Uniform Initialization and initializer_list[24] por Arne Mertz,
- The cost of `std::initializer_list`[25] por Andrzej Krzemieński,
- Objects, their lifetimes and pointers[26] por Dawid Pilarski,

[11] https://amzn.to/3t5tmS4
[12] https://www.youtube.com/watch?v=v0jM4wm1zYA
[13] https://www.youtube.com/watch?v=7DTlWPgX6zs
[14] https://www.youtube.com/watch?v=9BM5LAvNtus
[15] https://www.youtube.com/watch?v=fzsBZicBe88
[16] https://www.youtube.com/watch?v=dOfucXtyEsU
[17] https://www.cppstories.com/2015/02/non-static-data-members-initialization/
[18] https://www.cppstories.com/2018/02/staticvars/
[19] https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/
[20] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c51-use-delegating-constructors-to-represent-common-actions-for-all-constructors-of-a-class
[21] https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#c52-use-inheriting-constructors-to-import-constructors-into-a-derived-class-that-does-not-need-further-explicit-initialization
[22] https://arne-mertz.de/2015/08/new-c-features-inherited-and-delegating-constructors/
[23] https://docs.microsoft.com/es-mx/cpp/cpp/trivial-standard-layout-and-pod-types?view=msvc-170
[24] https://arne-mertz.de/2015/07/new-c-features-uniform-initialization-and-initializer_list/
[25] https://akrzemi1.wordpress.com/2016/07/07/the-cost-of-stdinitializer_list/
[26] https://blog.panicsoftware.com/objects-their-lifetimes-and-pointers/

- Tutorial: When to Write Which Special Member[27] por Jonathan Müller,
- The implication of const or reference member variables in C++[28] por Lesley Lai.

[27] https://www.foonathan.net/2019/02/special-member-functions/
[28] https://lesleylai.info/en/const-and-reference-member-variables/