POLYTECH
DIJON

2025/2026

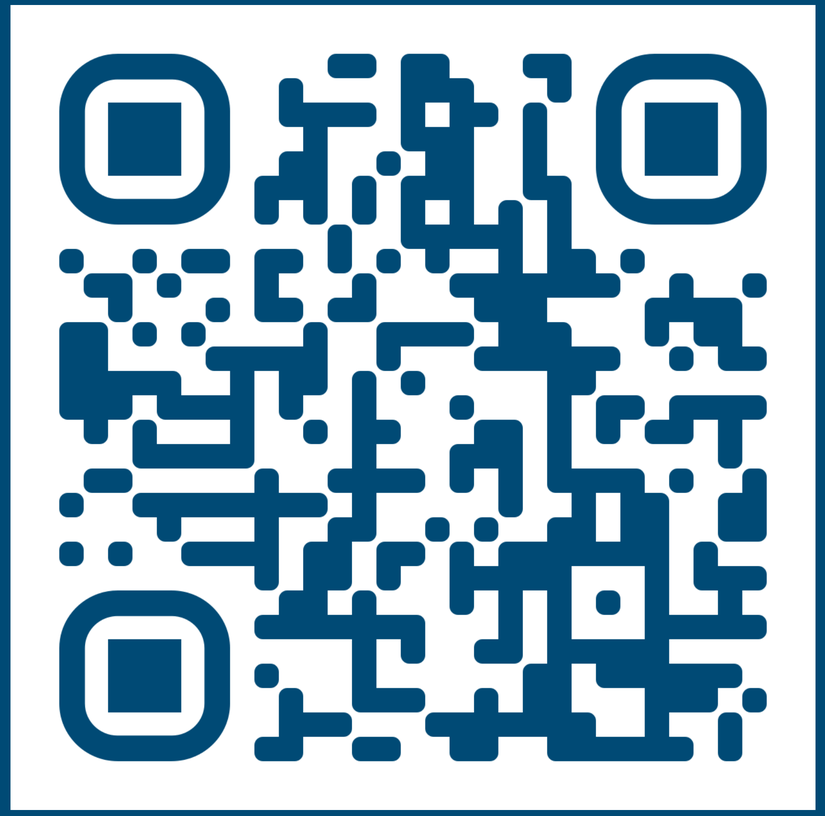
ITC313

Mastering C++:

🎨 The Art of Object-Oriented Programming

Pr. Dominique Ginhac
dginhac@ube.fr

Last updated on October, 21, 2025 by dG



https://ginhac.com/ITC313/01-usertypes.pdf

Lecture #01

https://ginhac.com/ITC313/01-usertypes.pdf

All code samples are available on GitHub in directory "samples/01-usertypes"

User-Defined Data Types

Photo by Alex Padurariu on Unsplash

Learn OOP fundamentals by creating simple, concrete classes and objects

Lecture #01

User-defined Data Types
Abstraction/Encapsulation

Lecture #00
Course Introduction

Today

Lecture #02
Inheritance

Lecture #03
Polymorphism

Lecture #04
STL Containers

Lecture #05
Templates

...

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of a class
3. Value, Reference, Pointer
4. Other user-defined types

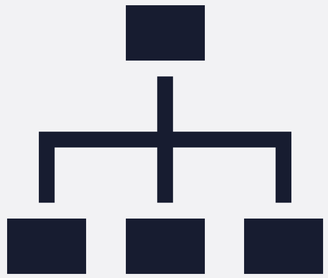
AGENDA

01 - User-defined Data Types

1.Basics of data types

2. A realistic example of a class
3. Value, Reference, Pointer
4. Other user-defined types

C++ Is a Strongly Typed Language



C++ TYPE CLASSIFICATION

Primitive types (built in)

🔤 Numbers (int, float, ...), booleans (bool), single characters (char).

User-defined types

🛠️ Created via libraries or your own programs: Classes, Structures, Enumerations, Unions, References, Pointers.

Strict type checking/safety

🔒 **Fixed types** → Variables must be declared with a type and cannot change it.

⚖️ **Type safety** → Operations are allowed only on compatible data types.

🧩 **Compiler checks** → The compiler enforces type rules, catching errors early.

✨ “Strong typing means more reliable, readable, and maintainable code.”

Reminder on Primitive Data Types

00-basic-types/main.cpp

```cpp
#include <iostream>

int main() {
    // integer type can be short, int, long, and unsigned
    int a_number_not_initialized; // not initialized
    int the_answer = 42; // expression from C-language
    int the_space_odissey(2001); // expression list from C++ Constructor
    int the_number_of_the_beast{666}; // initializer list since C++11
                                      // int n{3.14} -> Compilation error
    // floating point type (simple or double precision)
    float pi = 3.14159, golden_number(1.61803);
    double square_root_of_2 = 1.41421356237;
    // character type
    char a = 'a';
    // boolean : true or false
    bool true_or_false = true;
    // constant can not be modified and must be initialized when declared
    const int dalmatians = 101;

    std::cout << "The number is " << a_number_not_initialized << std::endl;
    std::cout << "The answer is " << the_answer << std::endl;
    std::cout << "The space odyssey is " << the_space_odissey << std::endl;
    std::cout << "The number of the beast is " << the_number_of_the_beast << std::endl;
    std::cout << "The value of pi is " << pi << std::endl;
    std::cout << "The golden number is " << golden_number << std::endl;
    std::cout << "The square root of 2 is " << square_root_of_2 << std::endl;
    std::cout << "The character is " << a << std::endl;
    std::cout << "The boolean is " << true_or_false << std::endl;
    std::cout << "The number of dalmatians is " << dalmatians << std::endl;
    return 0;
}
```

✍️ Always use snake_case for variables

Examples: counter, total_sum.

```
→  00-basic-types ls
app.cpp
→  00-basic-types clang++ app.cpp -o app
→  00-basic-types ls
app     app.cpp
→  00-basic-types ./app
The number is -15466400
The answer is 42
The space odyssey is 2001
The number of the beast is 666
The value of pi is 3.14159
The golden number is 1.61803
The square root of 2 is 1.41421
The character is a
The boolean is 1
The number of dalmatians is 101
```

✨ Always initialize variables.

In modern C++, always prefer {} initialization (or use =)
It prevents nasty surprises.

Automatic Type Inference With Auto

01-auto-types/auto.cpp

```cpp
#include <iostream>

int main() {
    // Typed variables can be defined without initialization
    // but auto variables must be initialized to infer the type
    auto one = 1;
    auto pi = 3.14159;
    auto a = 'A';
    auto valid = true;

    // generate a compilation error
    auto unknown;

    std::cout << "one = " << one << std::endl;
    std::cout << "pi = " << pi << std::endl;
    std::cout << "a = " << a << std::endl;
    std::cout << "valid = " << valid << std::endl;
    return 0;
}
```

```
→  01-auto-types ls
auto.cpp

→  01-auto-types clang++ auto.cpp -o auto
auto.cpp:12:7: error: declaration of variable 'unknown' with deduced type '
auto' requires an initializer
   12 |         auto unknown;
      |              ^
1 error generated.

→  01-auto-types clang++ auto.cpp -o auto

→  01-auto-types ./auto
one = 1
pi = 3.14159
a = A
valid = 1
```

✨ "With auto, the compiler infers the exact type — but only if you initialize. No initialization, no type, no variable."

"Prefer auto when the type is obvious or verbose, but write types explicitly when clarity matters."

A Reminder on Functions

- 🧩 **Definition** → A function is a block of code that performs a specific task.
- 🔁 **Reusability** → Write once, call many times from anywhere in your program.
- 📐 **Organization** → Functions make code clearer, structured, and easier to maintain.
- 📚 **Types of functions** → C++ has user-defined and library functions.
- ✍️ **Naming convention** → Use snake_case for functions to stay consistent with the C++ standard library

02-functions/no-function.cpp

```cpp
#include <iostream>

int main() {
    int a = 3, b = 7, c = 10;
    // Find max between a and b
    int max1;
    if (a > b) max1 = a;
    else max1 = b;
    // Find max between b and c
    int max2;
    if (b > c) max2 = b;
    else max2 = c;

    std::cout << "Max of " << a << " and " << b << " is "
              << max1 << std::endl;
    std::cout << "Max of " << b << " and " << c << " is "
              << max2 << std::endl;
    return 0;
}
```

❌

02-functions/with-function.cpp

```cpp
#include <iostream>

// User-defined function
int max_of_two(int x, int y) {
    if (x > y) return x;
    else return y;
}

int main() {
    int a = 3, b = 7, c = 10;
    int max1 = max_of_two(a, b);
    std::cout << "Max of " << a << " and " << b << " is "
              << max1 << std::endl;
    int max2 = max_of_two(b, c);
    std::cout << "Max of " << b << " and " << c << " is "
              << max2 << std::endl;
    return 0;
}
```

✅

A Reminder on Functions

```
→  02-functions ls
no_function.cpp   with_function.cpp
→  02-functions clang++ no_function.cpp -o no_function
→  02-functions clang++ with_function.cpp -o with_function
→  02-functions ./no_function
Max of 3 and 7 is 7
Max of 7 and 10 is 10
→  02-functions ./with_function
Max of 3 and 7 is 7
Max of 7 and 10 is 10
```

💡 "Functions turn repeated logic into clear, reusable building blocks — making your programs simpler, cleaner, and easier to maintain."

A Reminder on Functions

02-functions/with-function.cpp

```cpp
#include <iostream>

// User-defined function
int max_of_two(int x, int y) {
    if (x > y) return x;
    else return y;
}

int main() {
    int a = 3, b = 7, c = 10;
    int max1 = max_of_two(a, b);
    std::cout << "Max of " << a << " and " << b << " is "
              << max1 << std::endl;
    int max2 = max_of_two(b, c);
    std::cout << "Max of " << b << " and " << c << " is "
              << max2 << std::endl;
    return 0;
}
```

Problem ?

👉 "Right now, max_of_two() lives only inside with-function.cpp — hard-coded and not reusable."

📌 "What if I want to use the same function in another file or program?"

🚫 "Copy-pasting is not reusability."

Solution

✨ "To maximize reuse, we must **separate** a function's **declaration** from its **definition**."

Separate Declaration and Definition

Header file

.h is #included in .cpp

Source file

Interface

Contains declarations/signatures (what exists)

(tells the compiler what the function looks like)

Implementation

Contains definitions (how it works)

(tells the compiler how the function is implemented)

For *max_of_two()*, write *arithmetic.h* and *arithmetic.cpp*, then include *arithmetic.h* in any source file that uses the function.

✨ "Separating declarations and definitions means better modularity, easier reuse, cleaner organization, and faster compilation."

Separate Declaration and Definition

02b-functions/arithmetic.h

```h
// Include guards prevent multiple inclusions:
// they ensure each header is compiled only once
#ifndef ARITHMETIC_H
#define ARITHMETIC_H

// Function prototypes
// only the function signatures (return type, name, parameters)
int max_of_two(int x, int y);
bool is_even(int x);
#endif // ARITHMETIC_
```

02b-functions/arithmetic.cpp

```cpp
#include "arithmetic.h"

// Code of the maxOfTwo function
int max_of_two(int x, int y) {
    if (x > y) return x;
    else return y;
}
// Code of the isEven function
bool is_even(int x) {
    return x % 2 == 0;
}
```

02b-functions/main.cpp

```cpp
#include <iostream>
#include "arithmetic.h"
int main() {
    auto a = 3, b = 7, c = 10;
    auto max1 = max_of_two(a, b);
    std::cout << "Max of " << a << " and " << b << " is "
              << max1 << std::endl;
    auto evenNumber = is_even(c);
    std::cout << c << " is even? "
              << (evenNumber ? "true" : "false") << std::endl;
    return 0;
}
```

Howto

arithmetic.h = **"What exists"** → interface (signatures).

arithmetic.cpp = **"How it works"** → implementation (logic).

main.cpp = **"Use it"** → call the function with values.

Stop Compiling by Hand

```
→  02b-functions ls
arithmetic.cpp arithmetic.h   main.cpp
→  02b-functions clang++ arithmetic.cpp -c
→  02b-functions ls
arithmetic.cpp arithmetic.h   arithmetic.o   main.cpp
→  02b-functions clang++ main.cpp -c
→  02b-functions ls
arithmetic.cpp arithmetic.h   arithmetic.o   main.cpp       main.o
→  02b-functions clang++ arithmetic.o main.o -o app
→  02b-functions ./app
Max of 3 and 7 is 7
10 is even? true
```

🧯 **Error-prone** → Long commands are easy to mistype, and a single missing flag can break your build.

🔁 **No automation** → You must manually decide which files to recompile after every change.

🎲 **Inconsistent builds** → Without automation, each build may use different commands or options, leading to subtle errors.

📦 **Dependency pain** → Tracking which file depends on which header is almost impossible to manage by hand in larger projects.

⏳ **Slow & tedious** → Re-entering the same compilation commands again and again wastes time and focus.

✨ "Manual builds waste time, invite mistakes, and don’t scale — build systems solve all of this."

One-Command Compilation: a Shortcut, Not a Solution

```
→  02b-functions ls
arithmetic.cpp arithmetic.h   main.cpp
→  02b-functions clang++ arithmetic.cpp main.cpp -o app
→  02b-functions ./app
Max of 3 and 7 is 7
10 is even? true
```

✅ **Yes, it works...** → But only for very small projects.

🛠️ **The compiler is still doing separate compilation** → You just don't see the steps (objects + linking).

⏳ **Always recompiles everything** → Even if only one file changed, it forces full recompilation of the whole project every time."

✨ "One-command builds hide the real process, waste time on larger projects, and don't scale — build systems fix this."

Why Automate the Build?

🧯 **Avoid mistakes** → no more typing long commands by hand.

🔁 **Recompile only what changed** → save time.

🎲 **Consistent builds** → same process every time.

📦 **Dependency management** → no more guessing which file to rebuild.

✨ “Automation makes building faster, safer, and scalable.”

"There are many build tools — some integrated directly into IDEs, others available as standalone command-line tools like Make or CMake."

Automating Compilation With Makefiles

In a **Makefile**, you describe:

- 🎯 **What** you want to build (target).
- 🛠️ **How** to build it (commands).
- ⏳ **When** to rebuild it (dependencies).

Then you run *make* in a terminal 💻.

```
→  03-make ls
Makefile        arithmetic.cpp arithmetic.h    main.cpp
→  03-make make
clang++ -c main.cpp
clang++ -c arithmetic.cpp
clang++ main.o arithmetic.o -o app
→  03-make ./app
Max of 3 and 7 is 7
10 is even? true
```

03-make/Makefile

```
# Build target
app: main.o arithmetic.o
    clang++ main.o arithmetic.o -o app

# Rules
main.o: main.cpp arithmetic.h
    clang++ -c main.cpp

arithmetic.o: arithmetic.cpp arithmetic.h
    clang++ -c arithmetic.cpp

# Cleanup
clean:
    rm -f *.o app
```

✨ "One file, one command — and Make handles the rest."

"Make knows which files changed — it rebuilds only what's necessary."

From Hardcoded to Generic Makefiles

03b-make/Makefile

```
# Makefile with automatic dependencies management
# Compile all .cpp files from the current directory
# and link them to the executable into the build directory
# All object files are generated into the build directory
# (c) 2024 by Dom Ginhac (dginhac@ube.fr)
#
# The Makefile must be in the root directory of the project where the source
  files are located.
#
# Usage (in a terminal in the root directory of the project):
# make          # Compile and link all .cpp files
# make clean    # Clean up the build directory
# ./build/app   # Run the executable
#
# ------------------------------------------------------------
# Modify the following lines to fit your project
# SRC_FILES is the list of source files to compile, files are separated by a
space.
SRC_FILES  = main.cpp arithmetic.cpp
#
# APP is the name of the executable, the executable will be generated into the
build directory
APP        = app
#
# Choose your compiler - Use g++ on Linux, Windows and clang++ on Mac OS X
CXX        = clang++
# Compiler options - Wall for all warnings, std=c++17 for C++17
CXXFLAGS   = -Wall
#
# ------------------------------------------------------------
```

📝 **Easy to customize** → just set SRC_FILES, APP, CXX and CXXFLAGS

🔁 **Automatic dependency tracking** → Header changes trigger recompilation.

🧩 **Reusable** → Works for any project without rewriting rules.

⚡ **Efficient** → Recompiles only what changed.

🌐 **Available on GitHub** → Template Makefile ready to download from the course repository.

✨ "Set your variables, type make, and let the magic happen in the build directory."

```
→  03b-make ls
Makefile       arithmetic.cpp arithmetic.h   main.cpp

→  03b-make make
clang++ -Wall  -MMD -c main.cpp -o build/main.o
clang++ -Wall  -MMD -c arithmetic.cpp -o build/arithmetic.o
clang++ -Wall   -o build/app  build/main.o  build/arithmetic.o

→  03b-make ./build/app
Max of 3 and 7 is 7
10 is even? true
```

CMake: a Higher-Level Build Tool

Make is great… but it has limits. It works well for small projects, but as projects grow and need to run on different platforms or IDEs, something more powerful like CMake is needed.

🌍 **Portability** → Makefiles are platform-specific; CMake can generate build files for Linux, macOS, Windows, and many IDEs (Visual Studio, Xcode, etc.).

🧩 **Scalability** → Large projects with hundreds of files are easier to manage with CMake's higher-level syntax.

🔁 **Maintainability** → Instead of hand-writing long Makefiles, you describe your project once in a simple CMakeLists.txt file.

📦 **Industry standard** → Most modern C++ projects (OpenCV, LLVM, Qt, etc.) use CMake.

✨ "CMake doesn't replace Make — it generates Makefiles (or other build files) for you, making projects portable and scalable."

Minimal CMakeLists.txt: 4 Must-Have Lines

The 4 essentials in CMakeLists.txt:

- 📌 CMake version
- 🏷️ Project name
- ⚙️ C++ standard
- 🚀 Executable target

04-cmake/CMakeLists.txt

```
1  # 1. Specify the minimum CMake version
2  cmake_minimum_required(VERSION 3.15)
3
4  # 2. Define the project name
5  project(app)
6
7  # 3. Set the C++ standard
8  set(CMAKE_CXX_STANDARD 20)
9  set(CMAKE_CXX_STANDARD_REQUIRED True)
10
11 # 4. Define the executable target (name + source files)
12 add_executable(app main.cpp arithmetic.cpp)
```

```
→  04-cmake ls
CMakeLists.txt arithmetic.cpp arithmetic.h   main.cpp

→  04-cmake cmake -B build
-- The C compiler identification is AppleClang 17.0.0.17000319
-- The CXX compiler identification is AppleClang 17.0.0.17000319
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.5s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/d0m/Documents/teaching/
polytech/3A/ITC313/2025-2026/samples/01-usertypes/04-cmake/build

→  04-cmake make -C build
[ 33%] Building CXX object CMakeFiles/app.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/app.dir/arithmetic.cpp.o
[100%] Linking CXX executable app
[100%] Built target app

→  04-cmake ./build/app
Max of 3 and 7 is 7
10 is even? true
```

✨ "With these 4 lines, you can already configure, build, and run your C++ project anywhere."

A Generic CMakeLists.txt

04b-cmake/CMakeLists.txt

```
# The CMakeLists.txt file must be in the root directory of the project where the
source files are located
# Usage (in a terminal in the root directory of the project):
# cmake -S . -B build   # generate the build files in the build directory
# cmake --build build   # compile the project
# ./bin/app       # run the executable
#
# set the minimum required version of cmake
cmake_minimum_required(VERSION 3.15)
# ----------------------------------------------------------
# Modify the following lines to fit your project
# set the project name (default is app), version, description and language (C++ is
CXX)
project(app
        VERSION 0.0.1
        DESCRIPTION "A first example project with CMAKE and C++"
        LANGUAGES CXX)
#
# SRC_FILES is the list of source files to compile, files are separated by a space.
set(SOURCE_FILES arithmetic.cpp main.cpp)
#
# set the name of the executable
set(TARGET app)
#
# set the C++ standard to C++17
set(CMAKE_CXX_STANDARD 17)
# require the C++ standard
set(CMAKE_CXX_STANDARD_REQUIRED ON)
#
add_executable(${TARGET} ${SOURCE_FILES})
```

📝 **Easy to customize** → just set SOURCE_FILES, TARGET, CMAKE_CXX_STANDARD and app.

🌐 **Available on GitHub** → Template CMakeLists.txt ready to download from the course repository.

✨ "Set your variables, create the build directory, type cmake .., then make, and let the magic happen in the build directory."

```
➜  04b-cmake ls
CMakeLists.txt arithmetic.cpp arithmetic.h   main.cpp

➜  04b-cmake cmake -B build
-- The CXX compiler identification is AppleClang 17.0.0.17000319
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /Users/d0m/Documents/teaching/polytech/
3A/ITC313/2025-2026/samples/01-usertypes/04b-cmake/build

➜  04b-cmake make -C build
[ 33%] Building CXX object CMakeFiles/app.dir/arithmetic.cpp.o
[ 66%] Building CXX object CMakeFiles/app.dir/main.cpp.o
[100%] Linking CXX executable app
[100%] Built target app

➜  04b-cmake ./build/app
Max of 3 and 7 is 7
10 is even? true
```

Make vs CMake: Conclusion

Both Make and CMake are build automation tools — but they serve different needs.

✅ When to use Make

- 📝 **Good for learning** → Understand compilation steps.
- 🎯 **Small projects** → Just a few .cpp files.
- ⚡ **Direct control** → You write explicit rules.

✅ When to use CMake

- 🌍 **Cross-platform** → Generates Makefiles, Ninja, Visual Studio, Xcode...
- 🧩 **Medium to large projects** → Scales better.
- 📦 **Industry standard** → Used in modern C++ libraries and applications.

✨ "Use Make to learn the mechanics — use CMake to build real-world projects."

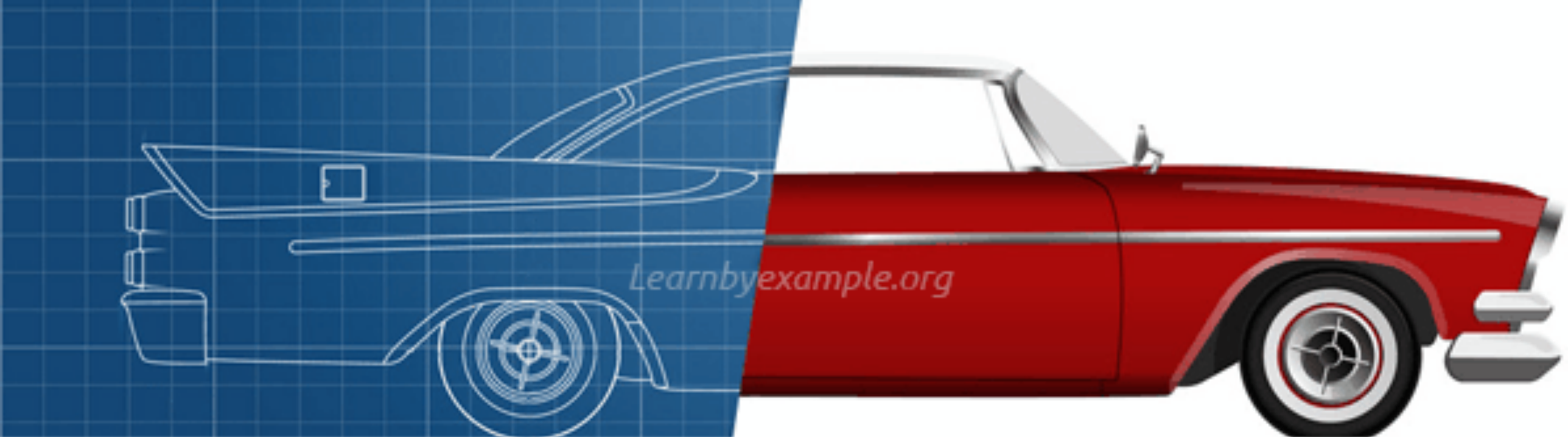


Photo by LearnByExample

User-defined types

To solve complex problems, we need precise models of the real world.

The closer our code reflects reality, the easier it becomes to design and maintain programs.

🚀 In C++, the key tools for building such models are **classes** and **objects**.

What Is a Class? What Is an Object?

🏗️ **Class** → A blueprint or model that defines a new type of object.

It groups together Attributes (variables → what it is/has) and Methods (functions → what it can do = its behavior)

Just a model — it defines what attributes and behaviors objects will have, but it does not exist in memory.

🏠 **Object** → An instance of a class.

Each object has its own values for the attributes and can perform the behaviors defined by the class.

A concrete instance of a class — it exists in memory, with its own copy of attributes. Two objects of the same class are independent, each with their own state.

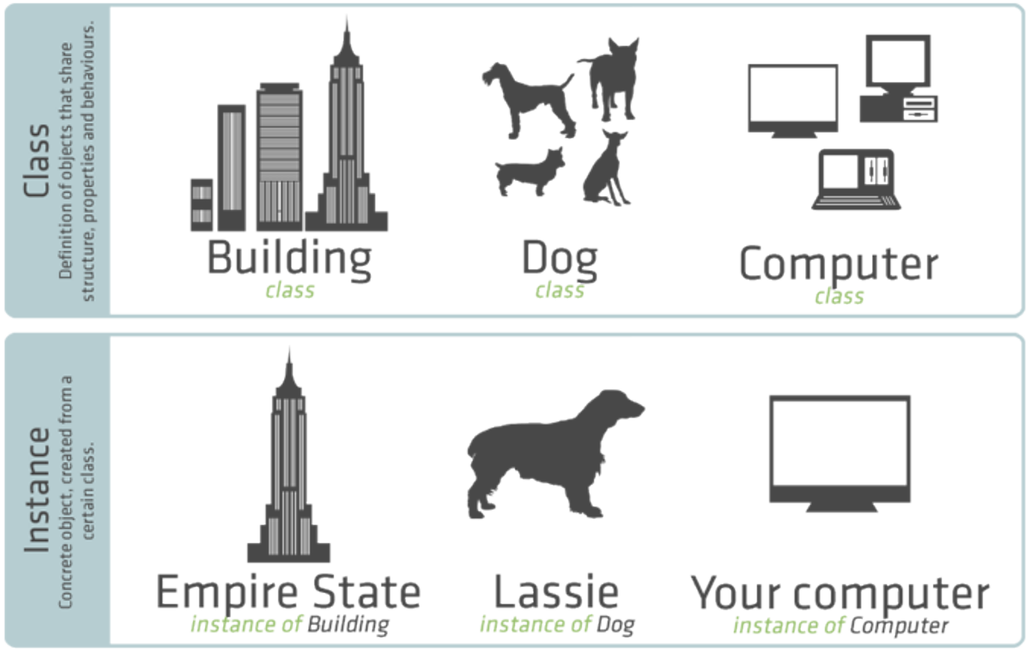
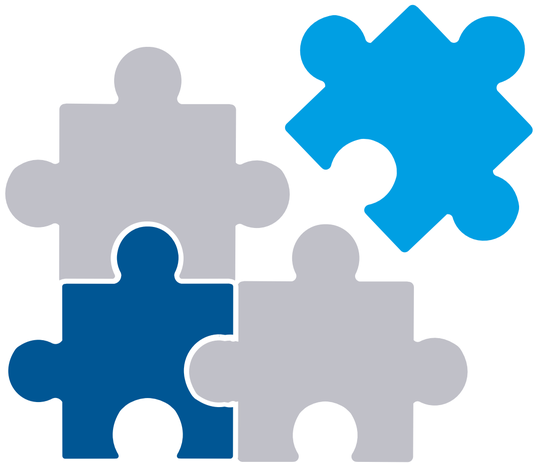


Illustration by Sencha

✨ "A class is the plan — an object is the real thing."

"A class is only the recipe; an object is the actual cake, baked with its own ingredients." 🍰

The Four Pillars of OOP



🏠 **Encapsulation** → Bundle related data and the methods that operate on it into a single, coherent unit — a class.

Ex: a BankAccount has a balance (data) and methods like deposit() or withdraw().

🎭 **Abstraction** → Expose only the essential features, hide the unnecessary details.

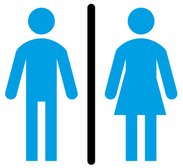
Ex: a CoffeeMachine offers brewEspresso() or brewLatte() methods — you don't need to know how water pressure or heating is managed internally.

👨‍👩‍👦 **Inheritance** → Create new classes from existing ones — reuse attributes and behaviors, then extend them.

Ex: an Ebook inherits from a Book, reusing attributes like title and author and adding specific features like url or download().

🔀 **Polymorphism** → One interface (the same function), many forms (depending on the object).

Ex: a Shape offers a draw() method — implemented differently in Circle, Square, and Triangle.



A First Basic Example

Let’s create a class for a person that includes the following information:

- ☑ firstname
- ☑ lastname
- ☑ gender

And constructs the full identity of the person (“Mr Dom Ginhac”).

Photo by Ben Sweet on Unsplash

Separate Declaration and Definition of the Class

Header file

.h is #included in .cpp

Source file

Interface

Contains declarations/signatures (what exists)

(tells the compiler what the function looks like)

person.h

Implementation

Contains definitions (how it works)

(tells the compiler how the function is implemented)

person.cpp

✨ "When you define a new class Person, you write person.h and person.cpp and you include person.h in each source file that uses Person objects."

Declaring a Class (person.h)

🧩 Syntax

- The keyword class introduces the declaration.
- Class names use PascalCase (ex: Person, MyExcitingClass).
- Contents of the class are enclosed in { } and end with ;.
- A class may contain several access sections: public, private, or protected. By convention, the public section comes first, then the private section, ...
- The sections includes Member variables and Methods.

📦 Member Variables

- Member variables are generally private.
- Named with snake_case (lowercase + underscores) with trailing underscore (older conventions sometimes use m_ or _ as a prefix).

⚙️ Methods

- Typically declared public.
- Named with camelCase (e.g., getFullName) ≠ Functions named with snake-case
- Only declared in the header (inputs/outputs), defined elsewhere.
- The special method Person() is the constructor — it initializes member variables when an object is created.

05-Person/person.h

```h
#ifndef PERSON_H
#define PERSON_H

#include <string>

class Person {
public:
  Person(std::string firstname,
         std::string lastname,
         int gender);
  std::string getFullName();

private:
  std::string firstname_;
  std::string lastname_;
  int gender_;
};

#endif // PERSON_H
```

Definition of a Class (person.cpp)

05-Person/person.cpp

```cpp
#include "person.h"

Person::Person(std::string firstname,
               std::string lastname, int gender) {
  firstname_ = firstname;
  lastname_ = lastname;
  gender_ = gender;
}

std::string Person::getFullName() {
  if (gender_ == 1) {
    return "Mr " + firstname_ + " " + lastname_;
  }
  if (gender_ == 2) {
    return "Ms " + firstname_ + " " + lastname_;
  }
  // Non-binary
  return firstname_ + " " + lastname_;
}
```

🧩 Syntax

📄 **Include the header** → "person.h" to use the class declaration.

🏷️ **Fully qualified name** → Person::getFullName specifies that the method belongs to the class Person.

🎯 **Direct access** to member variables inside methods (no special syntax needed).

🔙 **Return statement** outputs the result of a method (similar to functions).

🚀 **Constructor** initializes member variables with the given parameters.

✨ "Declaration in the header, definition in the .cpp — keep code organized and reusable."

Testing the Person Class

05-Person/main.cpp

```cpp
#include <iostream>
#include "person.h"

int main() {
  // Direct Initialization -- The classic, simple way.
  Person dg("Dom", "Ginhac", 1);
  std::string fullname = dg.getFullName();
  std::cout << "Hello " << fullname << std::endl;

  // Copy Initialization -- The = sign is used.
  auto taylor = Person("Taylor", "Swift", 2);
  std::cout << "Hello " << taylor.getFullName() << std::endl;

  // List Initialization -- The modern, unified way.
  auto miley = Person{"Miley", "Cyrus", 3};
  std::cout << "Hello " << miley.getFullName() << std::endl;

  std::cout << "That's all folks" << std::endl;
  return 0;
}
```

🧩 Syntax

📄 **Include the header** → #include "person.h".

👤 **Create objects** →
Person dg("Dom", "Ginhac", 1);
auto taylor = Person("Taylor", "Swift", 2);
auto miley = Person{"Miley", "Cyrus", 3};

⚙️ **Call methods** → object.method(params) (e.g., dg.getFullName();).

🛠️Compilation

⚡ Use make or CMake to compile all .cpp files.

```
→  05-Person ls
Makefile   main.cpp   person.cpp person.h

→  05-Person make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c person.cpp -o build/person.o
clang++ -Wall  -o build/app  build/main.o  build/person.o

→  05-Person ./build/app
Hello Mr Dom Ginhac
Hello Ms Taylor Swift
Hello Miley Cyrus
That's all folks
```

Questions

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of a class
3. Value, Reference, Pointer
4. Other user-defined types

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of
3. Value, Reference, Pointer
4. Other user-defined types

A First Realistic Example of Class

Suppose we need 📅 dates for an application.

What is the star wars day? May, 4 (i.e 05/04)

What is my birthday? May, 26 (i.e 05/26)

06-Time/main.cpp

```cpp
#include <ctime>
#include <iostream>

int main()
{
    std::time_t result = std::time(nullptr);
    std::cout << std::asctime(std::localtime(&result))
              << result << " seconds since the Epoch\n";
    return 0;
}
```

```
→  06-Time ls
Makefile main.cpp

→  06-Time make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall  -o build/app  build/main.o

→  06-Time ./build/app
Wed Sep 24 09:47:39 2025
1758700059 seconds since the Epoch
```

⚠️ C++ does not provide Date objects, only time-related types in ctime library that are not easy to use.

✨ "The only solution is to create our own Date class."

Some Principles To Design a New Class

A class should provide a clear, simple, and consistent representation within your system.

🪶 The **simplest design** that works is usually the best.

👉 *Favor clarity over cleverness.*

🔁 Duplicated code **multiplies bugs**.

👉 *Keep each piece of logic in one single place.*

⏳ Don't **implement features** "just in case".

👉 *Add only what you really need, when you need it.*

✨ Simplicity, consistency, and focus are the foundations of good class design.

Designing the Date Class

📦 Variables

We need two variables month and day represented as integers.

```
int month_; // Use snake_case for naming variables
int day_;   // Use “_” suffix for variables
```

🚀 Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

⚙️ Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // methods with explicit names
```

Photo by Charles Deluvio on Unsplash

Designing the Date Class

📦 Variables

We need two variables month and day represented as integers.

```
int month_; // Use snake_case for naming variables
int day_;   // Use “_” suffix for variables
```

🚀 Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

⚙️ Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // methods with explicit names
```

Photo by Charles Deluvio on Unsplash

Variables

Public

accessible everywhere

Forms the open interface of the class.

Attributes are directly accessible from outside.

⚠️ Danger: too much exposure breaks consistency.

Ex: In a Date class, someone could set month = 42.

Ex: In a BankAccount, anyone could change the balance.

vs

Members are not directly accessible from outside.

Access only through public methods.

Typically managed with Constructor (Initialize), Getters (Read) and Setters (Update with checks).

Encapsulation protects integrity by hiding the internal representation.

hidden from the outside

Private

✨ "Private variables are a tool of encapsulation, and they help us achieve abstraction."

Always hide the internal representation of a class with

Private

Variables

Designing the Date Class

📦 Variables

We need two variables month and day represented as integers.

```
int month_; // Use snake_case for naming variables
int day_;   // Use "_" suffix for variables
```

🚀 Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

⚙️ Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // methods with explicit names
```

Photo by Charles Deluvio on Unsplash

Constructor in C++

🔧 Special class function

- Constructors are special methods of a class.
- No return type for constructors (not even void).
- Constructors allocate storage and perform initialization of each new object.

✍️ How to create a constructor?

- Constructors have the same name as the class itself and can take arguments that are used to initialize the member variables.
- A class can have several constructors with different parameters.

🚀 How to use a constructor?

- Constructors are automatically called when a new object is created.
- No need to explicitly call them.

Photo by Silvia Brazzoduro on Unsplash

✨ "Constructors make sure every object starts life properly initialized."

3 Main Types of Constructors

Default — Minimal

- The most basic constructor.
- No input parameter.
- Ensures an object can always be created.

```
Date d;
```

Parametrized

- Constructor with arguments.
- Initializes attributes with given values.
- Allows customized object creation.

```
Date pi_day(3,14);
```

Copy

- Creates a new object from an existing one.
- Copies all member variables.
- Essential for passing/returning objects.

```
Date another_pi_day = pi_day;
```

What Happens if You Provide no Constructor at All?

🛠️ Don’t worry, C++ will create one for you… but not always the way you expect!

⚠️ The attributes may contain undefined (garbage) values → leading to invalid objects.

07-Date-no-constructor/main.cpp

```cpp
#include <iostream>
#include "date.h"

int main() {
    Date d1;
    std::cout << "d1: " << d1.month() << "/" << d1.day() << std::endl;
    Date d2 = Date();
    std::cout << "d2: " << d2.month() << "/" << d2.day() << std::endl;
    Date d3{};
    std::cout << "d3: " << d3.month() << "/" << d3.day() << std::endl;
    return 0;
}
```

```
→  07-Date-no-constructor make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o

→  07-Date-no-constructor ./build/app
d1: -86766160/1
d2: 0/0
d3: 0/0
```

month() and day() are getter methods allowing to read the values stored into the month_ and day_ private variables.

More details later.

✨ "Don’t rely on C++ defaults — always provide constructors to keep your objects consistent and valid."

Minimal Constructor

🛠️ A minimal constructor creates objects with fixed default values defined in the constructor.

08-Date-minimal-constructor/date.h

```h
class Date {
public:
    Date();
    int month();
    int day();
private:
    int month_;
    int day_;
};
```

08-Date-minimal-constructor/date.cpp

```cpp
Date::Date() {
    month_ = 1;
    day_ = 1;
}
```

```
→  08-Date-minimal-constructor make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
```

```
→  08-Date-minimal-constructor ./build/app
d1: 1/1
```

08-Date-minimal-constructor/main.cpp

```cpp
int main() {
    Date d1;
    std::cout << "d1: " << d1.month() << "/" << d1.day() << std::endl;
    return 0;
}
```

Parameterized Constructor

🎛️ "With a parameterized constructor, each object starts with the exact values you pass in."

✨ "Parameterized constructors customize each object at creation time."

09-Date-parameterized-constructor/date.h

```h
class Date {
public:
    Date();
    Date(int month, int day);
    int month();
    int day();
private:
    int month_;
    int day_;
};
```

09-Date-parameterized-constructor/date.cpp

```cpp
Date::Date() {
    month_ = 1;
    day_ = 1;
}

Date::Date(int month, int day) {
    month_ = month;
    day_ = day;
}
```

Parameterized Constructor

09-Date-parameterized-constructor/main.cpp

```
1  #include <iostream>
2  #include "date.h"
3
4  int main() {
5    Date d;
6    std::cout << "Default: " << d.month() << "/"
                 << d.day() << std::endl;
7    Date pi_day(3,14);
8    std::cout << "Pi day: " << pi_day.month() << "/"
                 << pi_day.day() << std::endl;
9    Date star_wars_day(5,4);
10   std::cout << "Star wars day: " << star_wars_day.month() << "/"
                 << star_wars_day.day() << std::endl;
11   return 0;
12 }
```

```
→  09-Date-parameterized-constructor make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
```

```
→  09-Date-parameterized-constructor ./build/app
Default: 1/1
Pi day: 3/14
Star wars day: 5/4
```

Parameterized Constructor

⚠️ Important rule

If you define a parameterized constructor, the compiler will not generate a default constructor.

❌ Example:

```
1 int main() {
2   Date d;
```

```
➜  10-Date-no-default-constructor make
clang++ -Wall -MMD -c main.cpp -o build/main.o
main.cpp:5:8: error: no matching constructor for initialization of 'Date'
    5 |   Date d;
      |        ^
./date.h:4:7: note: candidate constructor (the implicit copy constructor) not viable: requires 1 argument, but 0 were provided
    4 | class Date {
      |       ^~~~
./date.h:4:7: note: candidate constructor (the implicit move constructor) not viable: requires 1 argument, but 0 were provided
    4 | class Date {
      |       ^~~~
./date.h:7:4: note: candidate constructor not viable: requires 2 arguments, but 0 were provided
    7 |    Date(int month, int day);
      |    ^    ~~~~~~~~~~~~~~~~~~
1 error generated.
make: *** [build/main.o] Error 1
```

✅ Solution:

Use the = default; specifier (since C++11) to force the compiler to generate it.

The compiler's default constructor does no initialization.

```
 1 class Date {
 2 public:
 3     Date() = default;
 4     Date(int month, int day);
 5     int month();
 6     int day();
 7 private:
 8     int month_;
 9     int day_;
10 };
```

Default Parameters in Constructors

🌀 Writing both default and parameterized constructors is often **redundant** (violates DRY).

➕ They are just **overloaded versions** of the same constructor.

🎯 Use **default values** in the constructor → only one constructor needed.

📄 Default values go in the **declaration** (.h) — the definition (.cpp) stays unchanged.

11-Date-default-parameters/date.h

```h
class Date {
public:
    Date(int month=1, int day=1);
    int month();
    int day();
private:
    int month_;
    int day_;
};
```

11-Date-default-parameters/date.cpp

```cpp
#include "date.h"

Date::Date(int month, int day) {
    month_ = month;
    day_ = day;
}
```

✨ “Default parameters collapse multiple constructors into a single, clean definition.”

Attributes Initialization

By default, attributes are often initialized with the assignment operator (=) inside the constructor body.

✅ Works for many types.

❌ But does not work for: const attributes, references (&) and some other more complex types.

12-Person-InitialiserLists/person.h

```h
#include <string>
#include "date.h"

class Person {
    public:
        Person(std::string firstname,
               std::string lastname,
               const Date birthday);
    private:
        std::string firstname_;
        std::string lastname_;
        const Date birthday_;
};
```

12-Person-InitialiserLists/person.cpp

```cpp
Person::Person(std::string firstname,
               std::string lastname,
               const Date birthday) {
    firstname_ = firstname;
    lastname_ = lastname;
    // This would cause a compilation error
    birthday_ = birthday;
}
```

```
→  12-Initializer make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c person.cpp -o build/person.o
person.cpp:10:15: error: no viable overloaded '='
   10 |     birthday_ = birthday;
      |     ~~~~~~~~~ ^ ~~~~~~~~
./date.h:4:7: note: candidate function (the implicit copy assignment operator) not
 viable: 'this' argument has type 'const Date', but method is not marked const
```

✨ "Assignment happens too late (after the creation of the attributes)."

Member Initialiser Lists

C++ provides “Member initialiser list” starting with a colon after the list of parameters.

- ✅ **Concise** — Initialization happens directly.
- ⚡ **Efficient** — Avoids unnecessary default construction + reassignment.
- 🧩 **Universal** — Works with any user-defined type, Required for const, references, …

12-Person-InitialiserLists/person.cpp

```cpp
#include "person.h"

Person::Person(std::string firstname, std::string lastname, Date birthday) :
    firstname_(firstname), lastname_(lastname), birthday_(birthday) {
    // Nothing to do
}
```

✨ “Members are always initialized in the order they are declared in the class — not in the order of constructor parameters or the initializer list.”

Wrong ordering can cause unexpected behavior.

Copy Constructor

Photo by Sharon McCutcheon on Unsplash

Used to declare and initialise an object from another object in the following 3 cases:

🏗️ Object constructed from another object

```
1 Date starwars(5,4);
2 Date other_starwars_day = starwars;
3 Date another_starwars(starwars);
```

🔙 Object returned by a function

```
1 Date tomorrow = today.nextDay();
```

📤 Object passed to a function

```
1 Date pi_day(3,14);
2 bool b = starwars.before(pi_day);
```

Copy Constructor — Default vs Custom

Do we really need a Copy constructor?

❌ No (most of the time): the compiler automatically provides a default copy constructor, performing a memberwise copy.

⚠️ Yes (sometimes): you need to write your own only when your class manages resources like dynamic memory, pointers, or file handles

13-Date-copy-constructor/date.h

```h
class Date {
public:
    Date(int month=1, int day=1);
    // copy constructor
    Date(const Date& other);
    int month();
    int day();
private:
    int month_;
    int day_;
};
```

13-Date-copy-constructor/date.cpp

```cpp
#include "date.h"

Date::Date(const Date& other) : month_(other.month_),
                                day_(other.day_) {
    // Nothing to do
}
```

✨ “In C++, the default copy constructor is usually enough — unless your class owns resources that need special handling.”

How To Create A Valid Date Object

?

Photo by Sigmund on Unsplash

Ensure Object Validity at Construction

14-Date-create-valid-object/date.h

```h
class Date {
public:
    Date(int month=1, int day=1);
    int month();
    int day();
private:
    int month_;
    int day_;
    bool isDate(int month, int day);
};
```

14-Date-create-valid-object/date.cpp

```cpp
bool Date::isDate(int month, int day) {
    if ((day < 1) || (day>31)) return false;
    if ((month <1) || (month>12)) return false;
    if ((month == 2) && (day > 28)) return false;
    if (((month == 4) || (month == 6) || (month == 9)
        || (month == 11)) && (day > 30)) return false;
    return true;
}

Date::Date(int month, int day) : month_(month), day_(day) {
    if (!isDate(month, day)) {
        throw std::invalid_argument("Invalid date: "
            + std::to_string(month) + "/"
            + std::to_string(day));
    }
}
```

|| = OR

&& = AND

🔒 **isDate is private** — only used internally, not exposed outside.

🏗️ **Called inside the constructor** to validate object creation.

⚠️ Constructor uses **exception** to immediately stop execution if the date is invalid.

🎯 Every Date object used in the app is guaranteed to be **safe/valid** by design.

✨ “Constructor’s job is not just to build — it’s to guarantee validity!”

Without Try/Catch — Crash on Error

14-Date-create-valid-object/main.cpp

```cpp
#include <iostream>
#include "date.h"

int main() {
    Date pi_day_ok(3,14);  // valid date
    std::cout << "ok:" << pi_day_ok.month() << "/" << pi_day_ok.day() << std::endl;
    Date pi_day_error(14,3); // invalid date
    std::cout << "nok: " << pi_day_error.month() << "/" << pi_day_error.day() << std::endl;
    return 0;
}
```

⚠️ Uncaught exception **stops the program** immediately — no chance to recover.

🛑 **No error handling** — the program just crashes.

🔍 **Useful during debugging** — makes bugs visible right away.

🚫 **Not production-ready** — never rely on uncaught exceptions in real applications.

```
→  14-Date-create-valid-object ls
Makefile date.cpp date.h   main.cpp

→  14-Date-create-valid-object make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o

→  14-Date-create-valid-object ./build/app
ok:3/14
libc++abi: terminating due to uncaught exception of type std::invalid_argument:
Invalid date: 14/3
[1]    60483 abort      ./build/app
```

With Try/Catch — Handle Errors Safely

14-Date-create-valid-object/main.cpp

```cpp
int main() {
    try {
        Date pi_day_ok(3,14);  // Valid date
        std::cout << "ok:" << pi_day_ok.month() << "/" << pi_day_ok.day() << std::endl;
        Date pi_day_error(14,3); // Invalid date
        std::cout << "nok: " << pi_day_error.month() << "/" << pi_day_error.day() << std::endl;
    } catch (const std::invalid_argument& e) {
        std::cerr << "Error creating Date: " << e.what() << std::endl;
    }
return 0;
}
```

🛡️ **Catches exceptions** instead of crashing — the program stays alive.

🎯 **Handles errors gracefully** — you can log, display a message, or even fix the issue.

🔄 **Keeps control flow** — the program can continue running after the error.

🚀 **Production-ready** — error handling is crucial at data input. Once validated, data can be used safely throughout the program.

```
→  14-Date-create-valid-object ls
Makefile date.cpp date.h   main.cpp
→  14-Date-create-valid-object make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
→  14-Date-create-valid-object ./build/app
ok:3/14
Error creating Date: Invalid date: 14/3
```

Constructors — Key Takeaways

✅ **Guarantee validity** — Design your types so every object is valid at creation.

🏗️ **Constructors enforce rules** — They ensure only valid objects are created, rejecting invalid input.

⚡ **Prefer initializer lists** — Concise, efficient, and required for some members.

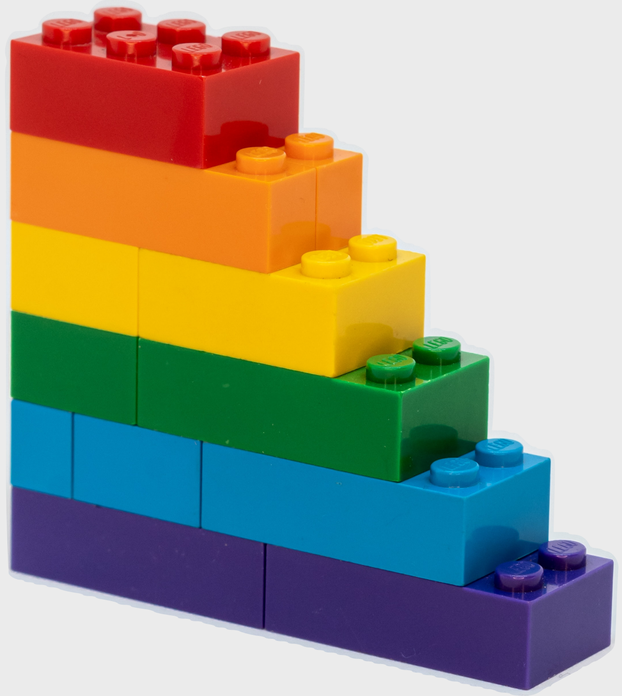


Photo by Xavi Cabrera on Unsplash

Destructors — Do We Really Need Them?

🎯 **Special function** called when an object goes out of scope.

🛑 Used to **release resources** (memory, files, connections, etc.).

🏗️ Same syntax as constructors — just add a leading ~

15-Date-Destructor/date.h

```h
class Date {
public:
    Date(int month=1, int day=1);
    ~Date();
    int month();
    int day();
private:
    int month_;
    int day_;
    bool isDate(int month, int day);
};
```

15-Date-Destructor/date.cpp

```cpp
Date::~Date() {
    std::cout << "Object Date "
              << month_ << "/" << day_
              << " is destroyed " << std::endl;
}
```

✅ In this example, the destructor simply prints a message, showing when the object is destroyed.

Destructors — Called Automatically at Object End-of-Life

15-Date-Destructor/main.cpp

```cpp
#include <iostream>
#include "date.h"

int main() {
    Date pi_day(3,14);
    std::cout << "That's all, folks!" << std::endl;
    return 0;
}
```

✅ You don't call destructors — the compiler does, automatically when the object leaves its scope (= block { ... }).

🛑 Default provided — compiler generates one automatically.

📦 Only needed if your class manages resources (dynamic memory, file handles, sockets, etc.).

```
→  15-Date-destructor ls
Makefile date.cpp date.h   main.cpp

→  15-Date-destructor make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o

→  15-Date-destructor ./build/app
That's all, folks!
Object Date 3/14 is destroyed
```

✨ "Most modern C++ classes don't need explicit destructors — unless they manage raw resources."

Designing the Date Class

📦 Variables

We need two variables month and day represented as integers.

```
int month_; // Use snake_case for naming variables
int day_;   // Use "_" suffix for variables
```

🚀 Constructor

We also need a constructor to initialise new objects.

```
Date(int month, int day);
```

⚙️ Methods/Functions

At least, we need two getter functions to read the variables.

In a second step, we will add other functions that will implement specific tasks.

```
int month();      // Use camelCase naming for
int day();        // methods with explicit names
```

Photo by Charles Deluvio on Unsplash

Getters — Accessing Private Data Safely

🔒 Member variables are **private** and cannot be accessed directly.

📖 For each variable that needs to be read, provide a **public getter** method.

📝 No input parameter, return type = type of the member variable.

🏷️ Naming conventions: **camelCase** with Obj-C/Swift style/ C++ STL: `int Date::month()` or Java style: `int Date::getMonth()`

16-Date-getters/date.h

```h
class Date {
public:
    Date(int month=1, int day=1);
    int month();
    int day();
private:
    int month_;
    int day_;
    bool isDate(int month, int day);
};
```

16-Date-getters/date.cpp

```cpp
#include "date.h"

int Date::month() {
    return month_;
}

int Date::day() {
    return day_;
}
```

✨ “Getters provide controlled, read-only access to private data.”

Const Methods — a Promise of No Change

🔒 A const method guarantees it will **not modify** any member variables.

🛡️ If code tries to change a member inside a const method → **compile-time error**.

📖 Useful both for the compiler (optimizations) and for humans (readability & trust).

✅ Always **mark methods as const** when possible.

16-Date-getters/date.h

```
1 int month();
2 int day();
```

❌

17-Date-const-getters/date.h

```
1 int month() const;
2 int day() const;
```

✅

16-Date-getters/date.cpp

```
1 int Date::month() {
2     return month_;
3 }
4 int Date::day() {
5     return day_;
6 }
```

❌

17-Date-const-getters/date.cpp

```
1 int Date::month() const {
2     return month_;
3 }
4 int Date::day() const {
5     return day_;
6 }
```

✅

✨ “Const turns your methods into a contract: read-only and safe.”

Setters — Updating Object State

✍️ Define **setters** when **updates** are needed → allow controlled modification of private variables.

📥 Input matches member type, return is void → argument type is the same as the variable type.

🏷️ Use **camelCase** naming convention → `setVariableName()` or `updateVariableName()`.

🔒 Not const → setters always modify the object's state.

18-Date-setters/date.h

```
1 class Date {
2 public:
3    Date(int month=1, int day=1);
4    void updateMonth(int new_month);
5    void updateDay(int new_day);
6 private:
7    int month_;
8    int day_;
9 };
```

18-Date-setters/date.cpp

```
1 void Date::updateMonth(int new_month) {
2     month_ = new_month;
3 }
4
5 void Date::updateDay(int new_day) {
6     day_ = new_day;
7 }
```

✨ "Setters give write access to private data — Use them only when updates are truly necessary."

Using Getters and Setters

18-Date-setters/main.cpp

```cpp
#include <iostream>
#include "date.h"

int main() {
    Date a_day(6,21);
    std::cout << "Summer: " << a_day.month() << "/" << a_day.day() << std::endl;
    a_day.updateMonth(9);
    std::cout << "Autumn: " << a_day.month() << "/" << a_day.day() << std::endl;
    return 0;
}
```

```
→  18-Date-setters ls
Makefile date.cpp date.h   main.cpp
→  18-Date-setters make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o
→  18-Date-setters ./build/app
Summer: 6/21
Autumn: 9/21
```

Do Not Forget To Ensure Object Validity When Updating

19-Date-update-objects/date.cpp

```cpp
void Date::updateDay(int new_day) {
    if (!isDate(month_, new_day)) {
        throw std::invalid_argument("Invalid date: "
            + std::to_string(month_) + "/" + std::to_string(new_day));
    }
    day_ = new_day;
}
```

19-Date-update-objects/main.cpp

```cpp
int main() {
    Date love(2,14);
    std::cout << "Valentine day: " << love.month() << "/" << love.day() << std::endl;
    love.updateDay(30);
    return 0;
}
```

```
➜  19-Date-update-objects ls
Makefile date.cpp date.h   main.cpp

➜  19-Date-update-objects make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c date.cpp -o build/date.o
clang++ -Wall  -o build/app  build/main.o  build/date.o

➜  19-Date-update-objects ./build/app
Valentine day: 2/14
libc++abi: terminating due to uncaught exception of type std::invalid_argument: Invalid date: 2/30
[1]    85008 abort      ./build/app
```

Designing a Next Day Method

Imagine you need a method to compute tomorrow's date... How would you write it?

20-Date-add-methods/date.cpp

```cpp
void Date::incDay() {
    if (day_==getDaysInMonth(month_)) {
        day_=1;
        month_=(month_)%12 + 1;
    }
    else {
        day_++;
    }
}
```

See the Github repo for the implementation details of getDaysInMonth().

20-Date-add-methods/date.cpp

```cpp
Date Date::nextDay() const {
    if (day_==getDaysInMonth(month_)) {
        return Date((month_)%12 + 1, 1);
    }
    return Date(month_, day_+1);
}
```

Option 1 — 🔄 Mutable method

"Modifies the current object."

✅ Efficient: no extra copy, directly updates the object.

❌ Risky: object changes in place, may cause side effects.

Option 2 — 🆕 Immutable method

"Returns a brand new object."

✅ Safer: original object unchanged, clearer semantics.

❌ Slight overhead: requires returning a new object.

✨ "Both designs are valid — Choice depends on your application's needs."

Designing Methods: When To Create Getters, Setters, or Custom Methods

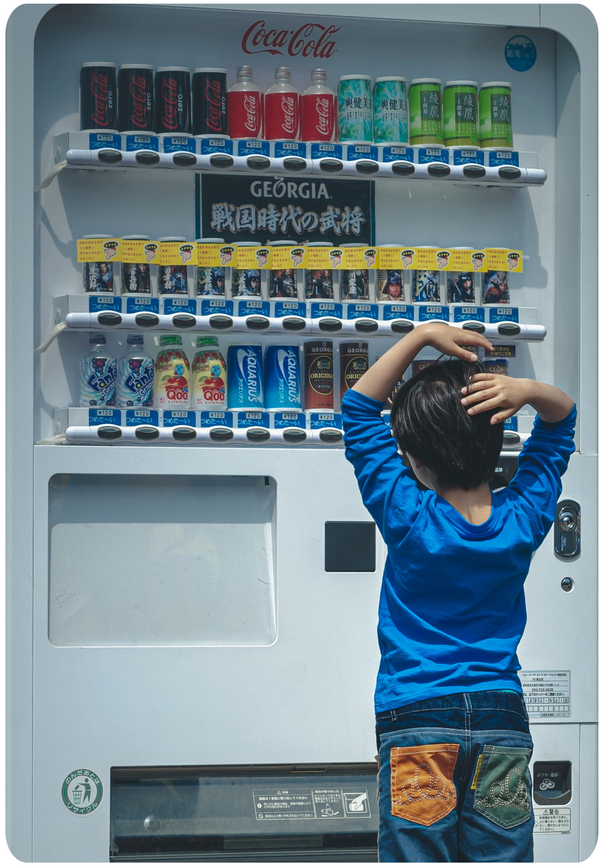


Photo by Egor Myznik on Unsplash

🔒 **Internal-only variables** → no getter, no setter.

Keep implementation details hidden.

📖 **Read-only variables** → getter only.

Exposed for reading, fixed after initialization.

✍️ **Read/Write variables** → getter + setter.

Full access when safe to change.

🔗 **Consistent variables** → update together.

Ensure integrity by grouping related changes.

➕ **Restricted update variables** → custom methods.

Encapsulate the allowed modifications.

✨ "Not every variable deserves a getter/setter — design methods that reflect the real usage and constraints of your class."

Free functions

🧩 **Independent block of code** → takes inputs, computes, returns output.

🌍 **Defined outside any class** → can be reused anywhere in the program.

📦 **Works only with external data** → operates exclusively on passed arguments.

Methods

🏷️ **Bound to a class** → declared/defined inside a class.

👤 **Belongs to an object** → invoked through an instance.

🔒 **Direct access to internal data** → and optionally combines it with external arguments.

Free Functions vs Methods: What's the Difference?

✨ "If a method doesn't really use the object's internal data, maybe it doesn't belong to the class at all — it could be a free function instead."

When a Method Should Be a Free Function

The case of Date::`isDate(int month, int day)`

❌ Does not use any member variable of Date → it does not depend on the object's state.

✅ Works only with the parameters passed in → it behaves like a regular function.

📝 Same code, different signature → Date::isDate(...) vs. is_date(...).

21-Date-methods-vs-functions/date.cpp

```cpp
bool Date::isDate(int month, int day) {
  if ((day < 1) || (day>31)) return false;
  if ((month <1) || (month>12)) return false;
  if ((month == 2) && (day > 28)) return false;
  if (((month == 4) || (month == 6) ||(month == 9)
  || (month == 11)) && (day > 30)) return false;
  return true;
}
```

❌

```cpp
bool is_date(int month, int day) {
  if ((day < 1) || (day>31)) return false;
  if ((month <1) || (month>12)) return false;
  if ((month == 2) && (day > 28)) return false;
  if (((month == 4) || (month == 6) ||(month == 9)
  || (month == 11)) && (day > 30)) return false;
  return true;
}
```

✅

✨ "Better implemented as a **free function** than as a class method."

Same case for `int Date::getDaysInMonth(int month)` which can be transformed into `int get_days_in_month(int month);`

When a Method Should Stay a Method

The case of Date::dayOfYear()

🔒 Uses object state (day_, month_) → this behavior belongs to Date.
It relies on internal attributes; keeping it as a method preserves encapsulation.

🔁 Could be written as a free function using getters → yes, it compiles…
…but it drifts toward exposing attributes via API (getters), increases coupling, and dilutes the type's responsibility.

🎯 dayOfYear() belongs to core semantics (domain rules) of Date; it's part of what a Date is/does.
Always keep domain rules inside the domain type.

21-Date-methods-vs-functions/date.cpp

```cpp
int Date::dayOfYear() const {
    auto day=day_;
    for (auto i=1;i<month_;i++) {
        day+=getDaysInMonth(i);
    }
    return day;
}
```

✨ "Moving domain rules outside a class dilutes its responsibility.

Keep core logic in the class."

Formatting a Date as “MM-DD” — Method or Function?

21-Date-methods-vs-functions/date.cpp

```cpp
std::string Date::toString() const {
    return std::to_string(month_) + “-"
         + std::to_string(day_);
}
```

❌

```cpp
std::string to_string(Date d) {
    return std::to_string(d.month()) + “-"
         + std::to_string(d.day());
}
```

✅

✅ The best choice: Free function

- 🎯 Formatting is **external** → not part of the core responsibility of Date.
- 🔄 Allows multiple formatting styles (ISO, FR, US…) without polluting the class.
- 🧩 Keeps Date focused on its true role: **representing and manipulating a valid date**.

✨ “A class should model only its core responsibility — delegate secondary concerns to free functions.”

From Methods vs Functions... to Design Principles

What we just did with `Date::isDate()`, `Date::dayOfYear()` and `to_string()` was not random.

📏 It was an application of a fundamental OOP design rule → Keep **domain rules inside the class**; delegate non-core concerns outside.

🧩 This design rule is named SRP — **Single Responsibility Principle**.

🚀 And SRP is only the first step of a broader set of guidelines for building robust, flexible, and maintainable OOP systems, called 🏛️ **SOLID.**

Robert C. Martin (*Uncle Bob*), "Design Principles and Design Patterns", Object Mentor, 2000

🌐 https://ginhac.com/ITC313/solid.pdf

S — Single Responsibility Principle (SRP)

📄 *A class should have only one reason to change.*

O — Open/Closed Principle (OCP)

📂 *Classes should be open for extension but closed for modification.*

L — Liskov Substitution Principle (LSP)

🔁 *Subtypes must be substitutable for their base types.*

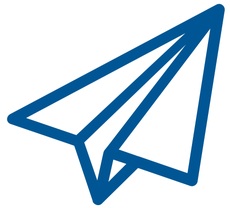
I — Interface Segregation Principle (ISP)

🧩 *Prefer many small, specific interfaces to one large, general-purpose interface.*

D — Dependency Inversion Principle (DIP)

⚡ *Depend on abstractions, not on concrete implementations.*

A SECOND TAKE HOME MESSAGE



#2

🔒 **Encapsulation** = Hide the “how” → Protect data & implementation details.

Getters, setters, and custom methods are the controlled gates to access or modify data.

🎭 **Abstraction** = Show the “what” → Expose only the meaningful operations.

Group related updates and design methods that reflect real-world operations.

📏 **Single Responsibility Principle** = Focus on the “why” → One domain rule for a class.

A class should change for only one reason: its core responsibility.

Questions

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of a class
3. Value, Reference, Pointer
4. Other user-defined types

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of a class
3. Value, Reference, Poi
4. Other user-defined types

Functions and arguments

“How can we pass/return objects to/from functions?”

Pass by Values (T)

Default behavior

Pass by References (T&)

Specific to C++

Pass by Pointers (T*)

Old school C-based mechanism

Passing By Value — the Copy Trap

22-Person-by-value/date.cpp

```cpp
void swap_names(Person p) {
    std::string firstname= p.firstname();
    p.setFirstname(p.lastname());
    p.setLastname(firstname);
}
```

22-Person-by-value/main.cpp

```cpp
int main() {
    Person me("Ginhac", "Dom", 1);
    std::cout << "Before swap: "
              << getFullName(me) << std::endl;
    swap_names(me);
    std::cout << "After swap: "
              << getFullName(me) << std::endl;
    return 0;
}
```

📄 let's write a simple swap_names() utility function to exchange the firstname and the lastname of a Person.

```
→  22-Person-by-value make
clang++ -Wall -g -MMD -c main.cpp -o build/main.o
clang++ -Wall -g -MMD -c date.cpp -o build/date.o
clang++ -Wall -g -MMD -c person.cpp -o build/person.o
clang++ -Wall -g  -o build/app  build/main.o  build/date.o  build/person.o

→  22-Person-by-value ./build/app
Before swap: Mr Ginhac Dom (5/26)
After swap: Mr Ginhac Dom (5/26)
```

❌ The code runs... but the names are not swapped.

🤔 Why did it fail? What's really happening under the hood?

🛠️ Time to dive in with LLDB and inspect memory step by step.

LLDB in Action

```
→  22-Person-by-value lldb build/app
Process 77186 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 1.1
    frame #0: 0x00000001000004fc app`main at main.cpp:5:15
   2    #include "person.h"
   3
   4    int main() {
-> 5        Person me("Ginhac", "Dom", 1, Date(5,26));
   6        std::cout << "Before swap: " << getFullName(me) << std::endl;
   7        swap_names(me);
   8        std::cout << "After swap: " << getFullName(me) << std::endl;
Target 0: (app) stopped.
(lldb) continue
Process 77186 resuming
Before swap: Mr Ginhac Dom (5/26)
Process 77186 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = breakpoint 2.1
    frame #0: 0x0000000100002ae0 app`swap_names(p=Person @ 0x000000016fdfebd8) at person.cpp:51:30
   48   }
   49
   50   void swap_names(Person p) {
-> 51       std::string firstname= p.firstname();
   52       p.setFirstname(p.lastname());
   53       p.setLastname(firstname);
   54   }
Target 0: (app) stopped.
(lldb) frame variable -L
0x000000016fdfebd8: (Person) p = {
0x000000016fdfebd8:   firstname_ = "Ginhac"
0x000000016fdfebf0:   lastname_ = "Dom"
0x000000016fdfec08:   gender_ = 1
0x000000016fdfec0c:   birthday_ = {
0x000000016fdfec0c:     month_ = 5
0x000000016fdfec10:     day_ = 26
  }
}
0x000000016fdfeb30: (std::string) firstname = ""
```

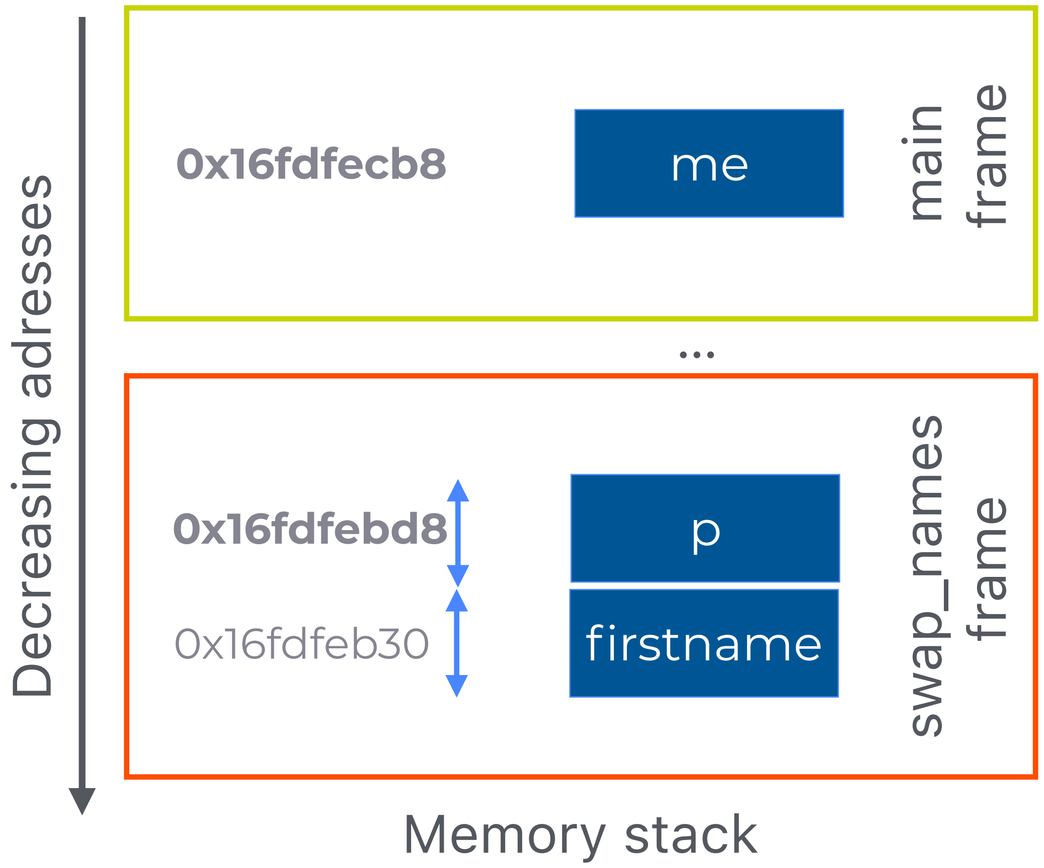
📌 ‘frame variable -L’ shows the variables of the current function (swap_dates) → here, they are copies of the originals.

📦 Different addresses = proof that values were copied from main into the function’s stack frame.

🛑 When the function ends, its stack frame disappears → changes don’t affect the original objects.

🚀 Time for a live demo — let’s break, inspect, and understand what really happens in memory with LLDB.

How Function Calls Really Work — the Copy Version



📦 The function call creates a **new stack frame** on the memory stack → swap_names frame

📌 Objects store only their attributes in the stack frame — methods live once in the code segment and are shared by all instances.

📦 Arguments are **copied** into the callee's frame → me from main frame is copied into p of swap frame.

🔄 The callee works only on its **local copies**, leaving the caller's variables unchanged.

⬆️ When the function ends, its frame is popped off the stack, and the **copies disappear.**

✨ "Passing by value (T) = copies live in the stack... and die 🪦 in the stack."

*So, how can we keep changes alive outside the function? Time to discover **references (&)**.*

Functions and arguments

“How can we pass/return objects to/from functions?”

Pass by Values (T)

Default behavior

Pass by References (T&)

Specific to C++

Pass by Pointers (T*)

Old school C-based mechanism

What Is a Reference?

📌 A reference (T&) is just another name for the same variable — not a new object.

🪞 Any change through the reference changes the original.

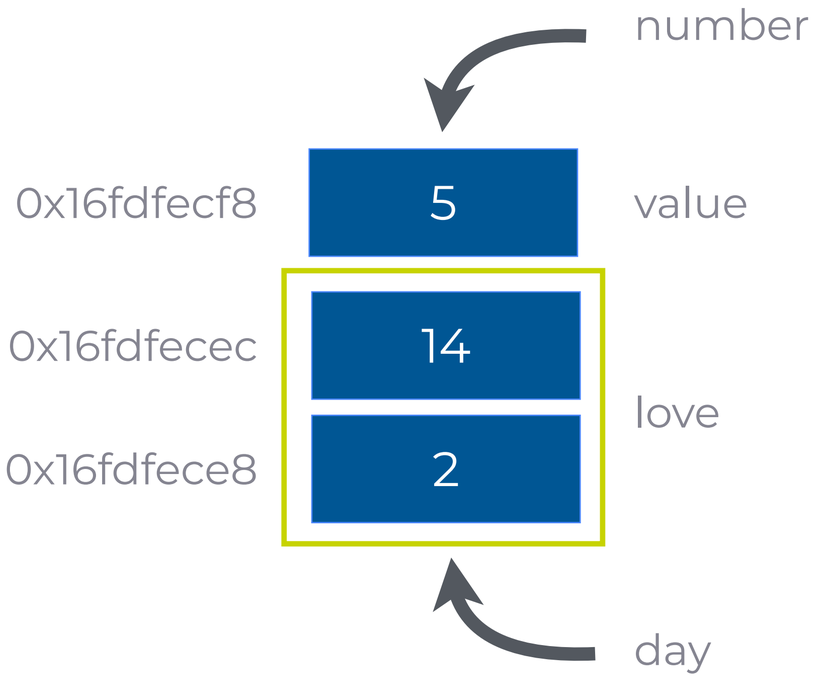
🚫 Must be initialized at declaration, cannot be null.

🔒 Once bound, a reference cannot be reseated to another variable.

✅ Same address in memory as the original variable.

23-References/main.cpp

```cpp
int value = 5;          // value is 5
int& number = value; // reference to value
Date love(2,14);      // Valentine's day
Date& day = love;     // reference to valentine's day
```



✨ “One variable, two names — or more. References (T&) are just aliases"

Passing by Reference — the Power of Sharing

23-Passing-by-reference/main.cpp

```cpp
void foo(int& y) {
    y++;
}

int main() {
    int x = 5;
    // Create an explicit reference z
    int& z = x;
    std::cout << "Before foo, x = " << x
              << " - z = " << z << std::endl;
    // Call foo with the reference z
    foo(z);
    std::cout << "After foo(z), x = " << x
              << " - z = " << z << std::endl;
    // Compiler creates an implicit reference
    // on x and call foo
    foo(x);
    std::cout << "After foo(x), x = " << x
              << " - z = " << z << std::endl;
    return 0;
}
```

🧩 References (T&) are often used as function parameters

→ *They let the function operate directly on the caller's variable.*

🚫 No copy is made

→ *The reference parameter (T&) is an alias, not a separate object.*

✍️ Modifications inside the function affect the original argument

→ *Changes persist after the function returns.*

⚡ Efficient and natural syntax

→ *You call the function exactly as if it took values — no extra symbols like & or *.*

```
→  24-Passing-by-reference make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall  -o build/app  build/main.o

→  24-Passing-by-reference ./build/app
Before foo, x = 5 - z = 5
After foo(z), x = 6 - z = 6
After foo(x), x = 7 - z = 7
```

Passing by Reference — the Power of Sharing

25-Person-by-reference/date.cpp

```cpp
void swap_names(Person& p) {
    std::string firstname= p.firstname();
    p.setFirstname(p.lastname());
    p.setLastname(firstname);
}
```

25-Person-by-reference/main.cpp

```cpp
int main() {
    Person me("Ginhac", "Dom", 1);
    std::cout << "Before swap: "
              << getFullName(me) << std::endl;
    swap_names(me);
    std::cout << "After swap: "
              << getFullName(me) << std::endl;
    return 0;
}
```

✅ The code runs... and the names are swapped.

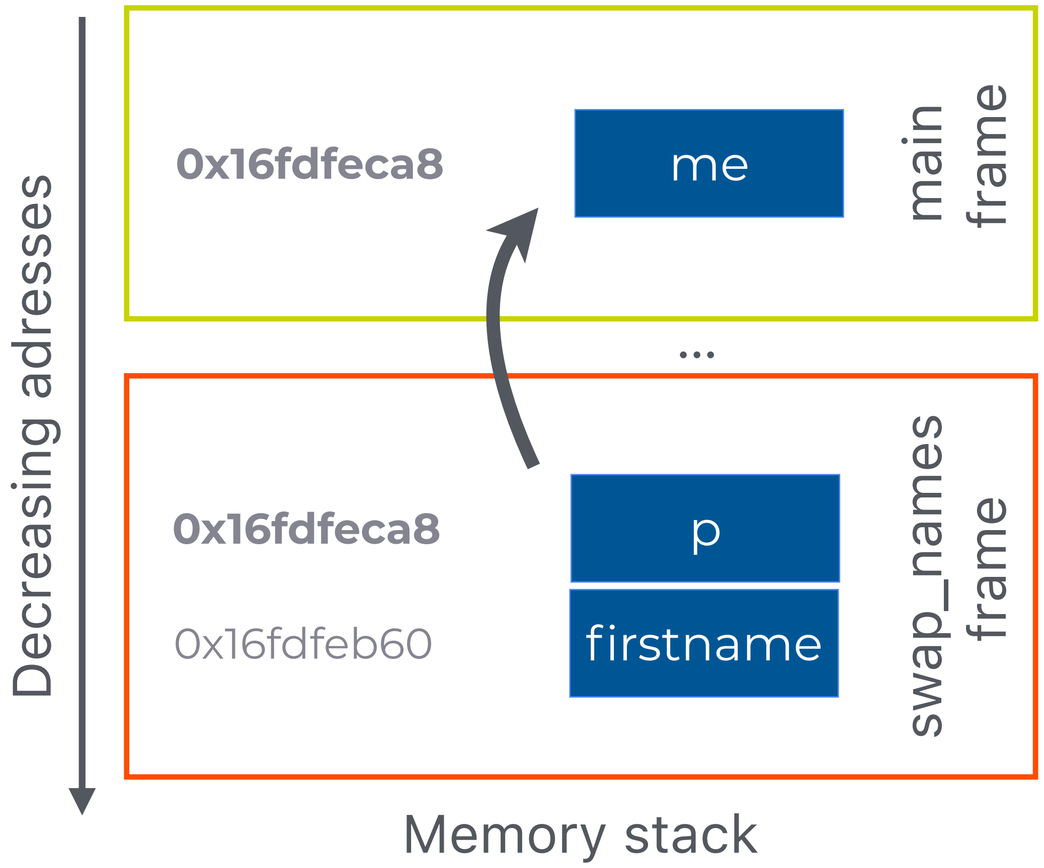
```
→  25-Person-by-reference make
clang++ -Wall -g -MMD -c main.cpp -o build/main.o
clang++ -Wall -g -MMD -c date.cpp -o build/date.o
clang++ -Wall -g -MMD -c person.cpp -o build/person.o
clang++ -Wall -g  -o build/app  build/main.o  build/date.o  build/person.o

→  25-Person-by-reference ./build/app
Before swap: Mr Ginhac Dom (5/26)
After swap: Mr Dom Ginhac (5/26)
```

💡 Same syntax as before, but a completely different memory behavior.

🔗 This time, the function works on the *original objects*, not on copies.

How Function Calls Really Work — the Reference Version



📦 The function call creates a new stack frame → here, the swap_names frame.

📦 The swap_names function uses a reference (T&) as parameter → A new reference (T&) is created from the variable “me” and copied into “p” in the swap frame.

🔗 This **local reference** (T&) **binds to the original object**, allowing direct modification.

⬆️ When the function ends, the swap frame is popped off the stack — the local reference disappears, but **the original Person object remains modified**.

✨ “Passing by reference (T&) = no copies… real impact on the originals.”

By reference — the stack frame disappears, but the changes persist.

Passing by Const Reference — Safe and Efficient

⚙️ No copy is made, just like a normal reference (T&).

🧱 Const guarantees the function cannot modify the original argument.

💨 Perfect for large objects — efficient and safe at the same time.

👀 The compiler enforces immutability → any modification raises a compile-time error.

26-Date-const-reference/date.cpp

```cpp
bool is_before(const Date& d1, const Date& d2) {
    if (d1.month() < d2.month()) return true;
    if (d1.month() == d2.month() && d1.day() < d2.day()) return true;
    return false;
}

bool is_after(const Date& d1, const Date& d2) {
    return is_before(d2, d1);
}
```

✨ “Const reference (const T&) = the speed of references, the safety of values."

Choosing How To Pass Arguments

✅ Advantages of References

🔄 Allow a function to modify the original argument.

📦 Enable returning multiple values efficiently.

⚡ Much faster than copying large objects.

⚠️ Drawbacks of References

🤔 Hard to see whether a reference parameter is for input, output, or both.

🕳️ Inside the function, you can't tell a reference from a local copy.

🎯 Guidelines

✍️ Use pass-by-value (T) for fundamental types (int, char, bool,...) that don't need to be modified.

🔗 Use pass-by-reference (T&) when the function must modify the data, whatever the type of data.

🔒 Use pass-by-const-reference (const T&) for large/complex objects that you don't need to modify — safe and efficient.

Functions and arguments

“How can we pass/return objects to/from functions?”

Pass by Values (T)

Default behavior

Pass by References (T&)

Specific to C++

Pass by Pointers (T*)

Old school C-based mechanism

The Concept of Pointers

📍 Pointers (T*) are one of the most powerful and unique features of C and C++.

→ *They allow direct access to memory — you can store, read, and manipulate addresses.*

💡 A pointer (T*) is a variable that holds the address of another variable.

→ *It doesn't store a value itself, but where that value lives in memory.*

🔄 Pointers (T*) look like references... but they are not the same.

→ *They can be reassigned, set to nullptr, or point to nothing at all — unlike references.*

Photo by Nathalie SPEHNER on Unsplash

✨ "Pointers (T*) give you control over memory — With great power comes great responsibility!"

Pointers Hold Addresses — Not Values!

27-Pointers/case1.cpp

📍 Pointers (T*) are **variables** that store **memory addresses** — not the values themselves.

→ *They allow indirect access and manipulation of data in memory.*

📝 Declared with an asterisk (*****) — means "**pointer to**".

→ *As any other variable (except reference), pointers can be declared without being initialized — But it's unsafe.*

```
1    int* iptr; // iptr is a pointer to int
2    double* dptr; // dptr is a pointer to double
3    Date* dateptr; // dateptr is a pointer to Date
```

📦 Initialized to **nullptr** when not pointing anywhere.

→ *This avoids undefined behavior from uninitialized pointers.*

```
1    iptr = nullptr; // ptr is initialized to null
2    dptr = nullptr; // dptr is initialized to null
3    dateptr = nullptr; // dateptr is initialized to null
```

Pointers Hold Addresses — Not Values!

27-Pointers/case2.cpp

🔗 Pointers (T*) can **change** what they point to, using the **address-of** operator (&).

→ *A pointer can be reassigned to another variable of same type.*

```
1     int value1 = 42;
2     int value2 = 666;
3     int* iptr = nullptr; // iptr is initialized to null
4     iptr = &value1; // iptr points to value1
5     iptr = &value2; // iptr now points to value2
```

⚙️ The **& operator** retrieves the **memory address** of a variable.

→ *Even if that variable is a pointer..*

```
1     // print the address of value1
2     std::cout << "&value1: " << &value1 << std::endl;
3     // print the address of value2
4     std::cout << "&value2: " << &value2 << std::endl;
5     // print the address stored in iptr (address of value2)
6     std::cout << "iptr: " << iptr << std::endl;
7     // print the address of iptr (different from address of value2)
8     std::cout << "&iptr: " << &iptr << std::endl;
```

```
➜  27-Pointers make -f Makefile2
clang++  -MMD -c case2.cpp -o build/case2.o
clang++   -o build/app  build/case2.o
```

```
➜  27-Pointers ./build/app
&value1: 0x16d776ea8
&value2: 0x16d776ea4
iptr: 0x16d776ea4
&iptr: 0x16d776e98
```

Pointers Hold Addresses — Not Values!

27-Pointers/case3.cpp

✨ Don't confuse the **two** potential **uses of &**.

In a declaration → Defines a reference.

In an expression → Gives the address of a variable.

```
1     // The golden ratio
2     double golden=1.61803;
3     // dptr is a pointer to golden
4     double* dptr = &golden;
5     // ref is a reference to golden
6     double& ref = golden;
7     std::cout << "golden = " << golden << std::endl;
8     std::cout << "ref = " << ref << std::endl;
9     std::cout << "dptr = " << dptr << std::endl;
10    std::cout << "&golden= " << &golden << std::endl;
11    std::cout << "&ref = " << &ref << std::endl;
12    std::cout << "&dptr = " << &dptr << std::endl;
```

```
➜  27-Pointers make -f Makefile3
clang++  -MMD -c case3.cpp -o build/case3.o
clang++   -o build/app  build/case3.o

➜  27-Pointers ./build/app
golden = 1.61803
ref = 1.61803
dptr = 0x16b8baea0
&golden= 0x16b8baea0
&ref = 0x16b8baea0
&dptr = 0x16b8bae98
```

✨ "Same symbol, two meanings — context is everything!"

Pointers Hold Addresses — Not Values!

27-Pointers/case4.cpp

🧭 The * (**dereference**) operator accesses the **value stored** at that address.

→ *It lets you read or modify the variable being pointed to*

```cpp
int value=42;
int* iptr = &value;
// print the value pointed to by iptr
std::cout << "*iptr: " << *iptr << std::endl;
// value2 is initialized to the value pointed by iptr plus 10
int value2 = *iptr + 10;
std::cout << "value2: " << value2 << std::endl;
// change the value pointed by iptr
*iptr = 21;
std::cout << "*iptr: " << *iptr << std::endl;
// value is also changed because iptr points to value
std::cout << "value: " << value << std::endl;
// but value2 is unchanged because it is a different variable
std::cout << "value2: " << value2 << std::endl;
```

```
→  27-Pointers make -f Makefile4
clang++  -MMD -c case4.cpp -o build/case4.o
clang++   -o build/app  build/case4.o

→  27-Pointers ./build/app
*iptr: 42
value2: 52
*iptr: 21
value: 21
value2: 52
```

✨ “Dereferencing gives direct access to the original variable, not a copy."

Pointers Hold Addresses — Not Values!

27-Pointers/case5.cpp

✨ Don't confuse the **two** potential **uses** of *

In a declaration → Defines a pointer type (T).*

In an expression → Dereferences (*) to access the pointed value.

```
1     Date* dateptr = nullptr; // dateptr is a null pointer to Date
2     Date pi_day(3,14); // pi_day is a Date
3     dateptr = &pi_day; // dateptr now points to pi_day
4
5     int day = (*dateptr).day();
6     std::cout << "day: " << day << std::endl;
7     // the above line can be written more simply as:
8     int day2 = dateptr->day();
9     std::cout << "day (v2): " << day2 << std::endl;
```

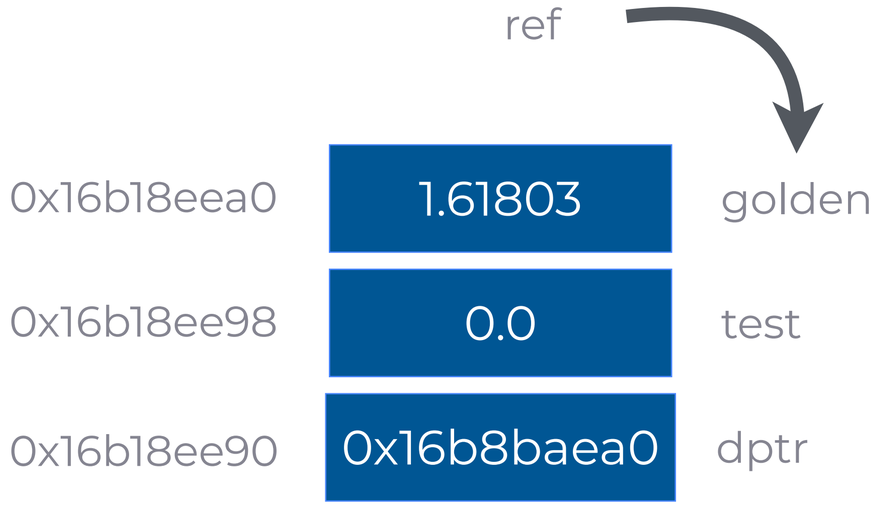
```
➜  27-Pointers make -f Makefile5
clang++  -MMD -c case5.cpp -o build/case5.o
clang++  -MMD -c date.cpp -o build/date.o
clang++   -o build/app  build/case5.o  buil
d/date.o
➜  27-Pointers ./build/app
day: 14
day (v2): 14
```

✨ "Same symbol, two meanings — context is everything!"

References vs Pointers — What Really Happens in Memory

27-Pointers/case6.cpp

```cpp
    double golden = 1.61803; // The golden ratio
    double test = 0.0; // A test variable
    double* dptr = &golden;
    double& ref = golden;
    std::cout << "&golden= " << &golden << std::endl;
    std::cout << "&ref = " << &ref << std::endl;
    std::cout << "dptr = " << dptr << std::endl;
    std::cout << "&dptr = " << &dptr << std::endl;
    std::cout << "&test = " << &test << std::endl;
    dptr = &test; // Now dptr points to test
    std::cout << "dptr = " << dptr << std::endl;
```




```
➜  27-Pointers make -f Makefile6
clang++  -MMD -c case6.cpp -o build/case6.o
clang++   -o build/app  build/case6.o

➜  27-Pointers ./build/app
&golden= 0x16b18eea0
&ref = 0x16b18eea0
dptr = 0x16b18eea0
&dptr = 0x16b18ee90
&test = 0x16b18ee98
dptr = 0x16b18ee98
```

✨ “A reference (T&) is the object it refers to — a pointer (T*) knows where the object is.”

Passing by Pointer — Indirection Power (and Risk)

📩 Pointers (T*) as parameters of functions/methods

→ *Instead of passing a variable, you pass its address.*

→ *The function receives a pointer to the argument variable.*

🔧 Direct access to original data

→ *By dereferencing the pointer (*ptr), the function can read or modify the original value.*

→ *Same capability as “pass by reference,” but with explicit indirection.*

⚠️ Handle with care

→ *A pointer can be nullptr or uninitialized — dereferencing it causes crashes.*

→ *Always check validity before use.*

🧭 Syntax reminder

→ *Use & when calling the function (to pass the address).*

→ *Use * inside the function (to access the pointed value).*

✨ “Pointers (T*) give you full control over memory access — but with great power comes great responsibility."

Passing by Pointer — Indirection Power (and Risk)

28-Person-by-pointer/date.cpp

```cpp
void swap_names(Person* p) {
    std::string firstname= p->firstname();
    p->setFirstname(p->lastname());
    p->setLastname(firstname);
}
```

28-Person-by-pointer/main.cpp

```cpp
int main() {
    Person me("Ginhac", "Dom", 1, Date(5,26));
    std::cout << "Before swap: "
              << getFullName(me) << std::endl;
    swap_names(&me);
    std::cout << "After swap: "
              << getFullName(me) << std::endl;
    return 0;
}
```

✅ The code runs… and the names are swapped.

```
→  28-Person-by-pointer make
clang++ -Wall -g -MMD -c main.cpp -o build/main.o
clang++ -Wall -g -MMD -c date.cpp -o build/date.o
clang++ -Wall -g -MMD -c person.cpp -o build/person.o
clang++ -Wall -g  -o build/app  build/main.o  build/date.o  build/person.o

→  28-Person-by-pointer ./build/app
Before swap: Mr Ginhac Dom (5/26)
After swap: Mr Dom Ginhac (5/26)
```

💡 **Same behavior as references, but explicit syntax with * and &.**

→ Pointers require you to manually handle addresses and dereferencing.

⚙️ **More flexible than references, but less safe.**

→ Pointers can be reassigned or be nullptr, while references are always bound and valid.

When Should I Use References, and When Should I Use Pointers?

Photo by JESHOOTS.COM on Unsplash

💡 Prefer references (T&)— they're safer and simpler.

→ *Always valid, automatically dereferenced, and cannot be null.*

→ *Ideal for function parameters and return values when ownership isn't needed.*

⚙️ Use pointers (T*) only when necessary.

→ *When you need optional parameters (nullptr means "no object").*

→ *When dealing with dynamic memory or resource management.*

🧩 Modern C++ best practice

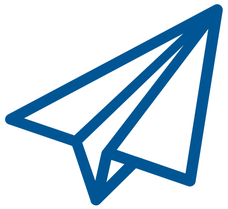
→ *Use references (T&) by default.*

→ *Use pointers (T*) intentionally, when needed.*

→ *Use smart pointers (unique_ptr, shared_ptr) for resource ownership.*

✨ "Use references (T&) for simplicity, pointers (T*) for flexibility, and smart pointers for safety."

A THIRD TAKE HOME MESSAGE



#3

C++ provides several ways to **pass arguments and return values** — each has its own intent and cost.

1. 🧩 To modify the object

   → *Pass by **reference** (T&) or by **pointer** (T*) if optional or nullable.*

2. 🔍 To read a complex object safely

   → *Use a **const reference** (const T&) to avoid copies and guarantee immutability.*

3. ⚡ For small or simple data (int, double, enum, ...)

   → *Pass and return by **value** (T) — copying is cheap and clear.*

4. 📦 To return large objects

   → *Prefer returning a **const reference** (const T&) — efficient and safe.*

Questions

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of a class
3. Value, Reference, Pointer
4. Other user-defined types

AGENDA

01 - User-defined Data Types

1. Basics of data types
2. A realistic example of a class
3. Value, Reference, Pointer

4.Other user-defined ty

Other C++ User-Defined Types

In addition to classes, C++ provides other ways to define custom data types — each with its own purpose and memory model.

STRUCTURES

PUBLIC CLASSES

Used mainly for simple data containers. Inherited from C.

ENUMERATION

FIXED SET OF VALUES

Used for variables that can only take one of several values. Perfect for states, modes, or labels

UNION

ONE MEMORY, MANY FORMS

Store different types in the same memory. Only one active type at a time.

A First Example of Struct

Struct = Public class

Used mainly for POD — plain old data (i.e. a bundle that just stores data with little logic).

Default access is public for backwards compatibility with C whereas default access is private for class.

29-Point/point.h

```h
#ifndef POINT_H
#define POINT_H

struct Point {
    double x; // x and y are public
    double y; // No need to write getters/setters
    // Constructor with default values
    Point(double xval=0.0, double yval=0.0);
    // Length from origin
    double length() const;
};

#endif // POINT_H
```

29-Point/point.cpp

```cpp
#include <cmath>
#include "point.h"

Point::Point(double xval, double yval) :
                x(xval), y(yval) {
}

double Point::length() const {
    return std::sqrt(x*x + y*y);
}
```

Using the Struct Point

29-Point/main.cpp

```cpp
#include <iostream>
#include "point.h"

int main() {
    Point p1;
    std::cout << "p1(" << p1.x << "," << p1.y << ")" << std::endl;
    std::cout << "Length from origin: " << p1.length() << std::endl;
    p1.x = 4.5; // x is public
    p1.y = 3.2; // x is public
    std::cout << "p1(" << p1.x << "," << p1.y << ")" << std::endl;
    std::cout << "Length from origin: " << p1.length() << std::endl;
    return 0;
}
```

```
→  29-Point make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c point.cpp -o build/point.o
clang++ -Wall  -o build/app  build/main.o  build/point.o
```

```
→  29-Point ./build/app
p1(0,0)
Length from origin: 0
p1(4.5,3.2)
Length from origin: 5.52178
```

Struct vs Class in C++

The only difference between struct and class is the default access level of their members.

30-Struct-vs-Class/pointstruct.h

```
1 struct PointStruct {
2     double x; // x and y are public
3     double y; // No need to write getters/setters
4 };
```

✅ PointStruct → accessible directly.

30-Struct-vs-Class/pointclass.h

```
 1 class PointClass {
 2     double x_; // x_ and y_ are private
 3     double y_; // No direct access from
outside the class
 4 };
```

❌ PointClass → compiler error if you try to access x or y

```
➜  30-Struct-vs-Class make
clang++ -Wall -MMD -c main.cpp -o build/main.o
main.cpp:12:8: error: 'x_' is a private member of 'PointClass'
   12 |     pc1.x_ = 4.5; // Error: x is private
      |         ^
./pointclass.h:5:12: note: implicitly declared private here
    5 |     double x_; // x_ and y_ are private
      |            ^
main.cpp:13:8: error: 'y_' is a private member of 'PointClass'
   13 |     pc1.y_ = 3.2; // Error: y is private
      |         ^
./pointclass.h:6:12: note: implicitly declared private here
    6 |     double y_; // No direct access from outside the class
      |            ^
main.cpp:14:31: error: no member named 'x' in 'PointClass'
   14 |     std::cout << "pc1(" << pc1.x << "," << pc1.y << ")" << std::endl;
      |                             ~~~ ^
main.cpp:14:47: error: no member named 'y' in 'PointClass'
   14 |     std::cout << "pc1(" << pc1.x << "," << pc1.y << ")" << std::endl;
      |                                             ~~~ ^
```

✨ “Struct is open and lightweight — ideal for plain data. Class enforces protection and encapsulation — ideal for logic.”

Struct vs Class in C++

The only difference between struct and class is the default access level of their members.

30-Struct-vs-Class/pointstruct.h

```
1 struct PointStruct {
2     double x; // x and y are public
3     double y; // No need to write getters/setters
4 };
```

✅ PointStruct → accessible directly.

30-Struct-vs-Class/pointclass.h

```
1 class PointClass {
2     double x_; // x_ and y_ are private
3     double y_; // No direct access
4 };
```

❌ PointClass → compiler error if you try to access x or y

30-Struct-vs-Class/main.cpp

```
1 int main () {
2     PointClass pc1;
3     pc1.x_ = 4.5; // Error: x is private
4     pc1.y_ = 3.2; // Error: y is private
5     std::cout << "pc1(" << pc1.x_ << ","
                  << pc1.y_ << ")" << std::endl;
6     return 0;
7 }
```

```
main.cpp:13:8: error: 'y_' is a private member of 'PointClass'
   13 |     pc1.y_ = 3.2; // Error: y is private
      |         ^
./pointclass.h:6:12: note: implicitly declared private here
    6 |     double y_; // No direct access from outside the class
      |            ^
main.cpp:14:31: error: 'x_' is a private member of 'PointClass'
   14 |     std::cout << "pc1(" << pc1.x_ << "," << pc1.y_ << ")" << std::endl;
      |                               ^
./pointclass.h:5:12: note: implicitly declared private here
    5 |     double x_; // x_ and y_ are private
      |            ^
main.cpp:14:48: error: 'y_' is a private member of 'PointClass'
   14 |     std::cout << "pc1(" << pc1.x_ << "," << pc1.y_ << ")" << std::endl;
      |                                                ^
./pointclass.h:6:12: note: implicitly declared private here
    6 |     double y_; // No direct access from outside the class
      |            ^
```

A First Example of Enum

Enum = Fixed set of values

An **enum** is a user-defined type that represents a fixed set of named constant values.

Each name corresponds to an integer constant (starting at 0 by default).

31-Status/status.h

```
1 enum Status { // Braces surround the entries
2     Pending,    // a comma-separated
3     Urgent,     // set of constants
4     Delayed,    // (integers) with unique names
5     Cancelled,
6     Done        // No comma after the last one
7 }; // Do not forget the semi-colon
```

🧠 Key Points

- 🏷️ Each label is internally an int (starting from 0, incremented automatically).
- 🚦 Useful for variables that can take **only one state** out of a defined set.
- 🪨 Improves code readability and maintainability compared to magic numbers.

✨ “Use enum to make your code more expressive and type-safe when working with a limited number of states (e.g. status, mode, level)."

A First Example of Enum

31-Status/main.cpp

```cpp
#include "status.h"

int main() {
    Status task = Delayed;
    std::cout << "Delayed task: " << task << std::endl;
    task = Done;
    if (task == Urgent) {
        std::cout << "Task requires immediate attention!"
                  << std::endl;
        return 0;
        }
    if (task == Done) {
        std::cout << "Task completed." << std::endl;
        return 0;
    }
    std::cout << "Task still in progress..." << std::endl;
    return 0;
}
```

31-Status/status.h

```cpp
enum Status {
    Pending,   // 0
    Urgent,    // 1
    Delayed,   // 2
    Cancelled, // 3
    Done       // 4
};
```

```
→  31-Status make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall  -o build/app  build/main.o
```

```
→  31-Status ./build/app
Delayed task: 2
Task completed.
```

Uniqueness of Values

Enum shares its names in the global scope → Naming conflicts possible between different enums!

```
1 enum Status { Urgent, Pending, Done };
2 enum Task   { Done, Skipped }; // ❌ Done redefined
```

C++11 has introduced enum classes (also called scoped enum), that makes enumerations both strongly typed and strongly scoped.

```
1 enum class Status { Urgent, Pending, Done };
2 enum class Task   { Done, Skipped }; // ✅ Scoped names, no conflict
```

32-Scoped-enum/main.cpp

```
1 int main() {
2     Status s = Status::Done;
3     Task t = Task::Done;
4     if (t == Task::Done)
5         std::cout << "Task is done!" << std::endl;
6     std::cout << "Status value: " << static_cast<int>(s)
                << std::endl;
7     return 0;
8 }
```

```
→  32-Scoped-Enum make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall  -o build/app  build/main.o
```

```
→  32-Scoped-Enum ./build/app
Task is done!
Status value: 2
```

A First Example of Union

Union = Shared memory Type

An **union** is a special user-defined type that allows storing different data types in the same memory location.

Only one member can be active at a time — all members share the same memory space.

33-Number/number.h

```
union Number {
    // 4 bytes - IEEE 754 floating point
    float real;
    // 4 bytes - signed integer (two's complement)
    int integer;
};
```

🧠 Key Points

⚙️ All members start at the same address.

📦 The union’s size = size of its largest member.

⚠️ Reading a member different from the one most recently written is undefined behavior.

🚫 You cannot store non-trivial types (e.g., std::string) unless using a variant or std::union wrapper.

✨ “Use union when memory matters or data overlap is intentional."

A First Example of Union

33-Number/main.cpp

```cpp
int main() {
    Number nb;
    nb.real = 3.14159;
    std::cout << "Bits: " << std::bitset<32>(nb.integer) << std::endl;
    std::cout << "Float (ok): " << nb.real << std::endl;
    std::cout << "Int (nok): " << nb.integer << std::endl;

    nb.integer = 42;
    std::cout << "Bits: " << std::bitset<32>(nb.integer) << std::endl;
    std::cout << "Int (ok): " << nb.integer << std::endl;
    std::cout << "Float (nok): " << nb.real << std::endl;
    return 0;
}
```

```
→  33-Number make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall  -o build/app  build/main.o
```

```
→  33-Number ./build/app
Bits: 01000000010010010000111111010000
Float (ok): 3.14159
Int (nok): 1078530000
Bits: 00000000000000000000000000101010
Int (ok): 42
Float (nok): 5.88545e-44
```

f3.14159 = d1078530000 = 0x01000000010010010000111111010000

d42 = 0x00000000000000000000000000101010 = f5.88545e-44

https://www.h-schmidt.net/FloatConverter/IEEE754.html

A More Complex Example of Union

34-SensorData/sensordata.h

```h
union SensorValue {
    int i;
    float r;
    char c;
    SensorValue(int value);
    SensorValue(float value);
    SensorValue(char value);
};

enum class ValueType { integer, real, character };

class SensorData {
public:
      SensorData(int data=0);
      SensorData(float data=0.0);
      SensorData(char data='\0');
      std::string to_string() const;
private:
      ValueType type_;
      SensorValue data_;
};
```

34-SensorData/sensordata.cpp

```cpp
SensorValue::SensorValue(int value) : i(value) {}
SensorValue::SensorValue(float value) : r(value) {}
SensorValue::SensorValue(char value) : c(value) {}
SensorData::SensorData(int data) : type_(ValueType::integer),
                                   data_(data) {}
SensorData::SensorData(float data) : type_(ValueType::real),
                                     data_(data) {}
SensorData::SensorData(char data) : type_(ValueType::character),
                                    data_(data) {}

std::string SensorData::to_string() const {
    if (type_ == ValueType::real) {
        return "real: " + std::to_string(data_.r);
    }
    if (type_ == ValueType::integer) {
        return "int: " + std::to_string(data_.i);
    }
    if (type_ == ValueType::character) {
        return "char: " + std::string(1, data_.c);
    }
    return "N/A";
}
```

A More Complex Example of Union

34-SensorData/main.cpp

```cpp
int main(int argc, char const *argv[]) {
    int int_nb = 42;
    SensorData data1(int_nb);
    std::cout << data1.to_string() << std::endl;
    float real_nb = 3.14159;
    SensorData data2(real_nb);
    std::cout << data2.to_string() << std::endl;
    char character = 'A';
    SensorData data3(character);
    std::cout << data3.to_string() << std::endl;
   return 0;
}
```

```
→  34-SensorData make
clang++ -Wall -MMD -c main.cpp -o build/main.o
clang++ -Wall -MMD -c sensordata.cpp -o build/sensordata.o
clang++ -Wall  -o build/app  build/main.o  build/sensordata.o
```

```
→  34-SensorData ./build/app
int: 42
real: 3.141590
char: A
```

✨ “For safer code, use modern C++17 alternatives → std::variant.”

Key Ideas To Remember

🧱 struct = public class

Ideal for Plain Old Data (POD) — simple bundles of related variables, with public access by default.

➜ *Use class when you need encapsulation, struct when you just store data.*

🎯 enum / enum class = clear choice sets

Replace arbitrary numbers or strings with named constants.

➜ *Make code more readable, safer, and self-documenting.*

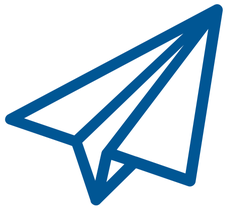
⚙️ union = shared memory for multiple views

All members occupy the same memory location.

➜ *Useful for low-level programming, binary data, or interfacing with C APIs.*

✨ “These user-defined types extend C++ expressiveness — from simple data containers (struct) to controlled choices (enum) and efficient memory reinterpretation (union), each serves a specific, well-delimited purpose.."

A FOURTH TAKE HOME MESSAGE



#4

C++ offers several ways to define your own types — each designed for a different level of abstraction and control.

Choose the simplest construct that fits your goal:

- 🏗️ *class for behavior + data,*
- 🧰 *struct for grouped values,*
- 🎯 *enum for fixed choices,*
- ⚙️ *union for efficient memory sharing.*

🧩 Together, they form the complete toolbox for modeling data in C++.

Questions

Contacts

Pr. Dominique Ginhac

dginhac@ube.fr

Come visit us at

https://github.com/dginhac/polytech-dijon-itc313