Introduction to C++

LaTeX by Min Soo Jeon
mjeon@uc.cl
Professor Danilo Ardagna
Politecnico di Milano

Semester 2024-2

Disclaimer:

This material is an unofficial resource created for educational purposes only. The authors and contributors do not assume any liability for errors, inaccuracies, or misinterpretations that may arise from its use. Readers should verify information from official sources.

Version 1.0.0

Introduction

These notes are based on the lectures delivered by Professor Danilo Ardagna during the Algorithms and Parallel Computing course at Politecnico di Milano during Semester 2024-2. Official course information can be found on the official program (Ardagna, 2024). Some sections, particularly the practical components, were taught by Teaching Assistant Federica Filippini.

This material may be outdated, the latest version is available in the repository. To report an error, please contact mjeon@uc.cl.

Contents

1 Introduction

1.1 The language

C++ is an oriented-object, static, and high-level programming language created by Bjarne Stroustrup in AT&T Bell Labs at Texas A&M University. It allows us to express ideas and perform tasks in code. C++ started from C, adding state-of-art features without losing high-efficiency performance.
The language is precisely and comprehensively defined by an ISO standard. The most recent standard is ISO C++ v23, we will focus on C++ v26

1.2 Oriented-object Programming

Oriented-object Programming is an approach to cope with large-scale software, that needs to be modified, evolved, and maintained. It consists of using objects to manage tasks in the programs. Objects are structures that are capable of storage attributes (data) and methods (functions to deal with object's related tasks)

1.3 Statically typed

Statically typed programming languages are the languages that check the variables at compile (see section 1.5) time rather than at run time. This means that variable types are determined, and as they are still from the beginning, they must be declared in the code, and can't change during the runtime.

This allows the program to catch certain types of errors early and identify them better, for example, mismatches in variable operations.

Because the compiler knows previously the type of all the variables, it can optimize the machine code (low-level code) to improve the performance. Even exist some commands that allow the program to infer the variable type, this is done at compilation time.

There are other languages like Python, JavaScript, Matlab, R, among others.

1.4 High-Level Language

Programming Languages are divided into low and high-level languages. A low-level programming language is very close to the computer language, such as machine code or assembly. On the contrary, a high-level programming language abstracts computer details from the programming procedure. C++ is considered a high-level language
However, this distinction is not strictly binary. Some languages are more or less close to these classifications. In this sense, among high-level languages, C++ is pretty low-level due to its ability to manipulate memory directly.

1.5 Compilation and run

Despite we are programming in a high-level language, the machine still reads machine code. The compilation is the process where our C++ code is "translated" into machine code.

This process is started by a compiler, a program that can write C++ code and write assembly code, a low-level and human-readable form of machine code that is specific to each processor architecture (This is also why the compiler should be specific for our machine).
Secondly, the assembler, another program, converts the assembly code into machine code, in form of 0's and 1's, that the CPU understands directly, this is stored in a .obj file.
Finally, the linker is the program in charge of combining all the single .obj files in an executable single program, that we will finally run as a result of our code. This is done because large codes are organized in libraries that provide functions and objects to the main file.

1.6 Scope

A scope is a group of code that works under the same conditions, variables, and functions. Most of the time, in C++ {} are used for delimiting groups of statements into a scope.
The scope of an identifier is the portion of the program in which the identifier can be referenced.

1.7 Memory

Computer's memory used by C++ is managed in defined blocks.

First, the executable code is stored in the code section. Here, the program stores every line we write, then it can compile the program and run it.

Secondly, there is the static data. Here, the program stores every global variable through the code (see section 2.1), the ones that are available everywhere. *These variables are a bad practice if they are not constants* .

In third place, there is the stack. Here, the program stores every local variable and function environment.

In the last place, there is the free store (or heap), which is reserved for the memory that the commands 'new' and 'delete' use (see section 7.5).

The stack and free store grow through the program executes. If we try to allocate more memory than is available, it will crash.

1.8 Indentation

Indentation is the formatting whitespace to the left of the code. In some languages, it allows to define scopes.

2 Basic commands

2.1 Variables

In C++, information is stored in different types of variables that have different purposes. Plus, variables can store diverse amounts of information, that configure a trade-off between the space used and the expressiveness of it.

There exist global variables, that are defined outside any function, and local variables,

that can be only used inside the scope they were declared.
Never use global variables, bad practice. Use them only if they are constant
Local variables also exist, which are defined inside a scope. When a local variable has the same name as a global variable, the local one will be used by the program in the related scope, this is called shadowing.

Variables have names that are used to identify them. They must start with a letter and contain letters, digits, and underscores only. They can't start with underscores since they are reserved for implementation and system entities. They are also reserved names that are system commands such as int, if, while, etc.

Variables have also a memory allocation size, that affects which range of values are possible to store inside them. It is recommended to be careful with the boundaries of the variables, since most of the time, if we exceed them, there won't raise an error, but our program will not perform as expected.

As C++ is a static language, variables need to be **declared**, which means to tell to the program what's the name and type of it. This will reserve a space in the memory for the data. After this, only operations defined by the specific type could be applied.
The syntax for it is as follows:

```
type name;
```

Nevertheless, most of the time we want to **assign** a value to the variable.
Global and static (see section 5.3) variables are initialized by their default value. On the contrary, local variables and not-static class members (see section 5.2) don't have a default value. If these kinds of variables are used without assigning any value, they could have stored anything inside (whatever is in the memory allocation).
The syntax for assignment is as follows:

```
name_var = value;
```

It is important to note that this creates a copy of value in name_var at the moment of the assignment. name_var is not attached to value, so if value changes, name_var will not do it necessarily.

Anyway, the declaration and assignment could be done in one line as follows:

```
type name = value;
```

Multiple declaration-assignations could be done as well in one line as follows:

```
type name1 = value1, name2 = value2;
```

Some of the most known built-in type variables are the following:

Type	Default Value	Value Size (bytes)	Value Range
`int`	0	4	$-256^4/2$ to $256^4/2-1$
`float`	0.0	4	-256^{38} to 256^{38}
`double`	0.0	8	2.2×10^{-308} to 1.7×10^{308}, including negatives
`char`	'\0'	1	0 to 256^1-1
`bool`	`false`	1 (in bits)	0 to 1
`short`	0	2	$-256^2/2$ to $256^2/2-1$
`long`	0	8	$-256^8/2$ to $256^8/2-1$
`long long`	0	8	$-256^8/2$ to $256^8/2-1$
`unsigned int`	0	4	0 to 256^4-1

Table 1: Type of data and ranges

short, long and long long are nicknames for short int, long int and lon long int.

There are other common ones that are provided by standard library:

- **string:** It stores sequences of characters, it is a container (see section 12). It is provided in the header <**string**> Unlike in the language C, strings are variable-length, so we don't have to worry about the length of them. The default value is "", an empty string. It dynamically allocates memory, then it doesn't have a fixed size. Unlike chars, there are used double quotes for delimiting strings. Note: There are special characters like "\n" that indicate a new line.
- **size_t:** It is an unsigned integer type that can store the number of the maximum size theoretically possible stored. It use at least 16 bits It is often used to make sure a loop variable does not go out of the size of the type of data of it. It is provided by the header <**iostream**>.

Let's see a declaration-assignation example:

```
std::string s1 = "Hello, world";
double u=7, v=12.2344;
```

We can also define type aliases, that are synonyms for another type. It is used especially when we have long original names. The syntax is the following:

```
typedef type alias;
using alias = type; // from c++11
```

Additionally, we can use the auto specifier. This is used to not explicit the type, and ask the compiler to deduce the type of the initializer based on what is assigned. It is important to initialize the variable in the same line than the declaration. This is done by the next syntax:

```
auto variable_name = initialized_value
```

2.2 Qualifiers

There are also exist type qualifiers, which are commands that provide additional information about the variable's behavior. The syntax is the following

```
type_qualifier type variable_name;
```

The type qualifiers are

- **const:** Indicates that the variable's value cannot be changed after initialization
- **constexpr:** Indicates that the variable is a constant expression, allowing the compiler to evaluate it at compile time
- **volatile:** Tells the compiler that the variable's value may change unexpectedly, often used in multithreaded programming

2.3 Function Matching

The function declaration inside the code must follow some rules, so the compiler can find always a best unique match for each call:

1. Two functions with the same name cannot differ only in their return type, they have to at least differ in the number or types of parameters.
2. If two functions have the same number of parameters, they can't be ambiguous between them. For example, an int and a double could be confusing to differentiate, since one can be converted in the other.
   Actually, it works while it does not have a problematic call, since the compiler tries to minimize the conversions. Anyway is a good practice to avoid those problems

It is allowed:

- To overload based on whether the parameters is a reference (or pointer) to the const or not-const version
- To overload based on the const or non-const version of a method member in a class (see section 5). It will match whether the object that calls is constant or not

2.4 Functions and Main

C++ uses functions to organize its code and to make it more scalable. Functions are pieces of code that receive inputs, perform certain tasks, and return an output. They are used for logically separate code, facilitating maintenance, and using the code more than once.
The syntax to declare a function is the following:

```
returned_type function_name(type1 input1, type2 input2, ...);
```

And the syntax to define a function is the following:

```
returned_type function_name(type1 input1, type2 input2, ...){
    code
    return output;
}
```

When more than 1 object needs to be returned, the function can change the values by reference (see section 7)
It is important to note that at the function call, C++ copies the value received in function parameters to the actual parameter memory location. So, each time we change an input parameter in the scope of the function, it will not change it outside of the scope.

When the function does not return something, it is called a procedure. Functions can change things outside them when we pass a reference (see section 7) to an outside-function object. This is recommended when the returned object is large, then the system does not have to copy the object, just work with the reference.
To implement a procedure, we use the next notation:

```
void function_name(type1 input1, type2 input2, ...){
    code;
}
```

We can add default parameters to the functions, that are used if we do not provide the respective parameters. The default parameters must be placed in the declaration only. The notation is:

```
void function_name(type1 input1, type2 input2, type3 input3=default_value1,
    type4 input4=default_value2,...);
```

Note that after a parameter with a default value, it cannot be placed a parameter without a default value.

Specifically, C++ uses the function main that returns an int to run it automatically when it compiles a file. Here we will put our main code, which should be run initially, and that will trigger the rest of the code.

```
int main(){
    code
    return 0;
}
```

By convention, it is used *return 0* to mark that the main function ran well.

Functions can be stored as variables with the header $< functional >$. Those are the objects that implement the 'operator ()' (see section 5.7). The syntax is as follows:

```
std::function<return_type (input_type1, input_type2, ...)>
```

This allows us to generalize functions that use other functions with the same domain and rank.

2.5 Libraries, headers, and namespaces

A library is a collection of precompiled code that implements code that can be used for different purposes. The standard library is the most common library in C++, and it is compiled automatically.

A header is a group of function declarations that can be included in the file, to declare the implementation of elements in a code. Some common headers in the standard library are string or vector.

A namespace is a named scope. It encapsulates certain definitions in order to avoid inadvertent collisions between the names we define or use in our code. Specifically, all the names in the standard library are in the **std** namespace

Objects in the namespace could be used with the following syntax:

```
namespace::object
```

We can also surpass this repetitive notation, but we have to be careful with the collisions in the names of the objects in our program, because it could lead to an unexpected behavior. It could be used as follows, but a lot of times is a bad practice.

```
using namespace namespace_name;
```

We can do the same with specific structures or functions as follows:

```
using namespace::obj_name
```

2.6 Input and Output

C++ does not define any statements for input or output, so they are managed by **iostream** library. They use the namespace **std**.
C++ manages the results of its instructions in buffers. A buffer is a zone of temporal memory where the data is stored to process it more efficiently. In terms of the output, it is used to group the data before printing it.
We list the most important functions:

- cout is used for printing in the console. It uses a buffer
- cin is used to ask for an input in the console. It stores in a variable until a space, tab, or linebreak is found. The input should be finished by the user with an enter. It uses a buffer, if we enter more characters than needed, the rest will be stored in the input buffer.
- endl is used for flushing the output buffer. In other words, it ensures that all the program's output has generated so far is written to the console.
- cerr is used for showing the error messages in the console. It doesn't use a buffer.
- clog is used for showing the logs messages, information about the execution of the program. it uses a buffer.

We see an example:

```
using namespace std;
cout << "text" << variable << endl;
cin >> variable;
std::cerr << "Error: Invalid input, please enter a number!" << std::endl;
std::clog << "Loop iteration " << i << std::endl;
```

An input could be read until an invalid value is entered with the next code using loops (see section 2.11).

```
int sum=0, value=0;
while (std::cin >> value){
    sum += value
}
```

This works because cin returns true or false whether the assignation was successful or not

2.7 Operations

Different types of data define their operators differently, this is called operator's overloading (see section 5.7). In the case of strings:

```
std::string c = a+b; // Concatenates a and b
c += d; // Equivalent to c = c+d;
// ++, - are not supported
```

In the case of integers and floating-points

```
\begin{lstlisting} [language=c++]
double c = a+b; // Sums a and b
double c = a*b; // Multiplies a and b
double c = a/b; // Divides a over b
double c = a%b // Modulus: The remain part of dividing a by b
c += d; // Equivalent to c = c+d
c++; // Uses the value in the operation and then adds 1 to the value
++c; // Adds 1 to the value and then uses the value in the operation
c--; // Uses the value in the operation and then subtracts 1 to the value
--c; // Subtracts 1 to the value and then uses the value in the operation

#include <cmath>
double a = sqrt(b); // Calculate the square root of the b
double c = pow(a,b); // Calculate a to the power of b
```

Mathematical operations follow the usual arithmetic order. ( ) parenthesis could be used.

As a general rule, when an operation is performed, the result keeps the types of the involved operands. In this sense, for example, if we perform a division between two integers, the result won't be the normal division, but it will be the integer division (// in some programming languages).

2.8 Type conversions

Mixing variable types has different behaviors depending on the types mixed.

Value assignment could be done from a value that does not match with the assigned variable, this is called implicit type conversion. In this case, a transformation occurs, that varies according to the case. A remarkable case is when we assign a double to an integer, in this case, the value assigned is the integer part of the double. There are other cases less useful, for instance, if we assign a negative value to an unsigned type, the value "wraps around", which means that it will start to subtract the negative value from the highest possible value in the type object.

The implicit conversion from char to numeric values consists of the ASCII character representation. To obtain a number, it is needed to subtract '0', as in the following example:

```
char a = '5';
int b = a - '0'; //b=5
```

In operations, if operands have different types, depending on the operation and involved types, the result is different. For instance:

- If we operate an integer with a double, the result will be double.
- if we operate an int with an unsigned int, the result will be an unsigned int, so we have to be careful if the result is negative

If we try to convert a string to a numeric type, it will be assigned the char conversion of the first character, having an unexpected behavior. To convert strings to numeric values, the standard library, in the $< string >$ header, provides stoX functions. These functions, convert strings to the type X. X can be chosen in the following list:

- i: int
- l: long
- f: float
- d: double

We can add a u to the unsigned version, or l for the long version (on ints, we omit the i on these types). See the following example:

```
// s is a string
unsigned num = std::stou(s);
```

These functions also work for char* type (see section 7).

To perform the inverse operation, convert a num type to a string, we can use the to_string function for all of them, as follows:

```
// num is a numeric type
std::string s = std::to_string(num);
```

2.9 Operators

Different types of values allow comparison between them. The usual operators are the following

```
== // Equal
!= // not equal
< // less than
<= // less or equal than
&& // logical and
|| // logical or
! // logical not
```

These operators have booleans as a result. That could be used for conditional (see section 2.10) and loops (see section 2.11).

2.10 Conditionals

Conditionals or selections are commands that allow performing certain lines of code only if a condition is met. It could be added also a second block of code that executes only if the condition does not meet. The syntax is the following

```
if (condition){
    code1
}
else{
    code2
}
```

For instance

```
if (a<b){
    max = b;
}
else{
    max = a;
}
```

When conditions several conditions are joined by an and, the following conditions will be evaluated only if the previous ones were true. This allows us to write conditions relying on assuming previous ones are true.
It could be used indentation instead of semicolons { }.

2.11 Loops

Usually, languages give tools to perform reiterative code. This means, doing a certain task more than once. In C++, there exist several commands that perform these kinds of tasks.

The first one is *for*. It is usually used to perform tasks a fixed number of times. The syntax is the following:

```
for (type iteration_var=initialization; termination_condition; var_update){
    code
}
```

See the following example:

```
int n = 25; sum = 0;
for (int i=0; i<n; ++i){
    sum += i;
}
```

The second command used is *while*. It is used when we want to perform a loop while a condition is true. The syntax is the following:

```
while (condition){
    code
}
```

See the following example:

```
int i = 0;
while (i < 50+3){
    std::cout << i << std::endl;
    i++;
    i *= 2;
}
```

The third command also uses *for*, but inside the parenthesis, it uses a different syntax. This is used to go over all the elements of an object (for example, a vector, see section 6), in this case, v.

```
for (int i: v){
    code
} // for each element in v
```

In this case, i is a copy of the element of v, but if we write:

```
for (int &i: v)
```

instead, i will be a reference (see section 7.4) to the object on v, and it will allow us to change v itself.

See the following example:

```
for (int &i : v) // for each element in v (note: i is a reference)
    i *= i; // square the element value
```

The fourth command is similar to while, but it checks the condition at the end of the code instead of the beginning

```
do {
    code
}
while (condition);
```

In all the cases we can add some statements that change the variable type, like *const* (see section 2.2).

Both for and while statements could be used with indentation instead of { } for simple code, but it is recommended to use { } in order to be as clear as possible.

3 Arrays and Structs

3.1 What is an array

An array is a built-in data structure that can store elements of a single data type.
They could be accessed by position using [ ] parenthesis, from 0 to the array size minus one.
They can store a maximum of a fixed number of elements. The dimension of it must be known at compilation time. To achieve this, we must use the "constexpr" type qualifier (sometimes "const" also works).
When we don't know exactly the number of elements, the usual way to use it anyway is over-estimate the dimensions.
It could be initialized empty as follows

```
constexpr std::size_t size;
// Assignment of value
type_elements name_array[size] = {}
```

It could be initialized with elements by extension as well as follows

```
type_elements name_array[size] = {elem1, elem2, elem3}
```

If we do the assignment at the same time we initialize the array, the compiler can just infer that the size is the number of elements we wrote on the code, without writing anything between [].
It could be initialized with "size" copies of an element as follow

```
type_elements name_array[size] = {elem}
```

All these three operations create copies of the elements
If we assign fewer elements than the array size, the rest of the elements will be values by default.

3.2 Memory allocation

The efficiency of arrays is given by the fixed space space they use, and the ease to access from outside, since the memory space is allocated contiguously, and therefore it is straightforward to find the element of a certain index in the array.

Actually, what C++ does is to assign a pointer (see section 7) of the address of the first element to the name of the array. This means that changes in the array elements within a function will be reflected outside the scope

3.3 Limitations

Arrays cannot be initialized as a copy of another array. They can be initialized as we have seen before, with { } initializer. It is also prohibited to assign one array to another. A copy needs to be performed element-wise, since the array itself is a pointer.

Plus, arrays cannot be compared through operators, they also have to be compared element by element

3.4 Multidimensional Arrays

An array element can contain other arrays, in order to create a matrix. The declaration could be done as follows:

```
type name[size1][size2];
```

The element could also be accessed with [ ]:

```
name[i][j];
```

For instance, we can create a matrix of ints:

```
int ia[3][4] = {
    {0, 1, 2, 3},
    {4, 5, 6, 7},
    {8, 9, 10, 11},
}
```

Being equivalent to do it without nested braces for each row. The compiler infers the positions:

```
int ia[3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
```

3.5 Structs

A struct is a structure to declare heterogeneous data The declaration could be done as follows:

```
struct StructName{
    type1 name1;
    type2 name2;
    type3 name3;
};
```

The initialization has to be done item by item using dot notation:

```
StructName s;
s.name1 = value1
s.name2 = value2
s.name3 = value3
```

We cannot compare operators (unless we provide the operator), but we can use the operator assignment, copying element by element:

```
Struct1 = Struct2
```

4 Declaration and Definition split

4.1 What are declarations and definitions

A declaration introduces a name into a scope, taking account the general structure of the elements. Some examples are:

```
double sqrt(double);// function body missing
struct Point; // members specified elsewhere
```

On the other hand, a definition is a declaration that also specifies the entity declared.

```
int a = 7;
int b; // an (uninitialized) int
double sqrt(double x) { ... }; // a function with a body
struct Point { int x; int y; };
```

As a general rule, things cannot be defined twice, but they can be declared twice.

4.2 Why declarations and definitions

To refer to something from another file it only needs to be declared, not defined. Then, we can build large programs without defining everything, just declaring our resources. This is useful when we want to write the code later or when that part of the could should be written by someone else. It is also useful to see what we have in a library without diving in the details of the implementation.

4.3 Code Organization

To place both declarations and definitions, C++ uses two types of files, header and source files. Header files are used to place the declarations of the code, they have a .hpp extension and are principally used for propagating declarations through code files. Source files, instead, are used to place the definitions of the code, they have a .cpp extension.
The command

```
#include "ExternalLibrary.h"
```

is used to copy the declaration of certain code to the current file.

4.4 Creating code files

A header file (.hpp) should declare everything we will define in our source file. Every declaration is often inside this statements

```
#ifndef NAME_LIBRARY
#define NAME_LIBRARY
// declarations
#endif // NAME_LIBRARY
```

that guarantee that header files will be included in the code at most once.
This is useful when we include certain declarations in a lot of files

On the other side, a source file (.cpp) should define the things we declared in the source file, then they will be actually available for external calls. To start the file, we have to include the header file:

```
#include "name_library.h"
```

Let's illustrate with an example:
myfriendlibrary.hpp:

```
#ifndef MY_FRIEND_LIBRARY_H
#define MY_FRIEND_LIBRARY_H
void bar(); //declaration
#endif // MY_FRIEND_LIBRARY_H
```

myfriendlibrary.cpp:

```
#include <iostream>
#include "MyFriendLibrary.h"
void bar(){//definition
    // Do something usefull
    std::cout << "MyFriendLibrary bar"
    << std::endl;
    return;
}
```

And then this function could be also used in other .cpp, including main:

```
#include <iostream>
#include "MyFriendLibrary.h"
int main() {
    std::cout << "main function" <<'\n';
    bar(); // here we use the library version
    return 0;
}
```

4.5 Including files

In practice, header files with their source codes are created to be used in other code files. To do this, just the header file should be added as:

```
#include "header.h"
```

5 Classes

5.1 What classes are

A class is a user-defined type that specifies how objects of its type must be created and used.

A class directly represents a concept in a program. In C++, it is a blueprint that defines the data members and the operations that an object supports. A member in C++ refers to any variable, function, or type that is defined within a class or struct.

On the other hand, an object is an instance of a class, a particular case. Each object has a class that states how the data is and how it behaves under certain circumstances and functions.

Classes are implementations of Abstract Data Types (ADT). Abstract Data types are data that have a domain and set of operations. It implements what the software does, how it is built, and the interface between it and its users. The instances show only their names, and the operation for manipulating the objects. On the contrary, they hide the type of structures and the operation's implementation from the user. Despite it is not seen, each object has states, that are values of the hidden data structure.

There is a common graphic representation for classes:

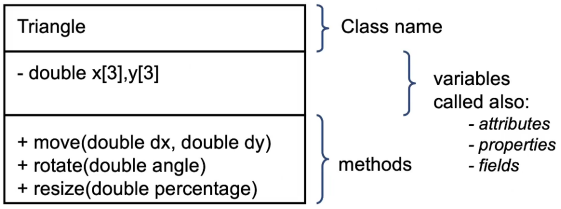

Figure 1: Classic graphic class representation example

5.2 Defining classes

The syntax to define a class is the following:

```
class class_name {
    // Data members
    // Function members
}
```

Data members are the data structures (any) that are loaded into the class. On the other hand, function members are the functions that are defined exclusively inside the class.

Each data or function member has to be defined as public, private, or protected. Public means that the member could be reached and used by any part of the program. Private means that the member could be used only inside the class. Protected means that only children (see inheritance, section 9) and the class itself can use the methods. This is done as follows:

```
class class_name {
    public:
        //  members
    private:
        // members
    protected:
        // members
}
```

It is a good practice to select carefully what are the members that we want the user to interact with, and only declare those members public. For example, it is a good practice to

keep all the private data members, and define setters and getters for interacting with them. This allows us to define **invariants**, rules that keep our members valid. If a class does not have invariants, probably it would be better defined as a struct For instancing a defined class, the syntax is as follows:

```
class_name instance_name;
```

To reach members outside the class, it is used dot notation, as follows:

```
instance_name.member_name;
```

On the other hand, if we have a pointer to the instance, it could be done as follows:

```
instance_name->member_name;
```

To reach members inside the class they could be called directly by their name. Let's see an example:

```
class X {
public:
    int m; // data member (usually a very bad idea to have public data member
    )
    int mf(int v) {
        int old = m;
        m=v;
        return old;
    }// function member
    }// Const function member
};

X var; // var is a variable of type X
var.m = 7; // access var's data member m
int x = var.mf(9);// call var's member function mf()
```

The object itself can also be referenced inside the class as 'this', using '$->$' to get members from it. This is used to differentiate between a member variable and a local variable with the same name. It is used when we want to return the object itself in the method.

```
    int mf2(int v, int m) const {
        int old = this->m + m;
    return old;
```

We can use 'const' command to indicate that the method does not change the object:

```
    int mf3(int v) const {
    int old = m + v;
    // m=v; // This would lead to an error, since we cannot change members in
     a const method
    return old;
```

If we call a method in a const method, the called method has to be declared as const as well.

5.3 Static Members

Static members are members that are shared among all the instances of a class. It can be even called without an instance, directly from the class. They are instanced as follows:

```
class ClassName {
public:
```

```
    static type data_member; // Static data member

    static type data_function() {
        // code
    }
};

type ClassName::data_member = value; // Static data member must be defined
    outside the class
```

As they don't belong to any instance, we can call members directly from the class with the following notation

```
ClassName::data_function()
```

The static members should be initialized in the .cpp file, they cannot be initialized in the .h file

These variables are stored along the global variables in the static data portion of the memory

5.4 Constructor

To define the initial state of objects in each class we must define a particular method that is automatically executed when the class is created, this method is called constructor. It is defined with the same class name and has no return type.

There exists a synthesized default constructor that receives no parameters and will be defined implicitly. Its behavior is to initialize the members as are initialized in-class. Otherwise, it default-initialize the member. Note that in case of having pointers, it is not recommended to use the default constructor. In case of having a reference, it is not possible to use it at all.

Despite being a default constructor, it could be defined a user-defined constructor as follows:

```
class_name(i_1, i_2, ..., i_n) member_1(i_1), member_2(i_3),..., member_m(i_e
    ) {
    // code
}
```

In case we want to define a constructor, the default constructor won't be created automatically, so if we want it to exist, we have to define it as well. Note that we can define multiple constructors depending on the parameters it receives. So the object can have different initializations depending on what parameters it receives.

It cannot be declared as const since it has to create the instance.

5.5 Destructor

The destructor operates inversely to the constructors and it is automatically invoked every time an object goes out of scope (then it is no more used). It has no return value and takes no parameters. Therefore, it is unique given a class. Destructors do whatever is needed to free the resources used by an object. Plus, the user could give an additional behavior. Anyway, even if we give it to it, the destruction itself is performed behind the scenes.

The destructor method has exactly the name syntax of the constructor but with a $\sim$ before, as follows

```
~class_name(){
    // code
}
```

One of the few cases where we overload the destructor is when we allocate memory in the constructor. Then, we must free that memory, usually in the destructor. Anyway, this kind of behavior can cause problems if we assign another pointer to the same allocated memory, because it would mean that we would free the memory twice. This can happen, for example, if we overload the assign operator (see section 5.8) with a copy to the same memory space.

5.6 Friends

A class can allow another class or function to access its nonpublic members by making it a friend, by the next syntax:

```
class ClassName{
friend returned_type function(parameters...);
}
```

5.7 Overloading Operators

We can use some commonly used characters to overload operations in classes. This can be useful to intuitively operate between instances of a class outside of it. This is done as follows:

```
// defining it inside the class: Here it does not have access to the private
    methods
bool operator<character>(const ClassName& a, const ClassName& b){
    // code
    return a.public_method() <character> b.public_method();
}
// defining it outside the class
// WARNING: Here we can access to the private methods of 'this', but not of
    other
bool operator<character>(const ClassName& other){
    // code
    return method() <character> other.oublic_method();
}
```

Then, outside the class we can just perform:

```
instance1 <character> instance2;
```

For instance, outside the class:

```
bool operator==(const Date& a, const Date& b){
    return a.year()==b.year() &&
        a.month()==b.month() &&
        a.day()==b.day();
}
bool operator!=(const Date& a, const Date& b){
    return  !(a==b);
}

void main(){
    // declarations and instantiation
```

```
    bool are_equal date1 == date2;
}
```

The usual operators that can be overloaded are +, −, =, + =, ∗, /, %, [], (), ˆ, !, &, <, <=, >=, >.
Note: We can't overload built-in operators, such as + in ints.
It is not recommended to overload ∗, &&, ||, and !, since they are natively used to other operations and then they are likely to fail.

5.8 Assignment

The assignment operator (or copy-assignment operator) is called when we define an object as follows:

```
type new_object = old_object
```

The default assignment copies all the values item-wise from the old to the new instance of the class. If some cannot be copy-assigned, the synthesized method won't be available. In the case of pointers and references, they will address the same object.
If we want to overload the copy-assignment operator, we use the following syntax:

```
MyClass& MyClass::operator=(const MyClass &other){
    // code
    return *this;
}
```

If we define the assignment operator, we have to define the copy constructor, which should behave very similarly, in order to keep the logic.

5.9 Copy

The copy operator (or copy initialization operator) is called when we define an object as follows:

```
type new_object(old_object)
```

The default member copies are performed as assignment operator The copy initialization is not only used in these cases, but also when:

- We pass an object as an argument to a non-reference parameter
- A function returns an object that is not a reference
- We brace initialize the elements in an array or the members of an aggregate class ({ })
- The standard containers are initialized when we insert or push a member

If we want to overload the copy constructor, we use the following syntax:

```
class MyClass{
public:
    MyClass(const MyClass& other){
        //code
    };
}
// As constructor, we can also use ':' syntax, leaving the body method empty
```

Unlike the default constructor, the copy constructor is synthesized even if we define other constructors. If some cannot be copied, the synthesized method won't be available.
When we use an object to initialize or insert an object in a container, a copy of the object value is placed, not the object itself, so if we change the variable outside, it won't affect inside.
If we define the copy constructor, we have to define the assignment operator, which should behave very similarly, in order to keep the logic.

6 Vectors

6.1 What is a vector

Vectors are a type of container (see section 12). A vector in C++ could be thought of as a variable-size array. They are part of the **standard** library, in the **vector** header.
The syntax to create a vector is the following:

```
std::vector<type_to_store> name_vector;
```

To add an element at the end of the vector it is used:

```
name_vector.push_back(element);
```

Let's see the following example:

```
#include <vector>
std::vector<double> temps; // declare a vector of type double to store
// temperatures     like 21.4
double temp; // a variable for a single temperature value
while (cin>>temp) // cin reads a value and stores it in temp
    temps.push_back(temp); // store the value of temp in the vector
```

We can define vectors of vectors as well, as the following example:

```
vector<vector<string>> names;
```

6.2 Structure

A vector is an object, where the first element is the size of itself, and the second one is a pointer (see section 7) to the head of the vector.
The size could be accessed by the following syntax:

```
vector.size();
```

It is often used for iterating through (i<vector.size();++i)
The pointer (see section 7) is pointing to the address of the first element of the vector (position 0), as shown in the example in figure 2.

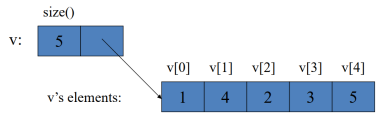

Figure 2: Vector Representation example

The elements part of the vectors are stored in arrays. As arrays have fixed size, when more space is needed, the vector automatically reallocates where it can place more memory, usually expanding the default size to double of the previous one. Although vectors use efficient strategies to choose the size of the vector. We can manually select the size of the array as well, with the following command:

```
vector.resize(size);
```

The access to the elements of a vector is managed by index. The syntax is the following:

```
vector[index];
```

See the next example:

```
vector<double> temps; // temperatures, e.g. 21.4
double temp;
while (cin>>temp)
    temps.push_back(temp); // read and put into vector
double sum = 0; // sums temperatures
for (size_t i = 0; i< temps.size(); ++i)
    sum += temps[i];
cout << "Mean temperature: " << sum/temps.size() << '\n';
```

There are also ways to access directly the iterators to the first and end of the array (see section 8).

```
vector.begin(); // Access the first element of vector
vector.cbegin(); // The same but returns a constant element
vector.end(); // Access the next position of the last element of vector
vector.cend(); // The same but returns a constant element
```

6.3 Initializations

1.
```
vector<T> v1;
```

   This kind of initialization initializes a v1 empty vector (default type elements) of elements of type T

2.
```
vector<T> v2(v1);
// or
vector<T> v2 = v1;
```

   This kind of initialization initializes v2 with an element-wise copy of v1

3.
```
vector<T> v3(n, val);
// or
vector<T> v3{n, val};
```

   This kind of initialization initializes v3 as n elements of value val

4.
```
vector<T> v4(n);
```

   This kind of initialization initializes a v4 empty vector (default type elements) of length n

5.
```
vector<T> v5{a,b,c,...}
// or
vector <T> v5 = {a,b,c,...}
```

   This kind of initialization initializes v5 as the elements between the { }

6.4 Operations

Let's see some of the most important extra vector operations:

- **v.empty()** returns true of v is empty (size=0), false otherwise.
- **v1==v2** v1 and v2 have the same number of elements and each element i is equal between the two vectors.
- **v1!=v2** not v1==v2.
- **¡, ¡=, ¿, ¿=** Order the vectors in dictionary order, looking over the elements in the order at the vector.
- **v.data()** returns a pointer to the vector internal array vector raw pointer (see section 7.5).

6.5 Efficiency

As vectors grow efficiently, it is often unnecessary to define a specific size, except when all the elements are of the same values, or we are sure about the size. In spite of this, we cannot subscript elements that are not initialized in a vector.
Inserting or deleting elements in the middle is expensive, but in the end is cheap.

7 Pointers

7.1 Memory Address Access

As we saw in section 2.1, each variable has a physical address associated, that is the place in memory where the data is stored.

The operator & allows obtaining the memory address of the location of that data. Note that it is not possible to obtain the allocation of literals, since they don't have a memory address assigned. The syntax to obtain this is the following:

```
&variable_name
```

The addresses of memory have to be stored in some place as well if we want to use it for our code. Redundantly, to store these addresses, there exists the pointer variable type.

These pointers usually require 2 or 4 bytes, depending on the architecture of the machine. Plus, they have to be pointers to a specific kind of variable. It is not possible to mix pointers to different types of data. This is why each pointer to each variable is a different kind of variable, i.e., a pointer to a string is a different type than a pointer to an int. The syntax to declare a pointer is:

```
type_data* pointer_name;
```

If we want to initialize a pointer without defining it, it could be done by assigning 'nullptr'. This allows verifying the pointer if it is not initialized as follows:

```
*pointer_name = nullptr;
// Code
if (pointer_name==nullptr){
    // Raise error
```

```
}
else{
    // Code
}
```

7.2 How do we use pointers

As we saw in section 2.4, input variables are received as copies from the call. Then, if we want a function to make changes over a variable out of the scope, we use pointers.

If we use a pointer as an input parameter, C++ will copy the value of the pointer, but that value will keep pointing to the memory address of the original variable, so we can make changes on the variable only with a copy of the pointer's value.
Let's see an example:

```
void sum(int* var, int add){
    *var += add;
}
a = 5;
b = 3;
sum(&a, b); // This will change a to 8
```

Note that if a pointer of a variable created inside the scope is returned, that pointer will still point to that part of the memory. That does not mean that the data remains there because, outside the function scope, that memory was released from the stack.

7.3 Operating with pointers

If we want to obtain the data the pointer is pointing to, this is called dereferencing. We use the next syntax:

```
*pointer_name;
```

The object stored in the address of a pointer can be in one of four states:

- It can point to an object
- It can point to the location just immediately past the end of an object
- it can be a null pointer
- It can be invalid

It is an error to try to access the value of an invalid pointer, worsening, the compiler is unlikely to detect this error.
A pointer could not be dereferenced if it is not attached to any object, even if we want to dereference it for assigning a value.
Note that the syntax for dereferencing is the same as when we are declaring a pointer.
Let's see an example:

```
int a=5;
int* p = &a;
int b = *p; // Dereferencing
*p = 7; // b now is 7
```

Addition and subtraction operators are implemented in pointers. When we sum or subtract n to a pointer, C++ moves to the pointer in n positions forward and behind, respectively.
Let's see an example

```
int* p;
// code, p is an array
b = *p // equivalent to p[0]
a = *(p+i) // equivalent to p[i]
```

Note that pointers can point to other pointers. this is simply declared with a double *.

7.4 References

References are automatically dereferenced pointers. Could be seen as an alternative name for an object. It has to be initialized in the next way:

```
type& name_reference = var_to_dereference;
```

The references must be initialized at the same time as the declaration as before. Plus, they cannot change after initialization, they always point to the same object.

References do not copy the values but are bound to the variables it was initialized with. This is the reason why they are recommended for large objects. They avoid the program copying them, wasting memory.

We can add qualifiers to the references, for example, if we don't want the variable to change by that reference. In case we are referencing a variable with an already declared qualifier, we have to necessarily declare it as well as follows:

```
qualifier type& name_reference = var_to_dereference;
```

7.5 Raw Pointers

We already have pointers accessing variables that already exist in the code. Raw pointers can reserve a memory's portion in the heap (or free store) before assigning anything. To reserve this space in memory, it is used:

```
type* var1 = new type; // It allocates one element of type
type* var2 = new type[size]; // It allocates size elements of type
```

Then, we should initialize this pointer before using it in any way.
To free the reserved memory, it is used:

```
delete var1 // It free var1
delete[ ] var2 // It free all the array var2
```

We have to be careful with freeing the memory we allocate because doing it incorrectly (not doing it actually) could lead to memory running out

7.6 Smart Points

Smart pointers are another way to reserve space in memory before assigning any variable to it. The difference with raw pointers is that they have a mechanism to free memory automatically.

Smart pointers implement a counter, that indicates how many variables are associated with the pointer, when the counter achieves 0, the heap memory is automatically released. This prevents the program from having memory leaks.

Smart pointers support indirection: $*$ and $->$ operators. Plus, it does not support $+$ or $-$ operators, to iterate over objects pointed by a smart pointer we have to rely on iterators There are two types of smart pointers in C++11:

- shared_ptr: Allows multiple pointers to refer to the same object
- unique_ptr. Owns the object to which it points. We won't dive into this

To define a shared pointer, we use the following syntax:

```
shared_ptr<type> p;
```

To allocate objects in shared pointers, we have to use the following syntax:

```
shared_ptr<type> p = make_shared<type>(args,...);
// args are the arguments given to the constructor of type
```

To create another pointer to the same object, we have to use the copy or assignment constructor:

```
shared_ptr<type> p = make_shared<type>(args,...);
auto q(p);
auto q = p;
```

Shared pointers have a counter, that is incremented when a new variable is associated with the pointer. The counter is decremented when we assign a new value to one of the pointers, or when the pointer is destroyed (e.g. it goes out of scope). Once a shared pointer counter goes to zero, the shared pointer automatically frees the object it manages. To access this count, we use the following syntax:

```
pointer.use_count();
```

Plus, we can use pointer.unique() to know if the counter is equal to 1.

The dereferentiation and member access are performed as raw pointers.

We can use p.get() to have the raw pointer that is managed by p. This is not recommended.

We can use the next syntax to swap pointers:

```
std::swap(p,q);
p.swap(q);
```

See the next example:

```
shared_ptr<int> p1 = make_shared<int>(42); // creates 42
shared_ptr<string> p2 = make_shared<string>(4, 'd'); // creates 'dddd'

shared_ptr<int> q1(p1);
shared_ptr<int> q2 = p1;

p1.use_count() // 3
p2.unique() // true
```

When we create containers with make_shared, they are not free until the counter of the shared pointers arrives at 0. This is a way to create objects in a scope that are not deleted until the last pointer is out of scope (or deleted or changed)

8 Iterators

8.1 Containers

Containers are objects from the standard library that contain other assignable objects. Assignable means that the object must be able to be initialized or changed after the declaration. Const objects (see section 2.2) and references (see section 7.4) are examples of non-assignable values.

Some containers, such as strings and vectors, can be subscripted. The subscript operator allows us to access a specific element in the container, that is stored by an index between 0 and size-1. The syntax to do this is the following:

```
container[index]
```

8.2 Iterators use

Iterators are used to access the container elements since subscription is not available for every container. Iterators allow us to access any container, having specific methods to deal with these kinds of objects. To declare them, the syntax is as follows:

```
type::iterator = iterator_name
type::const_iterator = const_iterator_name
```

Note that as pointers, const iterators are mandatory for const objects, but can also be used for non-const ones.

Very often, 'auto' notation is used to declare the iterators, so the code is more robust and flexible to changes in the container types.

Containers have methods that return iterators and help us access objects. The 'begin' and 'end' methods return iterators at the beginning and end of the container. They are used as follows:

```
auto beg = container.begin()
auto end = container.end()
// constant versions
auto beg = container.cbegin()
auto end = container.cend()
```

These iterators help us to iterate between them, where the object is stored. The syntax for doing it is as follows:

```
for (auto it=s.begin(); it != s.end(); ++it){
    element = *it
    // code
}
```

end method just holds an off-the-end space, the space that is just after the container. That is not dereferenceable, if we try to do it, we will have an undefined behavior.

There also exist reverse iterators, that address elements in the reverse order:

```
reverse_iterator c.rbegin(), c.rend();
const_reverse_iterator c.crbegin(), c.crend();
// Also the operations are reversed
```

8.3 Iterator operations

```
*iter // Returns a reference to the element denoted by the iterator iter
iter->memb // Dereferences iter and fetches the member memb from the
    underlying element
(*iter).memb // same as above
++iter // Increments iter to refer to the next element in the container
--iter // Decrements iter
iter1 == iter2 // Compare two iterators. To iterators are equal if they
    denote the same element or they are the off-the-end iterator for the same
    container
iter1 != iter2 // Negation of above
```

Iterators can be operated with algebra. For example, we can obtain the object in the middle with

```
v.begin() + (v.begin()-v.end())/2
```

8.4 $< ranges >$

Since C++20, the same iterators can be obtained through the $< ranges >$ library. The library also contains helpful functions for operators, for example:

```
#include <ranges>
ranges::sort(v);
```

9 Inheritance

9.1 What is inheritance

Inheritance provides a way to create a new class starting from an existing one. The new class is a specialized version of the existing class. This promotes code reuse and evolution. It can add both specializations and extensions.

An example of inheritance is the Animal class, which has certain behaviors, but we also have dogs, cats, lions, ants, and other classes, each one a *child* of Animal Class. They have specific behaviors in addition to the ones in Animal, that are shared.

9.2 What is inherited

C++ classes inherit both data and function members from the *parent*. Despite this, private members are not accessible from the derived class.

The derived object contains a subobject containing the non-static members defined in the derived class itself. Plus, it contains subobjects corresponding to each base class from which the derived class inherits.

In addition, each class contains its own static members, that are not duplicated among children either, they only belong to the parent. The children cannot access to the base constructor, assignment, friends, and destructor. Plus, static members are not inherited as well.

A child can add new data members and function members. Plus, it is able to overwrite the parent members.

Remind that 'protected' data is available for the children (see section 5). Members should be protected only if it is necessary for the derived class operation.

9.3 Create Inheritance

The notation to create a class child is the following:

```
// Suppose a created ParentClass
class ChildClass: public ParentClass
```

There also exists the private and protected inheritance, but we won't address them.

When we create a derived-class instance, the behavior is the following:

1. Space is allocated for the full object (base+derived class)
2. Base class constructor is called to initialize the parent part
3. The derived class constructor is called to initialize the rest of the members
4. The derived-class instance is usable

The destructor methods are called in the reverse way.
When we want to define a constructor in the derived class, the syntax is the following:

```
DerivedClass(parameters1,...): BaseClass(paramters), data1(parameters2),
    data2(parameters3){}
```

Note that depending on the parameters that we put on the base class, is the constructor that is going to be called on it. If we don't call it, the default constructor will be called anyway automatically. In this case, if the default constructor does not exist, C++ will raise an error.

9.4 Static and Dynamic Type

Because the derived class contains its base classes, we can bind a derived object reference to a base class reference, as follows:

```
BaseClass base;
DerivedClass deriv;
BaseClass *p = &base;
p = &deriv; // p now points to the base part of Derivedclass
BaseClass &r = derived; // The same as above but with a reference
```

This is an exception to the rule that we can only bind a pointer or reference to the same type.

This implies that when we use a reference or pointer to a base-class type, we don't know the actual type of the object. It could be the base type, or an object of a derived class.

In this sense, when we manage a variable, we have to distinguish between its:

- **Static Type:** It is the type with which a variable is declared. It is known at compilation time
- **Dynamic Type:** It is the type that the variable actually represents. It could not be known until run time

This is useful, for example, because we may want to create containers (see section 12, a vector is an example) that store objects that are across an inheritance relation. We then are able to use pointers, although smart pointers are preferred, to store them easily, as the next example:

```
vector<shared_ptr<Parent>> cont;
cont.push_back(make_shared<Parent>(arguments));
cont.push_back(make_shared<Child>(arguments));

cout << cont[1]->function(arguments) << endl;
```

This will call the child function (if it was overridden), even if we are storing parent objects.

9.5 Abstract Base Classes

Pure virtual functions are those that have no function definition in the base class, and therefore must be overridden in a derived class if we want to create objects. They are defined as follows:

```
virtual type f() = 0; // We add =0
```

The **Abstract Base Classes** are the classes that have at least one pure virtual function. These classes cannot have any objects, and their function is to be a basis for derived classes only.

10 Polymorphism

10.1 The concept

It is the ability of objects to respond differently to the same message or function call.
There are two types:

- Compile-time Polymorphism
- Run-time Polymorphism

A subclass can overwrite a base class method behavior by:

- **Overloading:** Defining functions with the same name but different parameters. This is done at compilation time.
- **Redefining:** Defining a function with the same name and parameters, but in a child class. This is done at compilation time.
- **Overriding:** Using a different function depending on the circumstances. This is done at run time.

The last one is the most powerful mechanism.

10.2 Overriding

Overriding is used when the parent marks that a method can be changed in the child interface, changing also the parent method in that instance. This is marked as follows:

```
virtual type method(parameters){
    // code
}
```

We have to use the command in the declaration only (i.e., only in the .h file), but not in the definition of the function. Plus, a function that is virtual in the parent is implicitly virtual in the child.

Finally, whenever we define a virtual method, we must explicit a virtual destructor as well. This is because it allows objects in the inheritance hierarchy to be dynamically allocated. The most common way is as follows:

```
virtual ~ClassName() = default;
```

Then, to modify this method, the child class should use the following syntax:

```
type method(parameters) override;
```

The unique case where the return type of a virtual and override function cannot match exactly, is when they return a pointer to themselves. In that case, the parent class can return ParentClass* and the child class ChildClass*

10.3 Use case

Define a base class

```
class Quote {
public:
    Quote() = default;
    Quote(const string &book, double sales_price):
        bookNo(book), price(sales_price) { }
    string isbn() const { return bookNo; }
    // returns the  total sales price for the specified number of items
    // derived classes will override and apply different discount
    // algorithms
    virtual double net_price(size_t cnt) const
    { return cnt * price; }
    // dynamic binding for the destructor
    virtual ~Quote() = default;
private:
    string bookNo; // ISBN number of this item
protected:
    double price = 0.0; // normal, undiscounted price
};
```

Derived Class

```
class Bulk_quote : public Quote {// Bulk_quote inherits from Quote
public:
    Bulk_quote() = default;
    Bulk_quote(const string &book, double sales_price,
        size_t min_qty, double disc_rate);
    // overrides the base version in order to implement the bulk
    // purchase discount policy
    double net_price(size_t cnt) const override;
private:
    size_t min_qty = 0;// minimum purchase for the discount to apply
    double discount = 0.0; // fractional discount to apply
};
```

External function: Polymorphic behavior is only possible when an object is passed by a reference or a pointer

```
double print_total(const Quote &item, size_t n)
{
    // depending on the type of the object bound to the item parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    cout << "ISBN: " << item.isbn() // calls Quote::isbn
    << " # sold: " << n << " total due: " << ret << endl;
    return ret;
}
```

Calls

```
// basic has type Quote; bulk has type Bulk_quote
print_total(basic, 20); // calls Quote version of net_price
print_total(bulk, 20); // calls Bulk_quote version of net_price
```

Further, if we pass a derived object to a function that receives the parent object (a reference), instead of raising an error, the function will use the base object stored inside the derived object.

11 Streams & I/O

11.1 Dealing with files

At a fundamental level, a file is a sequence of bytes numbered from 0 upwards. Standard library, in the header '$< fstream >$' allows the program interact with the files. We have to give it an string to indicate it the file name to be opened. The objects in this library do not accept the copy or assignment operator. In addition, they pass the stream through references, so they should be passed by reference and cannot be constant, since when it pass the information of a file, it changes its state.

11.2 Reading

We can read a file. In this case, we have to open it for reading. The syntax for doing this is the following:

```
ifstream file_variable {file_name};
```

Then, we can read from the file with the following syntax:

```
while (file_variable >> variable1 >> variable2 >> ... >> variablen){
    // store the variables
}
```

Where C++ will store data in a single variable until it finds a whitespace delimiter (such as spaces, tabs, or newlines). The condition will be true until the program finds a end-of-file, or it is not capable of matching the respective data with the variable type placed (that of course we have to declare first).

11.3 Writing

We can also write a file. In this case we have to open it for writing or create a new file. It erases what is in the file and starts to write from 0 The syntax for doing this is the following:

```
ofstream file_variable {file_name};
```

Then, we can write to the file with the following syntax:

```
while (file_variable << variable1 << variable2 << ... << variablen){
}
```

We can check if the file was successfully opened using '*file_variable*' as a bool. If it is not, the object is automatically destroyed

11.4 Alternative modes

There are specific modes that can be used in fstream functions, from the header '< *ios* >'

- **ios_base::app:** Append, i.e., add the content at the end of the file keeping the current content
- **ios_base::ate:** 'at end', open and seek to end
- **ios_base::binary:** Binary mode
- **ios_base::in:** for reading
- **ios_base::out:** for writing
- **ios_base::trunc:** Truncate file to 0-lenght. It erases what was first in the file. It is the default mode of ofstream

Those arguments could be given as a second argument to the writing and reading functions, as the next example:

```
ofstream ofs{name, ios_base::app};
```

Further, we can create objects for both, reading and writing, as the next example:

```
fstream fs {"myfile", ios_base::in | ios_base::out};
```

11.5 String Streams

String Stream allows us to read (and write) from (and to) a string rather than a file. This allows us to use the string as a stream and to handle the files in a easier way.
The functions are in the '< *sstream* >' header, and the syntax to read a file is the following:

```
istringstream stream_variable{string_variable};
```

We can then convert this variable, if possible, to another type as follows:

```
type converted_variable;
stream_variable >> converted_variable;
// it can be done with more than 1 variable following the same syntax,
    spliting by blank spaces
if (!stream_variable){//error}
```

Plus, we can use the function getline from the header '$< istream >$' to have a whole line in a string stream and facilitate the handling of the data. It get the data until certain character, by default '\n'. The syntax is as follows:

```
getline(stream_string, variable_line);
```

If we want it to be split by a specific character:

```
getline(stream_string, variable_line, '<character>');
```

Once we achieve the end of the stream, the function will return an (end of file) EOF, which can be used as a loop termination condition.

There also exists ostringstream, but we will not see it here.

12 Containers

12.1 Introduction

Containers are objects that are used to contain any specified built-in type and user-defined type that supports some elementary operations (copying and assignment). They are provided by the Standard Template Library, and each one is defined in a header with the same name as the type.

Containers store copies of the object values provided, because their arguments are not provided by reference.

Container classes share a common interface, that each one extends in its own way. Each container offers a different set of performance and functionality trade-off that makes them more suitable to only certain situations.
They are divided into three groups:

- **Sequential Containers:** They have an internal order in which the elements are stored and accessed. This order is position-depended
- **Associative Containers:** Store their elements based on a different key from the value. They are divided into:
  - **Ordered**
  - **Unordered**
- **Adaptors:** Created on top of pre-existing sequential containers

12.2 Sequential Containers

They provide fast sequential access to their elements. However, the costs to add or delete are not always that good. In addition, the costs to perform non-sequential access are high. The classes that belong here are:

- **Array:** Already seen. Fixed-size array. Supports fast random access. Cannot add or remove elements
- **Vector:** Already seen. Flexible-size array. Supports fast random access. Inserting or deleting elements other than the back is slow

- **String:** Specialized container for characters only. Similar to vector
- **Deque:** Double-ended queue. Supports fast random access. Plus, it has fast insert/delete at the front and back
- **List**: Double linked list. Supports only bidirectional sequential access. It has fast insert/delete at any point
- **Forward_list:** Singly liked list. Supports only sequential access in one direction. It has fast insert/delete at any point

12.2.1 List and Forward List

A linked list consists of elements that have pointers to the next one. The linked list itself has access only to the beginning (or also the end) element of the list. This means that to access an element a, we have to search element by element, through the pointers, the desired one. Once we have identified a, it is cheap to insert an element after it, since we only have to change the pointer from a headed to the new element and create a pointer from the new element to the next to a. Forward List implements a singly-linked list, while List implements a doubly-linked list, which uses the same mechanism but from both the beginning and the end (see figure 3)

Figure 3: Doubly-linked list representation

To initialize the lists we have to use the following notation:

```
std::forward_list<type> variable;
std::list<type> variable;
```

12.2.2 Deque

Besides the implementation being a little different, deques are similar to vectors. Containers have supper fast random access, but slow insertion in the middle. Plus, it has fast average insertion at the end and the beginning (unlike vectors). It has slow worst-case insertion anyway.
Deque organizes data in chunks of memory referred to by a sequence of pointers.

```
std::deque<type> variable;
```

12.2.3 Operations

Some specific sequential container functions are:

- **c.push_back(t):** Creates an element with value t at the end of c
- **c.emplace_back(t):** Same as above but emplacing
- **c.push_front(t):** Creates an element with value t at the front of c

- **c.emplace_front(t):** Same as above but emplacing
- **c.back():** Returns a reference to the last element in c. Undefined if empty
- **c.front():** Returns a reference to the first element in c. Undefined if empty
- **c[n]:** Returns a reference to the element indexed by n
- **c.at(n):** As c[n], but checks if out of range
- **c.pop_front():** Removes the element at the front of c
- **c.pop_back():** Removes the element at the end of c
- **c.resize(n):** Resize c so it has n elements. if $n < c.size()$, the rest of elements are discarded. Otherwise, elements are default initialized. Only for arrays, vectors, and deque
- **c.pop_back():** Removes the element at the end of c

We have to be careful with operations because they can invalidate pointers, references, and iterators. In the case of iterators, they can point to other unexpected values if we don't consider operations changes

Some specific sequential container functions based on assignment operator are:

- **seq.assign(b, e):** Replace elements in sequence with those in the range defined by b and e. b and e can't be iterators belonging to seq
- **seq.assign(i1):** Replaces elements in seq with those in the initializer list i1
- **seq.assign(n,t):** Replaces elements in seq with n elements with value t

12.3 Associative Containers

Elements in associative containers are stored and retrieved by a key that is constant. Therefore, elements do not support position operations based on the value.
Iterators iterate across associative containers in an ascending key order.

12.3.1 Ordered

Keys must be compared, so they rely on the $<$ operator of the object. Anyway, we can supply our own $<$ operator (if it meets the order properties). Map A map is a collection of $< key, value >$ pairs with unique keys. It is often referred to as an associative array, which is like a common array but its subscripts do not have to be integers. Values in a map are found by a key. To initialize a map, the syntax is the following:

```
std::map<type_key, type_value> map_variable;
```

We can also subscript map values based on the key of each one, with the next syntax:

```
c[k];
c.at(k);
```

In the case of the bracket operators, it value-initializes the key if it does not exist. Indeed, to change the value in the key, or create a pair key-value if the key does not exist, we can use the following syntax:

```
map_variable[key] = value;
```

Map iterators reference to pair types (see section 12.6), so we can use its interface. The map is implemented by red-black trees, a self-balancing binary search tree. Its algorithm is optimized so insertion and deletion are $\log n$ at the worst case. Set A set is a simple collection of objects. A set is most useful when we simply want to know whether a value is present in the structure.
To initialize a set, the syntax is the following

```
std::set<type_value> set_variable;
// It can be list-initialized with { }
```

We can add elements to the set with insert, as follows

```
set_variable.insert(element)
```

Plus, we find objects as follows:

```
set_variable.find(element);
```

If it does not find the object, the result will be equal to the end iterator 'set_variable.end()'
Through iterators, as the values are the keys, elements in a set cannot be changed.
Sets are also implemented by red-black trees.

12.3.2 Multi Types

Multi types are the same as the normal ones but they allow to store values with the same key. They just store the values contiguously. These types do not support the .at() and [] operators, since it can have more than 1 element with a determined key.
Both map and set have their multi-version, they are initialized as follows

```
std::multimap<type_key, type_value> map_variable;
std::multiset<type_value> set_variable;
```

To find every value with a key, can be used the following syntax

```
auto range = multi_variable.equal_range(key);

for (auto it = range.first; it != range.second; ++it) {
    std::cout << it->second << '\n';
}
```

12.3.3 Unordered

The classes that belong here are the same as in the ordered ones, but with an 'unordered-' precedent. Every type has the same structure as its ordered sibling but with a hash function that replaces the red-back tree order.
The hash function maps the keys of the maps to shared buckets, where all the elements mapped to the same hash are stored.
In this case, all elements with the same key will be in the same bucket necessarily.
The performance of finding an element is O(1) in average, but O(N) in the worst case
To declare the variables we just have to add the unordered_ word before, as follows:

```
std::unordered_map<type_key, type_value> map_variable;
std::unordered_set<type_value> set_variable;
std::unordered_multimap<type_key, type_value> map_variable;
std::unordered_multiset<type_value> set_variable;
```

12.3.4 Operations

Some operations that can be done across associative containers are

- **c.insert(v):** Insert in a map or set only if an element with the key already is not in c. Return a pair of an iterator referring to the element with the given key and a bool indicating whether the element was inserted. For multi structures, it returns an iterator to the new element
- **c.emplace(args):** The than insert but emplacing
- **c.insert(b, e):** b and e iterators denote a range of c::value_type elements
- **c.insert(il):** il could be a braced list of values. It will insert only the possible elements according by container rules
- **c.insert(p, v):** Like the first one, but uses p as a hint for where to begin to search the element. Returns an iterator to the element with the given key
- **c.emplace(p, args):** As above but emplacing.
- **c.erase(k):** Removes every element with key k from c. Returns size_type indicating the number of removed elements
- **c.erase(p):** Removes the element denoted by the iterator p. Returns an iterator to the element after p
- **c.erase(b,e):** Removes elements in a range from iterator b to e.
- **c.find(k):** Returns an iterator to the first element with key k, or off-the-end iterator if k is not in the container.
- **c.count(k):** Returns the number of elements with key k.
- **c.lower_bound(k):** Returns an iterator to the first element with a key not less than k. **c.upper_bound(k):** Returns an iterator to the first element with a key not greater than k.

12.4 Adaptors

They are interfaces created on top of a limited set of functionalities of pre-existing sequential containers. When you declare the container adaptor, it is needed to specify which sequential container to use as the underlying container.

- **Stack:**
  Provides last-in, first-out (LIFO) access.
  Elements are removed (pop) in the reverse order they are inserted (push). To retrieve the next element, it is used 'top()'.
  It usually goes on top of a deque

- **queue:**
  Provides first-in, first-out (FIFO) access. To retrieve the next element, it is used 'front()'.
  Elements are removed (pop) in the same order they are inserted (push).
  It usually goes on top of a deque

- **priority queue:**
  Provides sorter-order access to elements. To retrieve the next element, it is used 'top()'.
  Elements are inserted (push) in any order and then are removed (pop) based on the 'highest priority'.
  It uses a heap, which in turn is array-backed. Usually goes on top of a vector.

12.5 Complexities

We must choose the most suitable container in terms of complexity according to the situation, see table 2

Container	Insert	Find	Delete
`list/forward_list`	$O(1)$	$O(n)$	$O(1)$
`set/map`	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
`unordered set/map`	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$

Table 2: Comparison of time complexities for containers. In some cases: <average case> or <worst case>

12.6 Pair Type

It is a type that is used in structures with a key-value format. it has two public members called first and second.
They can be initialized as follows:

```
pair<type1, type2> var;
pair<type1, type2> var(val1, val2);
pair<type1, type2> var={val1, val2};
make_pair(v1, v2); // here, types are inferred from v1 and v2
```

Plus, its elements can be reached as follows:

```
var.first;
var.second;
```

From C++17, it is also available an automatic binding as follows:

```
auto [var1, var2] = var;
```

They accept relational operators. It checks first and if elements are equal second.

12.7 Types and Operations

Some common operations among containers are:

- **C c:** Default constructor

- **C c1(c2):** Default copy constructor
- **C c(b, e):** Copy elements from the range denoted by the iterators b and e
- **C c{a,b,c,...}:** List initialize c
- **c.size():** Number of elements in c (no for forward_list)
- **c.max_size():** Maximum number of elements c can hold
- **c.insert(args):** Insert args by copy constructor in c. Does not work for forward_list
- **c.emplace(inits):**Insert args by default constructor in c
- **c.erase(args):** Remove elements by iterators or by providing value/key
- **c1.swap(c2):** Using the assignment operator, exchange elements in c1 with those in c2. They must have the same type. It has O(1) complexity. Iterators, references, and pointers in the containers are still pointing to the original data, they don't swap
- **==, !=:** Equality value per element
- **¡, ¡=, ¿, ¿=:** Relationals (no for unorder associative containers). If all the elements are equal, the shorter one is smaller.

Some common container types are:

- **iterator**
- **const_iterator**
- **size_type:** Unsigned int large enough to hold the largest container size
- **difference_type:** Int large enough to hold the distance between two iterators
- **value_type:** Element type
- **reference:** synonymous for value_type
- **const_reference:** As reference but constant

13 Parallelization with MPI

13.1 Parallelization

One way to run applications faster is to use multiple computers/cores to solve a particular task, coordinating their computational efforts.

Parallelism is applying multiple processing units (PUs) to a single problem. It decomposes the computation into many pieces and assigns them to different processing units.

A parallel computer (system) is a computer (system) that contains multiple processing units. Each PU works on its section of the problem, and they can exchange information between them.

Anyway, not every algorithm can be parallelized. Indeed, algorithms can be parallelized only in certain parts, they have serial and parallel sections. Serial sections, where work cannot be divided, limit the parallel effectiveness.

The speedup is the ratio of the time required to run a code on one processor to the time required to run it in N processors.

Amdahl's Law, gives theoretical limits to the speed up we can achieve in a task:

$$t_n = \left(\frac{f_p}{N} + f_s\right) t_1$$

$$\implies S = \frac{1}{\frac{f_p}{N} + f_s}$$

where:

- f_s = Serial fraction of code
- f_p = Parallel fraction of code
- N = Number of processors
- t_n = Time to run the task in N processors
- S = Speedup (t_1/t_n)

This implies that it only takes a small fraction of serial code to degrade the parallel performance.
Even further, it does not consider the parallelization overhead:

- Costs of interprocessor communication.
- Load Imbalance: It is not always to divide the work evenly, so usually some processors have to just wait for the rest to finish.
- Extra computation: Other computations we have to do due to implement parallelization.

Load Imbalance, and Then, the performance degradation is even worse.

13.2 Flynn's Taxonomy

A stream means a sequence of items (data or instructions).
The Flynn's Taxonomy defines the types of parallel architecture, based on the number of instruction streams and data streams.

	Data stream	
2∗**Instruction stream**	SISD	SIMD
	(MISD)	MIMD

- **SISD:** Single Instruction, Single Data. Sequential Processing
- **SIMD:** Single Instruction, Multiple Data. Different data are dealt by different PUs, but with the same instruction.

- **MISD:** Multiple Instructions, Single Data. Does not exist, listed for completeness
- **MIMD:** Multiple Instructions, Multiple Data. Most common and general parallel machine. Multiple Instructions are given to different machines with different data

The issues we can face using parallel computing are:

- Data distribution and dependency
- Synchronization
- Communication costs

The memory in MIMD systems is divided in two:

- **Shared Memory:** Single Address space. All processors have access to a pool of shared memory. Processor-to-processor data transfers are done using shared areas in memory. It has scalability limits.
  It has two methods for accessing memory.
  - Bus, which consists of a unique space where processors place the information, but when one processor is writing in memory, the rest have to wait until it is free to write on it. Then, the bus is in charge of sending the data to the memory.
  - Crossbar, which consists of multiple spaces shared by a few processors, that then send the information to another space in memory, that writes on a memory divided into banks (slices of memory).
- **Distributed Memory:** Each processor has its own local memory. It must pass messages to exchange data between processors. It has a high scalability, but load balancing issues exist and I/O is difficult.

Parallel execution uses both, local variables for each processor and global variables where every element writes.

13.3 MPI introduction

MPI is the acronym for Message Passing Interface. It executes in both shared and distributed memory. Anyway, it was originally designed for distributed access

In message-passing programs, a program running on one core is usually called a process.

In MPI, parallelism is explicit, what each process processes is defined by the programmer.

Two processes can communicate by calling functions: One process calls a send function and the other matches it by calling the receive function.

To run MPI, it is needed to pass some values from the command line to C/C++ programs when they are executed. This is relevant to avoiding 'hardcoding' in the code. Specifically, the arguments are passed as char*, and are the following:

- **argc:** It is the first one and refers to the number of arguments passed (the id=0 is always the executable path).
- **argv[]:** Is an array of pointers, which points to each argument passed by the program.

Then, the main function should receive them as follows:

```
int main(int argc, char* argv[]){
    // code
    return 0;
}
```

To compile the program, the command used is the following:

```
g++ --std=c+23 file_name -o executable_name
mpicxx --std=c+24 -o mpi_file main.cpp
```

Then, to run the executable:

```
mpiexec -n number_of_processors executable_name
```

We can add an oversubscribe flag as well to emulate more processors than we have, as follows:

```
mpiexec -n -oversubscribe number_of_processors executable_name
```

To use the MPI functions, we have to always state in the main (and only in the main, or in the unique file where we use MPI) an environment that allows us to use them. MPI_Init tells the MPI system to do all the necessary setup: allocate storage for message buffers and decide which process gets which rank. It returns an int error code, but we won't dive on it. MPI_Finalize() deallocate all this memory.

It is done with the following syntax:

```
#include <mpi.h> // import header
int main(int argc, char* argv[]){
    MPI_Init(&argc, &argv);
    // code
    MPI_Finalize();
    return 0;
}
```

Plus, to let know the program what process is running and how many of them we have, we have to define the variables with the next syntax:

```
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
// Now size contains the total of processors
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Now rank contains the number of the process that is running, starting from
     0 to size-1
```

This has to be stated in any function we want to use the variables, not only in the main.

Definitively, rank is used to state C++ conditions on which process we want to use in certain task. For example, if we want the process 0 to do something and the rest something else, we have to use:

```
if (rank==0){
    // 0 code
}
else{
    // rest code
}
```

13.4 Point to Point Communication

MPI processes can be addressed via communicators. A communicator is a collection of processes that can send messages to each other.

The standard provides mechanisms for defining our communicators. Anyway, there's one that is predefined and collects all the processes: MPI_COMM_WORLD.

Point-to-point means that we explicitly state which among the communicator's processes we want to reach.

The syntax for sending a message from one process to another is the following:

```
MPI_Send(buf, max_length, MPI_TYPE, DEST, TAG, COMMUNICATOR);
```

where:

- buf: is a pointer or reference to what we want to send.
- max_length: is what's the length of the message. Be careful sending the appropriate amount of characters.
- MPI_TYPE: is the type of data we send.
- DEST: is the rank of the destination process.
- TAG: is an int-mark that gives additional information. It is used to distinguish messages traveling on the same connection.
- COMMUNICATOR: is the communicator used for the communication. Commonly used MPI_COMM_WORLD.

If a message is sent, the other rank has to match it to receive it. The syntax for receiving a message is the following:

```
MPI_Recv(buf, max_length, MPI_TYPE, SOURCE, TAG, COMMUNICATOR)
```

where:

- SOURCE: is the rank of the remittent process.
- max_length has the same meaning as the Send. It should be longer in the receiver than the sender to match the signals.
- The rest have the same meaning as MPI_Send, and the parameters should match exactly to complete the communication successfully.

Data types we can use in MPI are the following:

- MPI_CHAR
- MPI_SHORT
- MPI_INT
- MPI_LONG
- MPI_FLOAT
- MPI_DOUBLE

- MPI_UNSIGNED_CHAR
- MPI_UNSIGNED_SHORT
- MPI_UNSIGNED
- MPI_UNSIGNED_LONG
- MPI_LONG_DOUBLE
- MPI_BYTE

Messages have no order between different senders and receivers. Anyway, messages do keep the order if they go from the same sender to the same receiver.

We have to be careful to avoid deadlocks. When a Send or Receive is called, the process blocks communication, so if both processes, sender and receiver are blocked, the program will experiment a deadlock.

13.5 Broadcast

Most MPI implementations allow the process 0 to receive the standard input. The common practice then is to receive the information in 0 and then distribute it to all the processes. The collective routine involves all the processes in a communicator, matching them synchronously.
To deliver an exact copy of the data from a root, the syntax is the following:

```
MPI_Bcast(buf, max_length, MPI_TYPE, root, COMMUNICATOR)
```

where:

- root: is the source of the message
- The rest of the parameters mean the same as in send/recv

Note that even the max_length must be broadcasted before broadcasting the actual data we want to work with.

13.6 Reduce

When calculating a process by parts, it is necessary to aggregate the results to obtain the final result. This could be done by sending all the results to one process so it aggregates them. Anyway, this is inefficient, since that process takes all the work of the aggregation, while the rest are without work.
A smarter solution would be to aggregate the results incrementally, using different processes to calculate each of these each time more aggregated results. This is cumbersome to implement, so MPI offers a function that not only collects all the data but aggregates it. This function is called Reduce and is implemented as follows:

```
MPI_Reduce(sendbuf, recvbuf, max_length, MPI_TYPE, MPI_OP, DEST, COMMUNICATOR
    );
```

where:

- sendbuf is the buffer that will send the information

- recvbuf is the buffer that will receive the information
- MPI_OP is the aggregation operation that will be applied
- DEST is the process that will receive the final result
- The rest of the parameters are the same as in send/recv

Further, if we want this result available for every process, we have to rely on Allreduce. it does the same than Reduce but it stores the result in all the processes in the communicator. The syntax is as follows:

```
MPI_Allreduce(sendbuf, recvbuf, max_length, MPI_TYPE, MPI_OP, COMMUNICATOR);
```

In a collective communication we want to store the results in the same variable we are operating with, we can change the send buffer to the special placeholder MPI_IN_PLACE, and the information will be stored in the same variable. As an example:

```
MPI_Allreduce(MPI_IN_PLACE, &suma, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```

It is forbidden to put explicitly the same variable in the send and receive buffer.
In case using MPI_IN_PLACE in MPI_Reduce, the statement should be placed on the root only. In other ranks, replace with a nullptr. Data operations we can use in MPI are the following:

- MPI_MAX
- MPI_MIN
- MPI_SUM
- MPI_PROD
- MPI_LAND: Logical AND
- MPI_BAND: Binary AND
- MPI_LOR: Logical OR
- MPI_BOR: Binary OR
- MPI_LXOR: Logical XOR
- MPI_BXOR: Binary XOR
- MPI_MAXLOC: Returns the maximum and the process that had that value
- MPI_MINLOC: Returns the min and the process that had that value

13.7 Data Partition

When we have different processes to compute a single task n times, we usually divide the computation along the processes. We often compute $n\_local = n/size$ Data can be distributed among the processes in different ways:

- **Block Partitioning:** It consists of giving to the process rank the elements in the range: $[rank * n\_local, (rank + 1) * n\_local[$.
  It is used when data source is available on a single process.

- **Cyclic Partitioning:** It consists of giving the elements that meet $element \ mod \ (n\_local) = rank$ This is done by the next loop:

```
    for (unsigned i=rank; i<v.size(); i+=n/size){
        // use v[i]
    }

```

  It is used only when the data source is already available across all processes.

To implement Block Partitioning, MPI has a function that allows us to send the partition data to each process in the communicator. The function is called MPI_Scatter. The syntax is as follows:

```
MPI_Scatter(sendbuf, sendcount, send_MPI_TYPE, recvbuf, recvcount,
    recv_MPI_TYPE, root, COMMUNICATOR);
```

In this case, the sendbuf sends sendcount elements, while it will have sendcount*size elements.

In the remittent we need to state the sendbuf, while in the process that receives is not needed, it can be replaced with a 'nullptr'.

Gather does the inverse operation as Scatter. It joins portions of data in sendbuf from all the processes in a communicator to root, storing them all in recvbuf. The syntax is the following:

```
MPI_Gather(sendbuf, sendcount, send_MPI_TYPE, recvbuf, recvcount,
    recv_MPI_TYPE, root, COMMUNICATOR);
```

If we need the data available in all the processes, we can use MPI_Allgather as follows:

```
MPI_Allgather(sendbuf, sendcount, send_MPI_TYPE, recvbuf, recvcount,
    recv_MPI_TYPE, COMMUNICATOR);
```

References

Ardagna, D. (2024). *052496 - algorithms and parallel computing.* Retrieved from https://aunicalogin.polimi.it/aunicalogin/getservizio.xml?id_servizio=178&c_classe=837347 (Accessed: 2025-02-15)