UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence

Department of mathematics informatics and geosciences

Advanced Programming

Lecturer:
Prof. Pasquale Claudio Africa

Authors:

Christian Faccio
Andrea Spinelli

February 8, 2025

github.com/christianfaccio christianfaccio@outlook.it

Abstract

As a student of the "Data Science and Artificial Intelligence" master's degree at the University of Trieste, I have created these notes to study the course "Advanced Programming" held by Prof. Pasquale Claudio Africa. The course aims to provide students with a solid foundation in programming, focusing on the C++ programming language. The course covers the following topics:

- Bash scripting
- C++ basics
- Object-oriented programming
- Templates
- Standard Template Library (STL)
- Libraries
- Makefile
- CMake
- C++11/14/17/20 features
- Integration with Python
- Parallel programming (not covered in the lectures but useful for the HPC course)

While these notes were primarily created for my personal study, they may serve as a valuable resource for fellow students and professionals interested in this field.

Contents

1 Unix Shell

1.1 What is a Shell?

The shell is the primary interface between users and the computer's core system. When you type commands in a terminal, the shell interprets these instructions and communicates with the operating system to execute them. It serves as a crucial layer that makes complex system operations accessible through simple text commands.

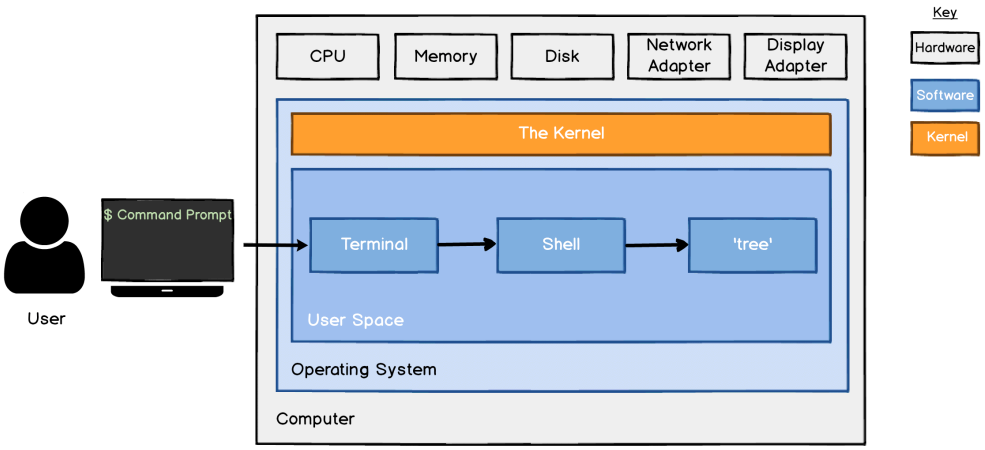

Figure 1.1: The Shell

Definition: *Shell*

A shell is a program that provides the traditional, text-only user interface for Linux and other UNIX-like operating systems. Its primary function is to read commands that are typed into a console […] and then execute (i.e., run) them. The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an interface between the user and the internal parts of the OS (at the very core of which is the kernel). *~Linfo*

1.1.1 Types of shell

Over time, various shells have evolved to meet different needs. The most widely used is `Bash` (Bourne Again Shell), named after its creator Stephen Bourne. It's the go-to shell for most Linux systems, serving both as a command interpreter and scripting language. Notably, macOS switched to `zsh` (Bash-compatible) from Catalina onward, while other alternatives include `Fish`, `ksh`, and `PowerShell`.

Tip: *Changing the Shell*

The shell might be changed by simply typing its name and even the default shell might be changed for all sessions.

Login vs. Non-login Shell

A **login shell** is invoked when a user logs into the system (e.g., through a virtual terminal by pressing `Ctrl+Alt+F1`). It requires the user to provide a username and password. Once authenticated, the user is presented with an interactive shell session. Login shells are typically the first point of interaction between a user and the system.
A **non-login shell**, on the other hand, does not require the user to log in again, as it is executed within an already active user session. For example, opening a graphical terminal in a desktop environment provides a non-login (interactive) shell. Non-login shells are commonly used in environments where the user is already authenticated.

Interactive vs. Non-interactive Shell

An **interactive shell** allows users to type commands and receive immediate feedback. Both login and non-login shells can be interactive. Examples include graphical terminals and virtual terminals. In interactive shells, the prompt (`$PS1`) must be set, which provides the user interface for command input. A **non-interactive shell**, however, is typically executed in automated environments, such as scripts or batch processes. Input and output are generally hidden unless explicitly managed by the calling process. Non-interactive shells are usually non-login shells since the user is already authenticated. For instance, when a script is executed, it runs in a non-interactive shell. However, scripts can emulate interactivity by prompting users for input.

1.2 Shell Scripting

1.2.1 Variables and Environmental Variables

Shells, like any program, use variables to store data. Variables are assigned values using the equals sign (`=`) without spaces. For example, to assign the value `1` to the variable `A`, one would type:

```
A=1
```

To retrieve a variable's value, the dollar sign (`$`) and curly braces are used. For example:

```
echo ${A}
```

Certain variables, called **environmental variables**, influence how processes run on the system. These variables are often predefined. For instance, to display the user's home directory, use: `echo ${HOME}`. To create an environmental variable, prepend the command `export`.

Example: *Setting the PATH*

For example, to add `/usr/sbin` to the `PATH` environmental variable:

```
export PATH="/usr/sbin:$PATH"
```

The `PATH` variable specifies directories where executable programs are located, ensuring commands can be executed without specifying full paths.
When a terminal is launched, the UNIX system invokes the shell interpreter specified in the `SHELL` environment variable. If `SHELL` is unset, the system default is used. After sourcing initialization files, the shell presents the prompt, which is defined by the `$PS1` environment variable.

1.2.2 Initialization Files

Initialization files are scripts or configuration files executed when a shell session starts. They set up the shell environment, define default settings, and customize behavior. Depending on the type of shell (login, non-login, interactive, or non-interactive), different initialization files are sourced.

- **Login Shell Initialization Files**:
  - Bourne-compatible shells: `/etc/profile`, `/etc/profile.d/*`, `~/.profile`.
  - Bash: `~/.bash_profile` (or `~/.bash_login`).
  - zsh: `/etc/zprofile`, `~/.zprofile`.
  - csh: `/etc/csh.login`, `~/.login`.
- **Non-login Shell Initialization Files**:
  - Bash: `/etc/bash.bashrc`, `~/.bashrc`.
- **Interactive Shell Initialization Files**:
  - `/etc/profile`, `/etc/profile.d/*`, and `~/.profile`.
  - For Bash: `/etc/bash.bashrc` and `~/.bashrc`.
- **Non-interactive Shell Initialization Files**:
  - For Bash: `/etc/bash.bashrc`.
    However, most scripts begin with the condition `[ -z "$PS1" ] && return`. This means that if the shell is non-interactive (as indicated by the absence of the `$PS1` prompt variable), the script stops execution immediately.
  - Depending on the shell, the file specified in the `$ENV` (or `$BASH_ENV`) environment variable may also be read.

1.2.3 Basic Shell Commands

To become familiar with the shell, let's start with some fundamental commands:

- `echo`: Prints the text or variable values you provide at the shell prompt.
- `date`: Displays the current date and time.
- `clear`: Clears the terminal screen.
- `pwd`: Stands for *Print Working Directory*, showing the current directory the shell is operating in. It is also the default location where commands look for files.
- `ls`: Stands for *List*, and lists the contents of the current directory.
- `cd`: Stands for *Change Directory*, and switches the current directory to the specified path.
- `cp`: Stands for *Copy*, and duplicates files or directories from a source to a destination.
- `mv`: Stands for *Move*, and transfers files or directories from one location to another. It can also rename files.
- `touch`: Creates a new, empty file or updates the timestamps of an existing file.
- `mkdir`: Stands for *Make Directory*, and creates new directories.
- `rm`: Stands for *Remove*, and deletes files or directories. To delete directories, the recursive option (`-r`) must be used.

⚠ Warning: *Remove Command*

Be cautious when using the `rm` command, as it permanently deletes files and directories **without moving them to the trash**.

1.2.4 Shell Scripts

Commands can be written in a script file, which is a text file containing instructions for the shell to execute. The first line of the script, known as the **shebang**, specifies the interpreter to use.

```
#!/bin/bash
#!/usr/bin/env python
```

To make a script executable, you need to change its permissions: `chmod +x script_file`

Not All Commands Are the Same

Commands in the shell can behave differently depending on how they are executed. For example, when a command like `ls` is run, it creates a **subprocess**, a separate instance that inherits the environment of the parent shell. This subprocess runs the command and then terminates, returning control to the parent shell.

Tip: ***Running Commands***

- **Subprocess**: Subprocesses can't modify the parent shell's environment or state. Changes made in subprocesses don't persist.
- `source`: The `source` command (or `.`) runs scripts in the current shell context, allowing environment modifications.
- **Scripts**: Running with `./script_file` creates a subprocess, isolating effects from the parent shell.

Types of Commands

In the shell, commands can fall into several categories:

- **Built-in Commands**: These are commands provided directly by the shell, such as `cd`, that are executed without creating a subprocess, necessary to update environment variables.
- **Executables**: These are standalone programs stored in directories specified by the `$PATH` environment variable. Examples include `ls`, `grep`, and `find`.
- **Functions and Aliases**: These are user-defined commands or shortcuts, often configured in initialization files like `~/.bashrc`.

To determine the type of a command and its location you can use:

- `type command_name` to identify if the command is built-in, an executable, or a function/alias.
- `which command_name` to find the exact location of an executable in the file system.

Warning: ***Spaces in File Names***

Avoid using spaces or accented characters in file names. Instead, use:

- `my_file_name` (snake case),
- `myFileName` (camel case),
- `my-file-name` (kebab case).

Spaces complicate scripts and make parsing error-prone.

1.2.5 Functions

A **function** in a shell script is a reusable block of code. The syntax is:

```
function_name() {
    # Commands to execute
}
```

Scripts or functions can access **input arguments** passed to them using special variables:

- `$0`: The name of the script or function.
- `$1`, `$2`, `$3`, etc.: The first, second, third, etc., arguments.
- `$#`: The number of arguments passed.
- `$@`: All arguments as separate words.
- `$*`: All arguments as a single word (rarely used).

Example: *Function that prints the sum of two numbers*

```
sum() {
    echo $(($1 + $2))
}
```

1.2.6 Additional Shell Commands

More Commands

- `cat`: *Concatenate*. Reads and outputs the contents of files. It can read multiple files and concatenate their content.
- `wc`: *Word Count*. Provides statistics like newline, word and byte count for a list of files.
- `grep`: *Global Regular Expression Print*. Searches for lines containing a specific string or matching a given pattern.
- `head`: Displays the first few lines of a file.
- `tail`: Displays the last few lines of a file.
- `file`: Examines specified files to determine their type.

Redirection, Pipelines, and Filters

Commands can be combined using operators to manipulate input and output streams:

- The **pipe operator** (`|`) forwards the output of one command to another.
  Example: `cat /etc/passwd | grep <word>` filters system information for a specific word.
- The **redirect operator** (`>`) sends the standard output to a file.
  Example: `ls > files-in-this-folder.txt`.
- The **append operator** (`>>`) appends output to an existing file.
- The **operator** (`&>`) redirects both standard output and standard error to a file.
- **Logical operators**:
  - `&&`: Executes the next command only if the previous one succeeds.
  - `||`: Executes the next command only if the previous one fails.
  - `;`: Executes commands sequentially, regardless of the status of the previous command.
- `$?`: Contains the exit status of the last command.

Advanced Commands

Cmd	Description	Syntax
`tr`	Translates characters in `SET1` to corresponding characters in `SET2`. If `SET2` is omitted, deletes characters in `SET1`. **Options:** • `-d`: Delete characters in `SET1`. • `-s`: Squeeze repeated characters.	`tr [options] SET1 [SET2]`
`sed`	Stream editor for filtering and transforming text. **Common uses:** • Substitution: `'s/pattern/replacement/flags'` • Deletion: `'Nd'` deletes the N-th line. **Flags:** • `g`: Global replacement. • `n`: Replace n-th occurrence.	`sed [options] 'command' file`
`cut`	Removes sections from each line. **Extraction methods:** • `-b` *list*: Select bytes. • `-c` *list*: Select characters. • `-f` *list*: Select fields; delimiter specified with `-d`.	`cut [options] file`
`find`	Searches for files in a directory hierarchy. **Expressions:** • `-name pattern`: Matches filenames. • `-type [f|d]`: Finds files (`f`) or directories (`d`). • `-exec command {} \;`: Executes command on each match.	`find [path ...] [expression]`
`locate`	Searches a database for filenames matching a pattern. Faster than `find`, but requires database updates. **Options:** • `-i`: Ignore case. • `-r`: Use regular expressions.	`locate [options] pattern`

Example: *Usage cases*

- `tr`:
  - `echo "hello" | tr 'a-z' 'A-Z'`: Converts lowercase letters to uppercase.
  - `echo "hello123" | tr -d '0-9'`: Removes digits from the input.
- `sed`:
  - `echo 'UNIX is great' | sed 's/UNIX/Linux/'`: Replaces 'UNIX' with 'Linux'.
  - `echo -e "1\n2\n3" | sed '2d'`: Deletes the second line.
- `cut`:
  - `cut -d ',' -f 1 file.csv`: Extracts the first field from a CSV file.
  - `cut -c 1-5 file.txt`: Extracts the first five characters of each line.

- `find` :
  - `find  -type f -name "*.txt"` : Finds all text files in the user's home directory.
  - `find . -type d -name "lib*"` : Finds all directories in the current directory whose names start with "lib".
- `locate` :
  - `locate -i foo` : Finds all files or directories whose name contains 'foo', ignoring case.
  - `locate -r '.*\.txt'` : Finds all files ending with '.txt' using a regular expression.

Quoting in Shell

Quoting affects how strings and variables are interpreted:

- `""` : Double quotes interpret variables.
- `''` : Single quotes treat everything as literal.

```
a=yes
echo "$a" # Outputs "yes".
echo '$a' # Outputs "$a".
```

The output of a command can be converted into a string and assigned to a variable for later reuse:

```
list=$(ls -l)$
# or equivalently:
list=`ls -l`
```

Processes

Managing background and foreground processes:

- `./my_command &` : Run a command in the background.
- `Ctrl-Z` : Suspend the current process.
- `jobs` : List background processes.
- `bg %n` : Resume a suspended process in the background.
- `fg %n` : Bring a background process to the foreground.
- `Ctrl-C` : Terminate the foreground process.
- `kill pid` : Send a termination signal to a process.
- `ps aux | grep process` : Find running processes.

Processes in the background are terminated when the terminal closes unless started with `nohup`.

Tip: ***How to Get Help***

- `command -h` or `command --help` : Display a brief help message.
- `man command` : Access the manual for the command.
- `info command` : Show detailed information (if available).

1.3 Git introduction

Git Cheatsheet

Version control is the practice of tracking and managing the changes in the software code. `Git` is an open-source version control system, which helps teams manage changes to the source code over time. It is **distributed**, that is, every developer has the full history of their code repository locally.

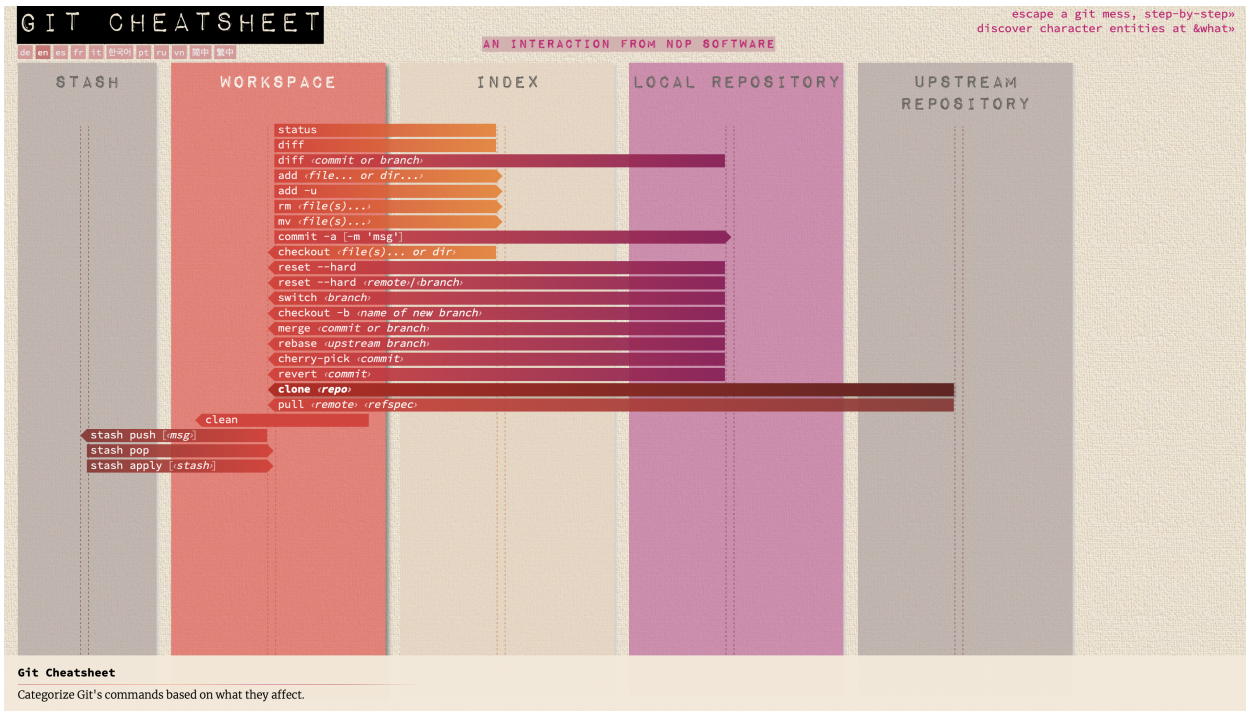

Figure 1.2: Git Cheatsheet

Basic Git commands:

- `git init`: Initialize a new Git repository.
- `git clone <url>`: Clone a repository from a URL.
- `git status`: Show the status of the working directory.
- `git add <file>`: Add a file to the staging area.
- `git commit -m "message"`: Commit changes to the repository.
- `git push origin <branch>`: Push changes to a remote repository.
- `git pull origin <branch>`: Pull changes from a remote repository.
- `git branch`: List all branches in the repository.
- `git checkout <branch>`: Switch to a different branch.
- `git merge <branch>`: Merge changes from a branch into the current branch.
- `git log`: Show the commit history.

To collaborate, just remember to pull the changes made by your colleagues before pushing your own changes. If you are working on the same files, the best practice is to create a new **branch**, such that you can merge the changes only when you are sure that everything works as expected.

When you create a branch, Git basically adds a new pointer to the latest commit. This pointer moves forward as you add new commits. When you switch branches, Git moves the pointer to the

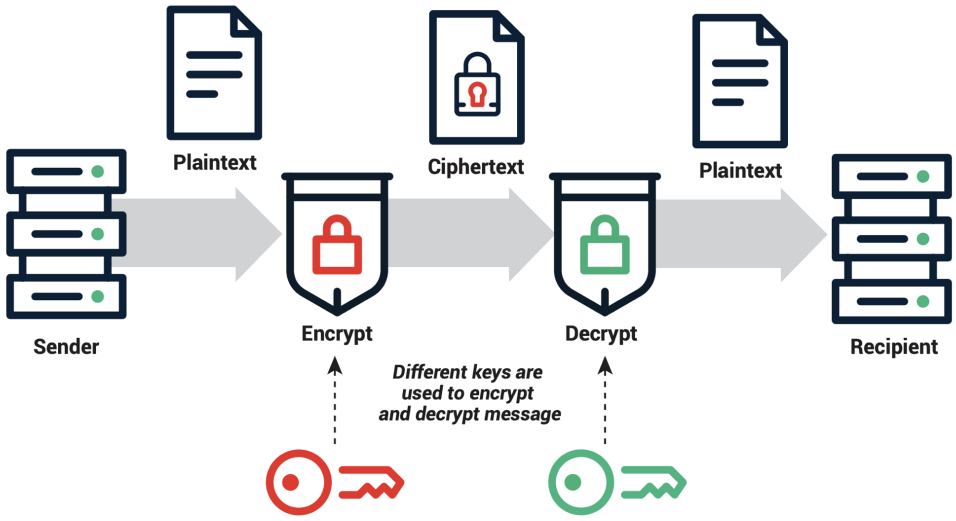

Figure 1.3: SSH

latest commit of the branch you are switching to. When you merge branches, Git combines the changes of the two branches and moves the pointer to the latest commit of the merged branches. If it encounters some conflics, it will ask you to solve them. Rebasing allow you to move the commits of a branch on top of another branch, so that the history is linear.

Best practices:

- Commit often, with meaningful messages.
- Use branches to isolate changes.
- Pull before pushing.
- Write tests for your code.
- Use a `.gitignore` file to exclude files from version control.

1.3.1 SSH Authentication

To avoid typing your username and password every time you push or pull from a repository, you can use SSH keys for authentication.

Tip: *SSH Keys*

1. Generate a new SSH key: `ssh-keygen -t rsa -b 4096 -C "youremail"`.
2. Add the SSH key to the SSH agent:
   `eval "$(ssh-agent -s)"; ssh-add ~/.ssh/id_rsa`.
3. Add the SSH key to your Git account.
4. Test the SSH connection: `ssh -T`
5. Change the remote URL to the SSH URL: `git remote set-url origin`
6. Test the connection: `git pull`.

2
C++ Basic

2.1 The Birth and Evolution of C++

Programming languages have always evolved to address the growing complexity of software development. Among these, C++ stands out as a language that bridges the gap between low-level system control and high-level abstraction. It combines the performance of C with the principles of object-oriented programming, enabling developers to tackle complex systems efficiently.

The story of C++ begins with C, a powerful yet simple language created by Dennis Ritchie at Bell Labs in the early 1970s. Its efficiency and portability made it a foundation for system programming. In 1979, Bjarne Stroustrup set out to enhance C by introducing support for object-oriented programming (OOP), creating what he called "C with Classes." This evolved into C++ in 1983, symbolizing an increment over C, and laid the groundwork for modern software engineering.

Standardization followed in the late 1980s, ensuring compatibility across platforms. Over the years, successive standards like C++11, C++17, and C++20 have introduced features such as smart pointers, lambda expressions, and modules, keeping C++ at the forefront of programming innovation.

2.1.1 Key Features of C++

- **Object-Oriented Programming (OOP)**: C++ introduced core OOP concepts like classes, inheritance, and polymorphism, enabling developers to design modular, reusable, and maintainable software.
- **Generic Programming**: The inclusion of templates brought the power of generic programming, allowing for flexible and reusable data structures and algorithms. Techniques like template metaprogramming extended this capability further.

2.1.2 Modern Applications and Impact

Today, C++ is used across diverse fields, including game development, embedded systems, scientific computing, and finance. Its combination of performance and expressiveness makes it a vital tool for building software that demands both efficiency and scalability.

An active open-source community, including projects like the Boost C++ Libraries, has greatly expanded its capabilities. With ongoing innovations like concepts and modules, C++ continues to adapt to the demands of modern software development, ensuring its relevance for decades to come.

2.2 The build process

2.2.1 Compiled vs. Interpreted Languages

C++ is a **compiled language**, which means that the source code must be translated into machine code before it can be executed. This translation process is performed by a **compiler**, which reads your source code and generates an executable file that can be run on a computer.

In contrast, **interpreted languages** like Python are executed line by line by an **interpreter**. The interpreter reads the code, evaluates it, and executes the corresponding instructions without generating a separate executable file.

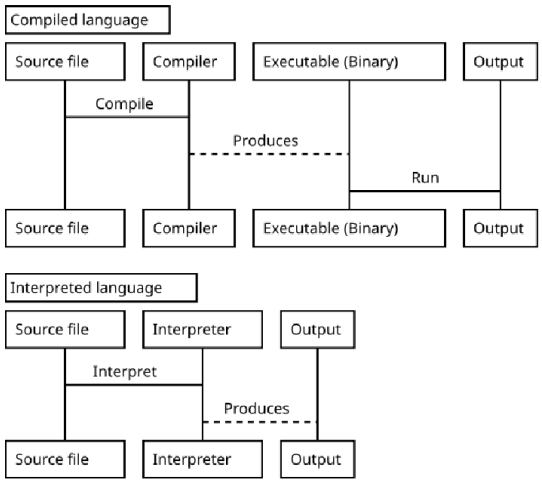

Figure 2.1: Compiled vs. Interpreted Languages

2.2.2 The Build Process

The build process for a C++ program involves several steps:

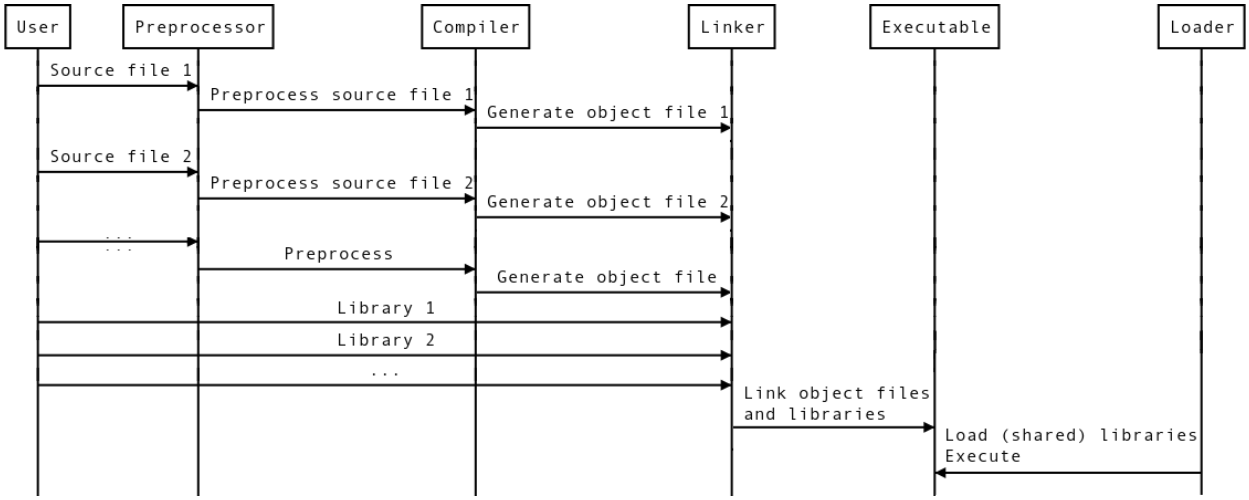

Figure 2.2: The Build Process

Preprocessor

The preprocessor is the first step in the build process. It processes directives that begin with `#` and modifies the source code before it is compiled. Common preprocessor directives include:

- `#include` for including header files;
- `#define` for defining macros;
- `#ifdef`, `#ifndef`, `#else`, `#endif` for conditional compilation;
- `#pragma` for compiler-specific directives.

Example: *Preprocessor*

Original source code:

```cpp
#include <iostream>
#define GREETING "Hello, World!"

int main() {
    std::cout << GREETING << std::endl;
    return 0;
}
```

Preprocessed source code:

```cpp
// Content of <iostream>, simplified for demonstration
namespace std {
    extern ostream cout;
    extern ostream endl;
}

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Compiler

The **compiler** translates the preprocessed source code into assembly or machine code. This phase involves multiple steps:

1. **Lexical Analysis**: Tokenizes the source code into meaningful elements like keywords, identifiers, and operators.
2. **Syntax Analysis (Parsing)**: Constructs a syntax tree or abstract syntax tree (AST) to represent the grammatical structure of the code.
3. **Semantic Analysis**: Checks for logical consistency, type compatibility, and adherence to language rules.
4. **Code Generation**: Converts the AST into assembly or machine code.
5. **Optimization**: Enhances the efficiency of the generated code.
6. **Output**: Produces object files containing machine code.

```
g++ main.cpp -o main.o
```

Common compiler options include:

- `-O`: Specify optimization levels (e.g., `-O2`, `-O3`).
- `-g`: Include debugging information for tools like `gdb`.
- `-std`: Specify the C++ standard (e.g., `-std=c++17`, `-std=c++20`).

Linker

The **linker** combines object files into a single executable. This step supports modular programming and ensures that all references between different parts of the program are resolved.
The linking process includes:

1. **Symbol Resolution**: Matches symbols (function, variable names, ...) between object files.
2. **Relocation**: Adjusts memory addresses to create a unified memory layout for the program.
3. **Output**: Produces an executable file.
4. **Linker Errors/Warnings**: Identifies missing symbols or conflicts.

```
{Linking Object Files}
    g++ main.o helper.o -o my_program
```

Linking can be static or dynamic:

- **Static Linking**: All required libraries are included in the final binary, resulting in a larger file size. Libraries do not need to be present on the target system.
- **Dynamic Linking**: Libraries are referenced at runtime, resulting in a smaller binary. Requires the necessary libraries to be present on the system during execution.

Loader

The **loader** prepares the executable for execution by loading it into memory, handling these steps:

1. **Memory Allocation**: Reserves memory for the executable and its data.
2. **Relocation**: Adjusts memory addresses as necessary to account for the executable's location in memory.
3. **Initialization**: Sets up the runtime environment for the program.
4. **Execution**: Begins executing the program's entry point (e.g., `main()` in C++).

Observation:

Dynamic linking at runtime enhances flexibility by including external libraries only when the program is executed. This approach reduces the initial binary size and allows for library updates without recompiling the application.

2.3 Structure of a Basic C++ Program

2.3.1 Overview of Program Structure

A typical C++ program is composed of a collection of functions. Every C++ program must include the `main()` function, which serves as the entry point. Additional functions can be defined as needed, and their statements are enclosed in curly braces `{}`. Statements are executed sequentially unless control structures like loops or conditionals are applied.

Example: *Basic Program Structure*

```cpp
#include <iostream>

int main() { // Entry point of the program.
    std::cout << "Hello, world!" << std::endl;
    return 0; // Indicates successful execution.
}
```

- `#include <iostream>` : Includes the Input/Output stream library.
- `int main()` : The entry point function.

- `std::cout` : Standard output stream for printing to the console.
- `<<` : Stream insertion operator.
- `"Hello, world!"` : The string to print.
- `<< std::endl` : Outputs a newline character and flushes the stream.
- `return 0;` : Indicates successful program termination.

How to Compile and Run

After writing your C++ program, use the GNU C++ compiler (g++) to create an executable:

```
g++ hello_world.cpp -o hello_world
```

Execute the compiled program from the terminal, optionally passing command-line arguments:

```
./hello_world [arg1] [arg2] ... [argN]
```

Verify the program's execution status by examining its exit code (0 typically indicates success):

```
echo $?
```

2.3.2 C++ as a Strongly Typed Language

C++ enforces strict type checking during compilation. Variables must be declared with a specific type, ensuring type safety and reducing runtime errors.

It provides various built-in "**Fundamental Types**" to handle data of different kinds and sizes. These types include integers, floating-point numbers, characters, Booleans, and more.

Data Type	Size (Bytes)	<cstdint>
`bool`	1	
`char`	1	
`signed char`	1	`int8_t`
`unsigned char`	1	`uint8_t`
`short`	2	`int16_t`
`unsigned short`	2	`uint16_t`
`int`	4	`int32_t`
`unsigned int`	4	`uint32_t`
`long int`	4 or 8	`int32_t` or `int64_t`
`long unsigned int`	4 or 8	`uint32_t` or `uint64_t`
`long long int`	8	`int64_t`
`long long unsigned int`	8	`uint64_t`
`float`	4	
`double`	8	

Table 2.1: Sizes of Fundamental Types in C++

Example: *Strong Typing in C++*

```cpp
int x = 5;
char ch = 'A';
float f = 3.14;

x = 1.6;        // Legal, but truncated to 1.
f = "a string"; // Illegal.

unsigned int y{3.0}; // Uniform initialization: illegal.
```

C++ supports several **integer types** with varying sizes and value ranges. Common types include `int`, `short`, `long`, and `long long`.

```cpp
int age = 30;
short population = 32000;
long long large_number = 123456789012345;
```

Floating-point types represent real numbers. These include `float`, `double`, and `long double`. They are ideal for representing decimal values.

```cpp
float pi = 3.14;
double gravity = 9.81;
```

Floating-Point Arithmetic

Floating-point numbers in C++ are represented using the format $\pm f \cdot 2^e$, where:

- f: the *significand* or *mantissa*, representing the precision of the number.
- e: the *exponent*, determining the scale of the number.
- 2: the base, as floating-point numbers are typically stored in binary form.

This representation enables efficient handling of very large or very small values but comes with certain limitations, such as rounding errors.

Example:

```cpp
double epsilon = 1.0;

while (1.0 + epsilon != 1.0) {
    epsilon /= 2.0; // Finding machine epsilon.
}
```

```cpp
double a = 0.1, b = 0.2, c = 0.3;

if (a + b == c) { // Unsafe comparison.
    // Due to precision limitations, this might not hold true.
}

if (std::abs((a + b) - c) < 1e-9) {
    // Use a tolerance for safe comparison.
}
```

Normalized Numbers: In normalized form, the most significant bit of the significand is always 1, which ensures efficient use of available precision and avoids redundant representations.
IEEE 754 Standard: The IEEE 754 Standard defines how floating-point numbers are represented and manipulated. It specifies:

- Standardized formats for `float`, `double`, and `long double`.
- Rounding rules to maintain accuracy.
- Special values such as `NaN` (Not-a-Number) and infinity for handling exceptional cases.

Characters and Strings

Characters in C++ are represented using the `char` type, while strings are sequences of characters represented by the `std::string` class.

```
char letter = 'A'; // Single character.
std::string name = "Alice"; // String of characters.
std::string greeting = "Hello, " + name + "!"; // Concatenation.
```

Boolean Types

C++ provides a built-in `bool` type for logical values. A `bool` variable can hold one of two values, `true` or `false`, useful for conditional statements and logical operations.

```
bool flag = true;
if (flag) {
    std::cout << "Flag is true" << std::endl;
}
```

Note that numbers can be implicitly converted to `bool`: 0 is `false`, while any other value is `true`.

```
if (0)    // false
if (42)   // true
```

2.3.3 `NULL`, `NaN`, and Key Differences

NULL

`NULL` is used to represent a null pointer or an invalid memory address. Its representation depends on the system and context:

- In C/C++, `NULL` is typically defined as `0` or `(void*)0`.
- In memory, a null pointer is often represented by a sequence of **all-zero bits**. (e.g. 32-bit system: `0x00000000`; 64-bit system: `0x0000000000000000`)

Example: *Representation of NULL*

```
int* ptr = NULL; // ptr points to memory address 0x00000000
if (ptr == NULL) {
    printf("Pointer is NULL\n");
}
```

NaN

`NaN` (Not a Number) is a special value used in floating-point arithmetic to represent undefined or unrepresentable results. Its representation is defined by the **IEEE 754 floating-point standard**:

- A `NaN` value is represented by an **exponent filled with 1s** (all bits set to 1) and a **non-zero significand**.
- There are two types of `NaN`, `qNaN` and `sNaN`:
  - **Quiet NaN**: Used for undefined or unrepresentable results. It has a leading `1` in the significand.
  - **Signaling NaN**: Used to trigger exceptions in certain operations. It has a leading `0` in the significand.

Example: *Representation of NaN*

```cpp
double x = 0.0 / 0.0; // Creates NaN
if (isnan(x)) {
    std::cout << "x is NaN\n";
}
```

Key Differences

Feature	`NULL`	`NaN`
Purpose	Represents a null pointer	Represents an undefined floating-point value
Data Type	Used with pointers	Used with floating-point numbers
Representation	All-zero bits	Exponent filled with 1s and non-zero significand
Context	Memory addresses, pointers	Floating-point arithmetic

2.3.4 Initialization and Aliases

Initialization assigns an initial value to a variable at the time of declaration. C++ supports several initialization methods: direct, copy, and uniform initialization.

```cpp
int x = 42;     // Direct initialization.
int y(30);      // Constructor-style initialization.
int z{15};      // Uniform initialization (preferred).
```

Type Aliases can create alternative names for existing types using `using` or `typedef`.

```cpp
using integer = int;    // Alias for int.
typedef float distance; // Alias for float.
```

2.3.5 The `auto` Keyword and Type conversion

The `auto` keyword allows the compiler to deduce the type of a variable based on its initialization value. This is useful for simplifying code and avoiding verbose type declarations.

```cpp
auto a{42};         // int.
auto b{12L};        // long.
auto c{5.0F};       // float.
auto d{10.0};       // double.
auto e{false};      // bool.
auto f{"string"};   // char[7].
```

Tip: *Best Practices*

- Use `auto` for complex types or when the exact type is unimportant.
- Avoid `auto` for publicly visible variables or ambiguous initializations.

C++ supports **implicit and explicit type conversions** to convert between different data types. Implicit conversions are performed automatically by the compiler, while explicit conversions require manual intervention.

Example: *Type Conversion*

Implicit conversion:

```
int x = 10;
double y = x; // int to double (implicit).
```

Explicit conversion:

```
double z = 3.14;
int w = static_cast<int>(z); // double to int (explicit).
```

2.4 Memory Management

In computer programming, memory is divided into two main regions: the **stack** and the **heap**. These regions serve different purposes and are managed differently. Below is a detailed explanation of their differences.

2.4.1 Stack Memory

The **stack** is used for **static memory allocation**, where memory is allocated and deallocated in a last-in, first-out (LIFO) order.

Key Characteristics of Stack Memory

- **Purpose**: Used for storing local variables, function parameters, and return addresses.
- **Management**: Memory allocation and deallocation are handled automatically by the compiler.
- **Speed**: Accessing the stack is very fast because it uses a simple pointer-based mechanism (the stack pointer).
- **Size**: The stack has a limited size, which is determined at the start of the program. If the stack exceeds its size, a **stack overflow** occurs.
- **Lifetime**: Memory is automatically freed when the function or block that allocated it exits.
- **Fragmentation-Free**: The stack does not suffer from fragmentation because memory is always allocated and freed in a strict order.

Example: *Stack Memory Example*

```
void foo() {
    int x = 10; // 'x' is allocated on the stack
    // Memory for 'x' is automatically freed when 'foo' exits
}
```

2.4.2 Heap Memory

The **heap** is used for **dynamic allocation**, where memory can be allocated and deallocated in any order.

Key Characteristics of Heap Memory

- **Purpose**: Used for dynamically allocated data (e.g., arrays, objects, or data structures whose size is not known at compile time).
- **Management**: Memory allocation and deallocation are managed manually by the programmer (e.g., using `malloc` / `free` in C or `new` / `delete` in C++).
- **Speed**: Accessing the heap is slower than the stack as it involves complex memory management.
- **Size**: The heap is much larger than the stack and can grow dynamically as needed (limited only by the system's available memory).
- **Lifetime**: Memory remains allocated until it is explicitly freed by the programmer. If not freed, it leads to **memory leaks**.
- **Fragmentation**: The heap can suffer from fragmentation over time, as memory is allocated and freed in arbitrary order.

Example: *Heap Memory Example*

```
1 void bar() {
2     int* ptr = new int(20); // 'ptr' points to memory allocated on
          the heap
3     // Memory for 'ptr' must be explicitly freed
4     delete ptr; // Free the memory to avoid a memory leak
5 }
```

2.4.3 Key Differences Between Stack and Heap

Feature	Stack Memory	Heap Memory
Purpose	Static memory allocation	Dynamic memory allocation
Management	Automatic (compiler-managed)	Manual (programmer-managed)
Speed	Very fast	Slower
Size	Limited (predefined size)	Large (limited by system memory)
Lifetime	Automatically freed when scope ends	Must be explicitly freed
Fragmentation	No fragmentation	Can suffer from fragmentation
Usage	Local variables, function parameters	Dynamically allocated data

When to Use Stack vs. Heap

- **Use the Stack**:
  - For small, short-lived data (e.g., local variables, function parameters).
  - When the size of the data is known at compile time.
  - When you want fast and automatic memory management.
- **Use the Heap**:
  - For large or dynamically sized data (e.g., arrays, objects).
  - When the lifetime of the data extends beyond the current scope.
  - When you need flexibility in memory allocation and deallocation.

2.4.4 Variables and Pointers

Variables represent named memory locations, while pointers store memory addresses, enabling direct access and manipulation. **Stack Variables** are declared locally within functions or blocks, are stored on the stack and are accessible directly. **Heap Variables** require pointers for access and explicit deallocation.

Example: *Working with Variables and Pointers*

```
int value = 10;        // Stack variable
int* ptr = &value;     // Pointer to stack variable

int* heap_var = new int(25); // Pointer to heap-allocated variable
*heap_var = 30;        // Modify heap variable through pointer

// Cleanup
delete heap_var;       // Deallocate heap memory
heap_var = nullptr;    // Reset pointer.
```

Lifetime and Scope

The lifetime of a variable refers to the duration it exists in memory, while its scope defines where it is accessible in code.

Stack Variables

- Limited to the scope of their defining function or block.
- Automatically deallocated when the scope ends.

Heap Variables

- Persist beyond their defining scope until explicitly deallocated.
- Risk memory leaks if not deallocated properly.

Example: *Lifetime and Scope Example*

```
void example() {                    //        Lifetime:
    int stack_var = 5;              // ends with the function.
    int* heap_var = new int(10);    // persists until delete.

    delete heap_var;                // Deallocate heap memory.
    heap_var = nullptr;             // Reset pointer.
}
```

Tip: *Best Practices for Memory Management*

- Use stack memory for small, short-lived variables.
- Use heap memory for large or long-lived data.
- Always match `new` with `delete` and `new[]` with `delete[]`.
- Prefer modern alternatives like `std::unique_ptr` or `std::shared_ptr` to manage heap memory safely.

2.5 Condition Statements

2.5.1 If-Else Statements

The `if-else` statement is used to execute code based on a condition. If the condition is true, the code inside the `if` block is executed; otherwise, the code inside the `else` block is executed.

```
int x = 15;
if (x != 10) {
    std::cout << "x is not equal to 10" << std::endl;
} else {
    std::cout << "x is equal to 10" << std::endl;
}
```

2.5.2 Switch Statements

The `switch` statement executes different code blocks based on the value of an expression. When a matching case is found, its code block is executed.

```
int choice = 2;
swit77ch (choice) {
    case 1:
        // Code for first option
        break;
    case 2:
        // Code for second option
        break;
    // ...
    default:
        // Code for when no cases match
        break;
}
```

2.5.3 For loop

The `for` loop is used to execute a block of code a specified number of times. It consists of three parts: initialization, condition, and increment/decrement.

```
for (int i = 0; i < 5; i++) {
    std::cout << i << std::endl;
}
```

2.5.4 While Loop

The `while` loop is used to execute a block of code as long as a specified condition is true.

```
int i = 0;
while (i < 5) {
    std::cout << i << std::endl;
    i++;
}
```

2.6 Functions and operators

2.6.1 Functions

Functions are blocks of code that perform a specific task. They are defined with a return type, name, parameters, and a body. Functions can be called from other parts of the program to execute the code inside them.

```
int add(int a, int b) {
    return a + b;
}
```

If a function does not output a value, its return type is `void`. A function can also output a general `auto` type, which allows the compiler to deduce the return type based on the return statement. Passing parameters by **value** creates a copy of the argument, while passing by **reference** allows the function to modify the original argument. The function can also return pointers or references, enabling the caller to access or modify the original data.

```
void increment(int& x) {
    x++;
}
auto add(int a, int b) {
    return a + b;
}
```

⚠ Warning: *const correctness*

`const` correctness is essential for writing safe and maintainable code. It prevents unintended modifications by marking parameters as read-only when they should not be altered.

- **Avoids unintended changes**: Prevents accidental modifications, improving safety.
- **Improves readability**: Clearly expresses the intent of function parameters.
- **Enables optimizations**: Helps the compiler optimize code by ensuring immutability.

```
void print(const std::string& message) {
    std::cout << message << std::endl;
}
```

Function Overloading

Function overloading is a feature in C++ that allows you to define multiple functions with the same name but different parameters. The compiler selects the appropriate function based on the number or types of arguments during the function call.

```
1 int add(int a, int b) {
2     return a + b;
3 }
4 int add(double a, double b) {
5     return a + b;
6 }
```

2.6.2 Operators

Increment and Decrement Operators

1. **Increment Operator (`++`)**: Increases the value of a variable by 1.
2. **Decrement Operator (`--`)**: Decreases the value of a variable by 1.

```
int x = 5;
x++; // x is now 6
x--; // x is now 5
```

2.7 User-defined types

- **enum**: Defines a set of named constant values.

```
enum Color { RED, GREEN, BLUE };
Color c = RED;
```

- **union**: Allows multiple data members to share the same memory location.

```
union Data {
    int x;
    float y;
};
Data d;
d.x = 10;
d.y = 3.14; // Overwrites 'x'
```

- **struct**: Groups related data members into a single unit.

```
struct Point {
    int x;
    int y;
};
Point p = {10, 20};
```

- **class**: Similar to a struct but with additional features like access control (public, private, protected).

```
class Circle {
public:
    double radius;
    double area() {
        return 3.14 * radius * radius;
    }
};
Circle c;
c.radius = 5.0;
double a = c.area();
```

2.8 Declarations and Definitions

- **Declaration**: Introduces the name and type of a variable, function, or class without allocating memory or defining its implementation.

```
// Declaration
extern int x;
void foo();
```

- **Definition**: Allocates memory and provides the implementation details for a variable, function, or class.

```
// Definition
int x = 10;
void foo() {
    std::cout << "Hello, world!" << std::endl;
}
```

2.9 Code Organization

Modular programming is a software design technique that divides code into separate modules, each responsible for a specific functionality. This approach enhances code readability, maintainability, and reusability. C++ modules can be organized into header files (`.h`), for declarations, and source files (`.cpp`), for definitions.

Example: *Header and Source Files*

```
1  // module.h (Header file)
2  #ifndef MODULE_H
3  #define MODULE_H
4
5  void foo();
6
7  #endif
8
9  // module.cpp (Source file)
10 #include "module.h"
11
12 void foo() {
13     std::cout << "Hello, world!" << std::endl;
14 }
```

2.9.1 Best practices

- **Header Guards**: Prevent multiple inclusions of the same header file.
- **Include Guards**: Use `#pragma once` or `#ifndef` guards to avoid redundant includes.
- **Separation of Concerns**: Keep declarations in header files and definitions in source files.
- **Forward Declarations**: Use forward declarations to reduce dependencies and improve compilation times.
- **Namespace Usage**: Enclose code in namespaces to prevent naming conflicts.

Example: *namespace usage*

```cpp
namespace math {
    int add(int a, int b) {
        return a + b;
    }
}

int main() {
    int sum = math::add(10, 20);
    return 0;
}
```

2.10 The Build toolchain in practice

2.10.1 Preprocessor and Compiler

The build process starts with the **preprocessor** (`cpp`) and the **compiler** (`g++` , `clang++`):

- **Preprocessor**: Handles directives like `#include` , performs macro substitution, and prepares code.
- **Compiler**: Translates preprocessed code into machine-readable object files.

These steps are often combined when using a compiler command like `g++` or `clang++` .

For a project with three files (`module.hpp` , `module.cpp` , `main.cpp`), the following commands illustrate preprocessing and compilation:

```
# Preprocessor step.
g++ -E module.cpp -I/path/to/include/dir -o module_preprocessed.cpp
g++ -E main.cpp -I/path/to/include/dir -o main_preprocessed.cpp

# Compilation step.
g++ -c module_preprocessed.cpp -o module.o
g++ -c main_preprocessed.cpp -o main.o
```

Observation:

- `-E` : Preprocess only.
- `-c` : Compile only (generate object file).
- `-I` : Include other directories (useful if the headers are not in the default directory).

2.10.2 Linker

The **linker** (`ld`) combines object files into an executable program by resolving external references between them. It also links external libraries if required.

```
g++ module.o main.o -o my_program
```

Linking Against Libraries

To link against external libraries, use the `-l` flag for library names (without the `lib` prefix or file extension) and the `-L` flag to specify the library directory:

```
g++ module.o main.o -o my_program -lmy_lib -L/path/to/my/lib
```

The `-lmy_lib` flag links to the `libmy_lib.so` (dynamic) or `libmy_lib.a` (static) file in the specified directory.

2.10.3 Preprocessor, Compiler, Linker: Simplified Workflow

For small projects with few dependencies, a single command can handle preprocessing, compilation, and linking:

```
g++ mod1.cpp mod2.cpp main.cpp -I/path/to/include/dir -o my_program
```

Warning: *Compiler Behavior*

Different compilers (e.g., GCC, Clang) may produce varying behaviors, warnings, or errors. For an example of such differences, see this comparison on GodBolt.

2.10.4 Loader

The **loader** is responsible for preparing the executable program for execution:

- Allocates memory for code and data sections.
- Resolves addresses for dynamically linked libraries.
- Starts program execution.

Running an Executable

```
./my_program
```

Dynamic Libraries and `LD_LIBRARY_PATH`

When linking against external dynamic libraries, the loader uses the environment variable `LD_LIBRARY_PATH` to locate them. Ensure the required library paths are included:

```
export LD_LIBRARY_PATH+=:/path/to/my/lib
./my_program
```

3 Object-Oriented Programming

3.1 Introduction

What is OOP

Object-Oriented Programming (OOP) is a programming paradigm that emphasizes the use of objects to represent real-world entities and concepts.

Definition: *OOP in C++*

C++ is not (only) an OOP language. It is a **multi-paradigm programming language** that supports procedural, object-oriented, and generic programming. C++ allows developers to combine these paradigms effectively for various programming tasks.

Key Principles of OOP

OOP is based on several key principles:

- **Encapsulation**: bundles data (attributes) and the functions (methods) that operate on the data into a single unit called an *object*. This promotes data hiding and reduces code complexity.
- **Inheritance**: allows you to create new classes (derived or child classes) based on existing classes (base or parent classes). It enables code reuse and the creation of class hierarchies.
- **Polymorphism**: enables objects from different derived classes to be handled through a common base class. This facilitates code reuse and provides a way to handle objects abstractly.

RAII (Resource Acquisition Is Initialization)

RAII is a C++ idiom that binds resource management to an object's lifetime. Resources are acquired in the constructor and released in the destructor, ensuring automatic cleanup when the object goes out of scope.

- Eliminates manual resource management and reduces leaks.
- Ensures deterministic resource release, even in case of exceptions.
- Manages memory, file handles, locks, and network connections efficiently.

Advantages of OOP

OOP offers numerous advantages, including:

- **Modularity**: OOP encourages the division of a complex system into smaller, manageable objects, promoting code modularity and reusability.
- **Maintenance**: Objects are self-contained, making it easier to maintain and update specific parts of the code without affecting other parts.
- **Flexibility**: Inheritance and polymorphism provide flexibility, allowing you to extend and modify the behavior of classes without altering their existing code.
- **Readability**: OOP promotes code readability by organizing data and functions related to a specific object within a class.

3.2 Classes and Objects in C++

3.2.1 Classes and Members

In C++, a class encapsulates `data` (member variables) and `behavior` (member functions), providing a blueprint for creating objects. This encapsulation ensures that data and related functionality are bundled together, promoting modularity and reusability.
The following example defines a simple `Car` class that models a real-world vehicle. It includes attributes such as the manufacturer, model, and year, along with a method to start the engine.

```
1  class Car {
2  public:
3      std::string manufacturer;
4      std::string model;
5      unsigned int year;
6
7      void start_engine() {
8          std::cout << "Engine started!" << std::endl;
9      }
10 };
```

Creating and Using Objects

Once a class is defined, objects can be instantiated to represent real entities. You can create an instance of the `Car` class and interact with its members using the dot `.` operator.

```
11 Car my_car;                      // Creating an object of class Car.
12 my_car.manufacturer = "BMW";
13 my_car.model = "X5";
14 my_car.year = 2024;
15 my_car.start_engine();           // Invoking a method.
```

C++ allows objects to be managed dynamically using pointers. This provides flexibility, especially when working with objects whose lifetimes extend beyond the current scope. The arrow operator `→` is used to access members of a pointer to an object.

```
16 Car* my_car_ptr = new Car{};
17 my_car_ptr->manufacturer = "Alfa Romeo";
18 my_car_ptr->model = "Giulietta";
19 my_car_ptr->year = 2010;
20 my_car_ptr->start_engine();      // Invoking a method.
21 delete my_car_ptr;               // Releasing allocated memory.
```

Member Variables and Functions

A class consists of two main components:

- **Member Variables**: Store the state of an object. They are also known as attributes or fields.
- **Member Functions**: Define the behavior of an object. They are also known as methods.

In the previous example, the member variables (`manufacturer` , `model` , `engine_running`) store the car's state, while member functions (`start_engine` , `is_running`) define its behavior.

Static Members

Static members are shared by all instances of a class. They are declared with the `static` keyword and accessed using the class name rather than an object. Consider the `Circle` class below, which uses a static constant for `PI`:

```
class Circle {
public:
    static const double PI = 3.14159265359;   // Static member.
    double radius;                            // Non-static member.

    double calculate_area() {
        return PI * radius * radius;
    }

    static void print_shape_name() {
        std::cout << "This is a circle." << std::endl;
    }
};
```

Static members can be accessed using the class name directly, while non-static members require an instance of the object:

```
Circle circle;
circle.radius = 5.0;
const double area = circle.calculate_area();  // non-static member.
const double pi_value = Circle::PI;           // static member.
Circle::print_shape_name();
```

Const Members

The keyword `const` can be applied to member variables and member functions to indicate immutability. In the example below, the constructor initializes a constant member, and the member function `print_value` is declared as `const` to signal that it does not modify the object.

```
class MyClass {
public:
    MyClass(int x) : value(x) {}

    void print_value() const {
        // value *= 2; // Illegal: cannot modify a const member.
        std::cout << "Const version: " << value << std::endl;
    }

    const int value;
};
```

Tip: `mutable` *Keyword*

If a const member function needs to modify a member variable, that variable can be declared as `mutable` (although this should be used sparingly).

```
mutable int var = 0; // Can be modified in a const member function
```

Example: *Const and Non-Const Member Functions*

Here is an example illustrating both const and non-const versions of a member function:

```
class Counter {
public:
    Counter() : count(0) {}

    // Non-const version can modify the object
    void increment() {
        count++;
    }

    // Const version can only read the object
    int get_count() const {
        return count;
    }

private:
    int count;
};

Counter c1;       // Non-const object
const Counter c2; // Const object

c1.get_count();   // OK      non-const obj can call const func
c1.increment();   // OK      non-const obj can call non-const func

c2.get_count();   // OK      const obj can call const func
c2.increment();   // Error   const obj cannot call non-const func
```

The `this` Pointer

The `this` pointer is automatically passed to non-static member functions and points to the object invoking the function. It is useful for resolving ambiguity between member variables and parameters.

```
class MyClass {
public:
    int x;

    void print_x() const {
        std::cout << "Value of x: " << this->x << std::endl;
    }
};
```

3.2.2 Constructors and Destructors

Constructors are special member functions that initialize objects when they are created. They share the same name as the class and may accept arguments to initialize member variables.

Default Constructor

A default constructor takes no arguments. If a class has no explicitly defined constructor, C++ generates a default constructor automatically.

```cpp
class MyClass {
public:
    // Default constructor.
    MyClass() = default;

    std::string name;
    unsigned int length;
};

MyClass obj;    // Default initialization.
MyClass obj2{}; // Uniform initialization (preferred).
MyClass* ptr = new MyClass(); // Calls the default constructor.
```

Observation: ***Default Constructor and Initialization***

If you provide any custom constructors, C++ *will not generate a default constructor* unless explicitly defined.
Default initialization of primitive types (e.g., `int`, `double`) sets them to zero, while objects may be initialized by their own default constructors.

Parameterized Constructor

A parameterized constructor allows objects to be initialized with specific values.

```cpp
class Student {
public:
    Student(std::string name, unsigned int age) {
        this->name = name;
        this->age = age;
    }

    std::string name;
    unsigned int age;
};

Student student1("Alice", 20); // initialization using a constructor.
Student student2{"Bob", 23}; // Uniform initialization.
```

A good practice is to use an **initializer list** to initialize member variables in the constructor:

```cpp
class Rect {
public:
    Rect(double length, double width) : length(length), width(width) {
        // Constructor body (if needed).
    }

    double calculate_area() const {
        return length * width;
    }

    double length;
    double width;
};

Rect rect{5.0, 3.0}; // initialization using an initializer list.
const double area = rect.calculate_area();
```

When using an initializer list, the member variables **must be initialized in the order they are declared** in the class definition.

Tip: ***Initializer List***

Using an initializer list avoids redundant assignments and improves efficiency.

Copy Constructor and Copy Assignment

A copy constructor creates a new object as a copy of an existing one, while the copy assignment operator assigns the contents of one object to another existing object.

```
class Book {
public:
    Book(std::string title, std::string author)
        : title(title), author(author) {}

    // Copy constructor.
    Book(const Book& other)
        : title(other.title), author(other.author) {}

    // Copy assignment operator.
    Book& operator=(const Book& other) {
        if (this != &other) {
            title = other.title;
            author = other.author;
        }
        return *this;
    }

    void display_info() const {
        std::cout << "Title:" << title << ", Author:" << author << "\n";
    }

    std::string title;
    std::string author;
};

Book book1{"The Catcher in the Rye", "J.D. Salinger"};
Book book2 = book1; // Copy constructor.
Book book3{"Marcovaldo", "I. Calvino"};
book3 = book1; // Copy assignment operator.
```

Observation: ***Pass and Return by Value***

When passing or returning objects by value, the copy constructor is invoked to create a copy of the object.

```
1 void some_function(Student s) {
2     // Calls the copy constructor when s is passed.
3 }
4
5 Student create_student() {
6     Student s{"Bob", 22};
7     return s; // Calls the copy constructor when s is returned.
8 }
```

Destructor

A destructor is a special member function that cleans up resources when an object is destroyed. It has the same name as the class but is prefixed with `~`. Destructors are automatically called when:

- An object goes out of scope
- The `delete` operator is used on a pointer to the object
- The program ends and static/global objects are destroyed

```
class FileHandler {
public:
    FileHandler(std::string filename) : filename(filename) {
        file.open(filename);
    }

    ~FileHandler() {
        if (file.is_open()) {
            file.close();
        }
    }

    std::string filename;
    std::ofstream file;
};

{ // The object is automatically destroyed when it goes out of scope.
    FileHandler file{"data.txt"};
} // Destructor is called here, ensuring proper resource cleanup.
```

Observation: *Rule of Three*

Copy constructor, copy assignment operator and destructor are part of the **Rule of Three** in C++, which states that if a class defines any of the following:

- Destructor
- Copy constructor
- Copy assignment operator

then it should provide all three to ensure proper memory management.

Compiler-Generated Special Member Functions

Table 3.1: The compiler automatically generates special member functions when not explicitly defined. [**howardhinnant**]

user declares	default constructor	destructor	copy constructor	copy assignment	move constructor	move assignment
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
default constructor	user declared	defaulted	defaulted	defaulted	defaulted	defaulted
destructor	defaulted	user declared	defaulted	defaulted	not declared	not declared
copy constructor	not declared	defaulted	user declared	defaulted	not declared	not declared
copy assignment	defaulted	defaulted	defaulted	user declared	not declared	not declared
move constructor	not declared	defaulted	deleted	deleted	user declared	not declared
move assignment	defaulted	defaulted	deleted	deleted	not declared	user declared

3.2.3 The `inline` Directive

The `inline` keyword in C++ suggests that the compiler replaces a function call with the actual function code at the call site. This can reduce function call overhead and improve performance for small, frequently used functions.

Syntax

```
// Inline function definition.
inline int add(int a, int b) {
    return a + b;
}

// Function call.
const int result = add(5, 7);
std::cout << "Result: " << result << std::endl;
```

Inline functions have the following characteristics:

- **Limited to small functions**: Inlining is beneficial for short functions but can lead to code bloat for larger functions.
- **Compiler optimization**: The `inline` keyword is only a suggestion; the compiler may choose whether to inline a function based on optimization settings.
- **Usage in header files**: Inline functions should be defined in header files to avoid multiple definition errors, but proper use of header guards is essential.
- **Impact on readability**: Excessive inlining can make debugging harder and lead to poor code organization.

Advantages

- Reduces function call overhead.
- Eliminates stack frame creation for simple functions.
- Helps avoid multiple definitions across translation units.
- Useful for small, performance-critical functions.

Considerations

- Excessive use can increase executable size.
- Compiler may ignore the `inline` directive if deemed inefficient.
- Debugging inlined functions is more difficult.
- Should be used cautiously to maintain code clarity.

Tip: *When to Use* `inline`

The `inline` directive is most effective when:

- Use for very small functions, such as one-liners.
- Use when function call overhead is significant.
- Avoid inlining large or recursive functions.
- Ensure proper placement in header files to prevent linkage issues.

3.2.4 In-Class and Out-of-Class Definitions

In C++, member functions of a class can be defined either inside the class declaration (in-class, implicitly `inline`) or separately in a source file (out-of-class). Each approach has its advantages and trade-offs.

In-Class Definition (Implicitly Inline)

A member function defined inside the class is automatically considered `inline`. This is common for short functions that are one-liners or concise operations.

```
// my_class.hpp
class MyClass {
public:
    // Inline function defined inside the class.
    int add(int a, int b) {
        return a + b;
    }
};
```

The compiler treats this as if it were explicitly marked `inline` when defined outside the class.

```
// my_class.hpp
class MyClass {
public:
    int add(int a, int b);
};

// Inline keyword is implicit when defined inside the class.
inline int MyClass::add(int a, int b) {
    return a + b;
}
```

Advantages

- Improves readability for simple functions.
- Allows potential inlining by the compiler, reducing function call overhead.

Disadvantages

- May lead to code bloat with large functions.
- Changes to the function force recompilation of all translation units, including header files.

Out-of-Class Definition (Separation of Interface and Implementation)

Member functions can also be declared in the class but defined separately in a source file (`.cpp`). This is typically done for larger functions or when separating interface from implementation.

```
// my_class.hpp
class MyClass {
public:
    int add(int a, int b); // Function declaration.
};
```

```
// my_class.cpp
#include "my_class.hpp"

int MyClass::add(int a, int b) {
    return a + b;
}
```

Advantages

- Improves code organization.
- Changes to function implementation do not require recompilation of all translation units.

Disadvantages

- Slightly more verbose.
- Requires managing separate `.cpp` files.

Tip: ***Best Practices for Function Definitions***

1. Use **in-class definitions** for very short functions (e.g., accessors, mutators) where inlining might improve performance.
2. Use **out-of-class definitions** for more complex functions to keep headers clean and separate interface from implementation.
3. Balance readability, maintainability, and performance when deciding where to define functions.

In practice, a combination of both approaches is used to maintain structured and efficient code.

3.2.5 Encapsulation and Access Control

Encapsulation is a key principle in object-oriented programming (OOP) that bundles data (attributes) and methods (functions) into a single unit: an object.
It restricts direct access to an object's internal state, ensuring data integrity and security.

Encapsulation in C++

Encapsulation allows defining a clear interface while hiding implementation details. Data members should generally be private, with public methods providing controlled access.

Example: ***Encapsulation Example***

```
class BankAccount {
public:
    BankAccount(std::string account_holder, double balance)
        : account_holder(account_holder), balance(balance) {}

    void deposit(double amount) {
        balance += amount;
    }

    double get_balance() const {
        return balance;
    }

private:
    std::string account_holder;
    double balance;
};
```

Access Specifiers in C++

C++ provides three access specifiers to control member visibility:

- `public`: Accessible from any part of the program (forms the class's public interface).
- `private`: Only accessible within the class itself (internal implementation).
- `protected`: Similar to `private`, but also accessible in derived classes (used in inheritance).

```
class MyClass {
public:
    int public_var;      // Accessible from anywhere.
    void public_func() { /* ... */ }

private:
    int private_var;     // Accessible only within this class.
    void private_func() { /* ... */ }
};
```

Class vs. Struct

In C++, both `class` and `struct` can be used to define objects with member variables and functions.

The primary difference between them lies in their **default access specifiers**:

- In a `class`, members are **private** by default.
- In a `struct`, members are **public** by default.

```
// Class vs. Struct Example
class MyClass {
    int x; // Private by default.
public:
    int y; // Explicitly public.
};

struct MyStruct {
    int x; // Public by default.
private:
    int y; // Explicitly private.
};
```

When to Use Class vs. Struct

While functionally equivalent, structs and classes serve different purposes in common C++ conventions.

Use `class` when:

- Encapsulating data with private members
- Defining objects with complex behavior
- Implementing OOP concepts (e.g. inheritance)
- Maintaining strong encapsulation

Use `struct` when:

- Grouping related public variables
- Using simple data structures (e.g. point, color)
- Interacting with C-style structs or APIs
- Defining POD (Plain Old Data) types

Getter and Setter Methods

Getter and setter methods (also known as **accessors and mutators**) provide controlled access to private member variables.

- **Getters**: Allow reading private data.
- **Setters**: Allow modifying private data with validation or restrictions.

Example: *Getter and Setter*

```
class TemperatureSensor {
public:
    double get_temperature() const {
        return temperature;
    }

    void set_temperature(double new_temperature) {
        if (new_temperature >= -50.0 && new_temperature <= 150.0) {
            temperature = new_temperature;
        } else {
            std::cout << "Invalid temperature value!" << std::endl;
        }
    }

private:
    double temperature;
};
```

Using getters and setters enforces encapsulation while allowing controlled data access.

3.2.6 Friend Classes

A `friend` class allows another class to access its private and protected members. This is useful when two classes need close interaction but maintaining strict encapsulation is impractical.

Let's consider a `Circle` class with a private member `radius`.

```
// circle.hpp
class Circle {
public:
    friend class Cylinder;
    // Cylinder class has access to private members of Circle.

    Circle(double r) : radius(r) {}

    double get_area() const {
        return 3.14159265359 * radius * radius;
    }

private:
    double radius;
};
```

Let's consider now a `Cylinder` class that requires access to the private member `radius` of the `Circle` class:

```cpp
// cylinder.hpp
class Cylinder {
public:
    Cylinder(const Circle &circle, double height)
        : circle(circle), height(height) {}

    double get_volume() const {
        // Accessing the private member 'radius' of Circle.
        return circle.radius * circle.radius * height;
    }

private:
    double height;
    const Circle circle;
};

Circle circle{1.0};
Cylinder cylinder{circle, 0.5};
const double volume = cylinder.get_volume();
```

Tip: ***Best Practices for Friend Classes***

- **Minimize usage**: Prefer encapsulation unless two classes require deep interaction.
- **Use only when necessary**: Friend classes break encapsulation, so they should be well justified.
- **Maintain modularity**: Avoid excessive interdependencies between classes.

Friend classes allow controlled access to private members but should be used sparingly to maintain encapsulation principles.

3.2.7 Operator Overloading

Operator overloading in C++ allows defining custom behaviors for operators when used with user-defined classes. This makes objects behave more naturally and improves code readability.

Category	Operators
Arithmetic	`+` , `-` , `*` , `/` , `%`
Comparison	`==` , $\neq$, `<` , `>` , $\leq$, $\geq$, $\iff$ (C++20)
Assignment	`=` , `+=` , `-=` , `*=`
Increment/Decrement	`++` , `--`
Stream	`<<` (output), `>>` (input)
Subscript	`[]` (array-like behavior)
Function Call	`()` (used in functor classes)
Pointer	$\rightarrow$, `*` (smart pointers, iterators)

Table 3.2: Commonly Overloaded Operators in C++

By overloading operators you can extend their functionality for custom types. Instead of writing verbose function calls, operator overloading lets you use familiar syntax for objects.

Example: *Operator Overloading*

```
class Complex {
public:
    Complex(double r, double i) : real(r), imag(i) {}

    // Overloading the + operator.
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    void print() const {
        std::cout << real << " + " << imag << "i" << std::endl;
    }

private:
    double real, imag;
};

Complex a{2.0, 3.0};
Complex b{1.0, 2.0};
Complex c = a + b; // Using the overloaded '+' operator.
c.print();
```

Operators can be overloaded as **member functions** or **non-member functions**.

Overloading as a Member Function

If the left operand is an object of the class, the operator can be overloaded as a member function.

```
// Overloading as a Member Function
class Vector {
public:
    Vector(int x, int y) : x(x), y(y) {}

    Vector operator+(const Vector& other) {
        return Vector(x + other.x, y + other.y);
    }

    void print() const {
        std::cout << "(" << x << ", " << y << ")" << std::endl;
    }

private:
    int x, y;
};

Vector v1{2, 3}, v2{1, 1};
Vector result = v1 + v2; // Works because + is a member function.
result.print();
```

Overloading as a Non-Member Function

If the left operand is not an object of the class, the operator must be overloaded as a non-member function.

```cpp
// Overloading as a Non-Member Function (Friend Function)
class MyClass {
public:
    MyClass(int v) : value(v) {}

    // Declaring the '<<' operator as a friend function.
    friend std::ostream& operator<<(std::ostream& os, const MyClass& obj
        );

private:
    int value;
};

// Overloading the '<<' operator as a non-member function.
std::ostream& operator<<(std::ostream& os, const MyClass& obj) {
    os << obj.value;
    return os;
}

MyClass obj{42};
std::cout << obj << std::endl; // Calls the overloaded operator.
```

Tip: *Operator Overloading Guidelines*

1. **Operators that cannot be overloaded**
   Some operators like `::`, `.*`, and `? :` cannot be overloaded.
2. **Preserve the operator's meaning**
   Overloading should make sense in the class's context (e.g., `+` for addition).
3. **Respect operator precedence**
   Overloaded operators should follow the same precedence as their built-in counterparts.
4. **Avoid excessive overloading**
   Overloading too many operators can make code hard to maintain.

Operator overloading enhances code readability when used correctly, making custom types feel like native types.

3.3 Class Collaborations

In object-oriented programming, classes do not exist in isolation; they often collaborate in various ways to form complex software systems.

These collaborations can be classified into different types of relationships.

3.3.1 Association

A **"*loose*" relationship** where one class holds a reference or pointer to another but does not control its lifetime. This is useful for relationships where objects interact without being strongly coupled.

Here's an example of Association: A Student can be associated with multiple Courses.
Let's firstly define the `Course` class:

```
// course.hpp
class Course {
public:
    Course(const std::string& name) : course_name(name) {}

    const std::string& get_course_name() const {
        return course_name;
    }

private:
    std::string course_name;
};
```

Next, let's define the `Student` class which can maintain associations with multiple courses through enrollment:

```
// student.hpp
class Student {
public:
    Student(const std::string& name) : student_name(name) {}

    void enroll_course(Course *course) {
        enrolled_courses.push_back(course);
    }

    void list_enrolled_courses() const {
        std::cout << student_name << " is enrolled in:" << std::endl;
        for (const auto& course : enrolled_courses) {
            std::cout << "- " << course->get_course_name() << std::endl;
        }
    }

private:
    std::string student_name;
    std::vector<Course*> enrolled_courses; // Association with Course.
};
```

The `Student` class has a vector of `Course` pointers, representing an association between the two classes.

This implements a many-to-many relationship pattern, where a student can be enrolled in multiple courses, and courses can have multiple students.

```
// main.cpp
Course math{"Math"};
Course physics{"Physics"};
Course biology{"Biology"};

Student alice{"Alice"};
Student bob{"Bob"};

alice.enroll_course(&math);
alice.enroll_course(&physics);
bob.enroll_course(&biology);
bob.enroll_course(&physics);

alice.list_enrolled_courses(); // Alice is enrolled in: - Math - Physics
bob.list_enrolled_courses(); // Bob is enrolled in: - Biology - Physics
```

Associations are useful for modeling relationships between classes without strong dependencies: one class can exist without the other.

3.3.2 Aggregation

A **"*has-a*" relationship** where one class contains another, but the contained object can exist independently.

```
class Wheel {
public:
    void rotate() { /* ... */ }
};

class Car {
private:
    std::vector<Wheel*> wheels; // Car uses existing Wheels.

public:
    Car(std::vector<Wheel*>& wheels) : wheels(wheels) {}

    void drive() {
        for (auto* wheel : wheels) {
            wheel->rotate();
        }
    }
};
```

In this example `Wheel` objects wheels can exist independently of the `Car`:

```
    Wheel front_left, front_right, rear_left, rear_right;
    std::vector<Wheel*> wheel_set = {&front_left, &front_right, &
       rear_left, &rear_right};

    Car my_car(wheel_set); // Car uses externally created Wheels.
    my_car.drive(); // Wheels are rotated.
```

Aggregation is useful for modeling objects that are composed of other objects but can exist independently.

3.3.3 Pointer vs. Reference for Aggregation

Choosing between pointers and references when aggregating polymorphic objects depends on the intended behavior.

Reference-Based Aggregation

- Use a **(const) reference** if the object remains unchanged over time.
- Guarantees that an object always exists.
- Not default-constructible.

Pointer-Based Aggregation

- Use a **pointer** if the object may change over time.
- Initialize raw pointers to `nullptr`.

Observation: *Memory Management*

Raw pointers can cause memory issues; consider smart pointers as `std::unique_ptr` or `std::shared_ptr` for automatic resource management.

3.3.4 Composition

Composition is a stronger **"*part-of*" relationship** where the contained object is an integral part of the containing object and cannot exist independently. The lifetime of the contained object is tied directly to the containing object's lifetime.

```
class Room {
public:
    void clean() { /* ... */ }
};

class Apartment {
private:
    Room living;  // Rooms cannot exist without an Apartment.
    Room kitchen;
    Room bedroom;

public:
    void clean() {
        living.clean();
        kitchen.clean();
        bedroom.clean();
    }
};
```

When the containing object is destroyed, all composed objects are automatically destroyed as well.

```
1 Apartment my_apartment;
2 my_apartment.clean(); // Cleans all rooms.
```

Observation: *Composition vs. Aggregation*

Composition and aggregation are both forms of object containment, but they differ in the strength of the relationship:

- **Composition**: Stronger relationship where the contained object is part of the containing object.
- **Aggregation**: Weaker relationship where the contained object can exist independently.

3.3.5 Views (Proxies)

A proxy (or view) is a specialized form of aggregation that provides an alternative interface for accessing an object's elements. A proxy does not store the actual data but redirects access to another underlying object while applying specific transformations or constraints.

```
class Matrix {
public:
    double & operator()(int i, int j);
};

class DiagonalView {
public:
    DiagonalView(Matrix &mat) : mat(mat) {}

    double & operator()(int i, int j) {
        return (i == j) ? mat(i, i) : 0.0;
    }

private:
    Matrix &mat;
};
```

`DiagonalView` provides a restricted view of a Matrix, allowing access only to diagonal elements. It prevents modifications to off-diagonal elements by returning `0.0` for those cases.

3.4 Inheritance and Polymorphism

3.4.1 Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows a new class (the derived or subclass) to inherit properties and behaviors from an existing class (the base or superclass). It establishes an **"*is-a*" relationship**, enabling code reuse and extension.
For example, a `Shape` class can serve as a base for `Circle`, `Rectangle`, and `Triangle` classes, all of which share common characteristics while extending functionality.

Inheritance in C++

In C++, inheritance is specified using using the `class` or `struct` keyword followed by a colon `:` and an access specifier (`public`, `protected`, or `private`) and the name of the base class.

Example: *Inheritance*

An example of inheritance is shown below, where the `Circle` class inherits from the `Shape` class:

```
class Shape { // Base class.
public:
    void f() { std::cout << "f (base class)." << std::endl; }
    void draw() { std::cout << "Drawing a shape." << std::endl; }
};

class Circle : public Shape { // Derived class.
public:
    void g() { std::cout << "g (derived class)." << std::endl; }
    void draw() { std::cout << "Drawing a circle." << std::endl; }
};

Circle circle;
circle.f();    // Calls f() from Shape
circle.g();    // Calls g() from Circle
circle.draw(); // Calls draw() from Circle (overrides Shape method)
```

Inheritance and Access Control

The choice of access specifier determines how base class members are inherited:

- **Public Inheritance**: Preserves the original access specifiers. Public members remain public, protected members remain protected.
- **Protected Inheritance**: Public members become protected; protected members remain protected.
- **Private Inheritance**: Public and protected members become private.

```
class Base {
public:
    int public_data;
protected:
    int protected_data;
private:
    int private_data;
};
```

The accessibility of each member in the derived class varies based on the inheritance type. The **??** below illustrates how different inheritance types affect member accessibility:

Member	Original Access	Inherited Access
Public Inheritance		
public_data	public	public
protected_data	protected	protected
private_data	private	inaccessible
Protected Inheritance		
public_data	public	protected
protected_data	protected	protected
private_data	private	inaccessible
Private Inheritance		
public_data	public	private
protected_data	protected	private
private_data	private	inaccessible

Table 3.3: Access Control in Different Types of Inheritance

```
class DerivedPublic : public Base {
    // public_data remains public.
    // protected_data remains protected.
    // private_data is inaccessible.
};

class DerivedProtected : protected Base {
    // public_data becomes protected.
    // protected_data remains protected.
    // private_data is inaccessible.
};

class DerivedPrivate : private Base {
    // public_data becomes private.
    // protected_data becomes private.
    // private_data is inaccessible.
};
```

Construction of a Derived Class

When constructing a derived object:

1. The base class constructor is called first.
2. The derived class constructor executes, initializing its own members.

```
class BaseClass {
public:
    BaseClass(int base_param) { /* Initialize base members */ }
};

class DerivedClass : public BaseClass {
public:
    DerivedClass(int derived_param, int base_param)
        : BaseClass(base_param) { /* Initialize derived members */ }
};
```

Delegating Constructor

In the constructor of a derived class, you can call the constructor of the base class, which is useful if you need to pass arguments. If no arguments are passed, the default constructor of the base class is used (in this case, the base class must be default constructible).

```
class B {
public:
    B(double x) : x(x) { /* ... */ }
private:
    double x;
};

class D : public B {
public:
    D(int i, double x) : B(x), my_i(i) { }
private:
    int my_i;
};

D d(4, 12.0); // Initializes d.x = 12.0, d.my_i = 4
```

Inheriting Constructors

Constructors are not inherited by default, but they can be explicitly made available in a derived class using `using`.

```
class B {
public:
    B(double x) : x(x) { /* ... */ }
};

class D : public B {
    using B::B; // Inherits B constructor.
private:
    int my_i = 10;
};

D d(12.0);
```

In the example above, when you create an instance like D d(12.0), the using declaration makes `B`'s constructor available in `D`. The constructor `B::B(double)` is called with `12.0`, setting `d.x` accordingly, while `d.my_i` is automatically initialized to its default value of `10`.

Destruction of a Derived Class

When an object of a derived class is destroyed:

1. The derived class destructor is executed first.
2. The base class destructor is called automatically.

```
DerivedClass::~DerivedClass() {
    // Clean up derived class resources.
    // ~BaseClass() is automatically called after this.
}
```

Multiple Inheritance

C++ allows multiple inheritance, where a class inherits from more than one base class. If multiple base classes have methods with the same name, ambiguity arises, which can be resolved using fully qualified names.

```
class B { public: double f() { return 1.0; } };
class C { public: double f() { return 2.0; } };

class D : public B, public C {
public:
    void fun() {
        double x = B::f(); // Resolving ambiguity by specifying base
            class.
    }
};
```

Multiple inheritance can lead to the ***diamond problem***, where a class indirectly inherits from the same base class through multiple paths. C++ provides `virtual` inheritance to ensure that only one instance of a shared base class is created.

```
class A { /* Base class */ };
class B : virtual public A { /* Inherits A virtually */ };
class C : virtual public A { /* Inherits A virtually */ };
class D : public B, public C { /* D contains only 1 instance of A */ };
```

3.4.2 Dynamic (Runtime) Polymorphism

Polymorphism

Public inheritance enables **polymorphism**, which allows objects from different classes within a hierarchy to be treated uniformly while still exhibiting type-specific behavior.

- A pointer or reference to a derived class `D` is implicitly convertible to a pointer or reference to a base class `B` (upcasting).
- A method declared as `virtual` in `B` can be overridden in `D` by providing a method with the same signature.
- If a base class pointer `B *b = new D` holds an instance of a derived class, calling a `virtual` method will execute the derived class's version.

⚠ Warning: *Virtual Methods return type*

Overridden virtual methods must have the same return type, with the exception that a method returning a pointer (or reference) to a base class may be overridden by a method returning a pointer (or reference) to a derived class.

"Is-a" Relationship

Polymorphism should be used only when there is a strong **"is-a"** relationship between the base and derived class. The public interface of the base class must define behaviors common to all members of the hierarchy, ensuring that derived class objects can be safely used in place of base class objects.

Class Collaboration	Inheritance
has-a	is-a

3.4.3 Function Overriding

Function overriding is a fundamental feature of polymorphism. It allows a derived class to redefine a function from the base class, ensuring that the most specific implementation is used at runtime.

```
class Base {
public:
    virtual void display() {
        std::cout << "Base class." << std::endl;
    }
};

class Derived : public Base {
public:
    void display() override { // 'override' is recommended for clarity.
        std::cout << "Derived class." << std::endl;
    }
};

Base *ptr = new Derived();
ptr->display(); // Calls Derived::display() dynamically.
```

Tip: ***Function Overriding vs. Overloading***

Function overriding should not be confused with function overloading. Overloading allows multiple functions with the same name but different parameters, whereas overriding replaces an inherited method with a new implementation.

Virtual Methods and Dynamic Dispatch

The `virtual` keyword indicates that a method can be overridden by derived classes. When such methods are called, **dynamic dispatch** selects the appropriate implementation based on the object's actual type during program execution.

```
class Polygon {
public:
    virtual double area() { return 0.0; } // Base implementation.
};

class Square : public Polygon {
public:
    double area() override { return side * side; }
private:
    double side;
};

void f(const Polygon &p) {
    const double a = p.area(); // Calls the correct area() dynamically.
}

Square s;
f(s); // Upcasting: 'Square' is treated as 'Polygon'.
```

Warning: ***Polymorphism, Pointers, and References***

Polymorphism applies only when using pointers or references. Passing objects by value leads to slicing, where derived class-specific behavior is lost.

```
void f(Polygon p) { // Passed by value (bad practice).
    const double a = p.area(); // Calls Polygon::area(),
                               // not Square::area().
}

Square s;
f(s); // Object slicing: Square is reduced to a Polygon.
```

Factory Pattern with Polymorphism

Polymorphism allows for flexible object creation based on runtime conditions.

```
unsigned int n_sides;

std::cout << "Number of sides: ";
std::cin >> n_sides;

Polygon *p;
if (n_sides == 3)
    p = new Triangle{...};
else if (n_sides == 4)
    p = new Square{...};
else {
    // Handle other cases...
}

std::cout << "Area: " << p->area() << std::endl;

delete p; // Always delete allocated memory!
```

3.4.4 Virtual Destructors

For any class that serves as a base class in a polymorphic hierarchy, implementing a `virtual` destructor is essential to guarantee proper cleanup of derived class resources.

```
class Polygon {
public:
    virtual ~Polygon() {} // Virtual destructor
};
```

Without a virtual destructor, deleting a derived class object through a base class pointer leads to **undefined behavior**.

```
// Polygon has no virtual destructor.
Polygon *p = new Square();
// ...
delete p; // Calls ~Polygon(), causing potential memory leaks.
```

Tip: *Compiler Warning*

Use `-Wnon-virtual-dtor` with your compiler to receive warnings when a base class destructor should be virtual.

When is a Virtual Destructor Unnecessary?

A virtual destructor is not required if:

- The class is not used polymorphically (i.e., objects are never accessed through base class pointers or references).
- The base class does not introduce new members that require cleanup.

Protected and Private Polymorphism

Private polymorphism is a form of **private inheritance** where `virtual` methods are overridden without allowing access to them through a base class pointer. Similarly, **protected polymorphism** is a form of **protected inheritance** where `virtual` methods are overridden, but external calls through a base class pointer are restricted.

However, both private and protected inheritance are not commonly used for polymorphism. **Polymorphism in C++ is typically associated with public inheritance**, which preserves the *is-a* relationship and ensures that derived classes maintain the interface of the base class.

Observation: ***When to Use Protected and Private Inheritance***

- Use **protected inheritance** when you want to limit polymorphism to derived classes.
- Use **private inheritance** when you need implementation reuse but not polymorphism.

3.4.5 Selective Inheritance

Sometimes, you may want to expose only part of the base class interface in a derived class. This can be done using `using`:

```
class Base {
public:
    double fun(int i);
    // Other methods...
};

class Derived : private Base {
public:
    using Base::fun; // Only fun() is exposed publicly.
};
```

This allows controlled exposure of base class methods while hiding the rest of the interface.

Tip: ***Best Practices for Polymorphism***

- Always use `override` in derived classes for clarity.
- Use `virtual` destructors in polymorphic base classes.
- Prefer reference or pointer parameters when working with polymorphic objects.
- Consider selective inheritance for controlled interface exposure.

3.5 Advanced Polymorphism Concepts

3.5.1 Abstract Classes

In some cases, a base class represents an abstract concept, and it does not make sense to instantiate concrete objects of that type. Instead, the base class defines the common public interface for a hierarchy but does not fully implement it.

For this purpose, C++ introduces the concept of an **abstract class**, which is a class that contains at least one **pure virtual function**. A pure virtual function is declared using the `= 0` syntax.

```
class Shape {
public:
    virtual double area() = 0; // Pure virtual method
};

Shape s; // Illegal! Cannot instantiate an abstract class.
```

Characteristics of Abstract Classes

- An **abstract class** cannot be instantiated. It serves as a blueprint for other classes and enforces a common interface for its derived classes.
- Abstract classes contain at least one pure virtual function, which lacks implementation in the base class and is marked with `= 0`.
- They can have regular member functions with implementations and data members.
- Derived classes that inherit from an abstract class must implement **all** pure virtual functions in order to become concrete (instantiable) classes.
- Pure virtual functions act as placeholders for functionalities that must be provided by derived classes. They enforce a specific method signature that derived classes must follow.

Implementing an Abstract Class

When deriving from an abstract class, all pure virtual functions must be implemented in the derived class. The derived class becomes concrete (instantiable) only when all pure virtual functions are implemented. Otherwise, it remains abstract.

```
class Triangle : public Shape {
public:
    double area() override { return 0.5 * base * height; }
private:
    double base;
    double height;
};

class Square : public Shape {
public:
    double area() override { return side * side; }
private:
    double side;
};

Triangle t{1.5, 3.0};   // Legal
Square s{0.5};          // Legal
```

3.5.2 The `final` and `override` Specifiers

C++ provides two important specifiers to improve the clarity and correctness of inheritance hierarchies: `final` and `override`.

The `final` specifier prevents further overriding of a virtual method or inheritance of a class, ensuring that the implementation remains unchanged in derived classes. When applied to a class, it forbids any further derivation, and when used with a virtual function, it ensures that no subclass can override that function.

```
class A {
public:
    virtual void f() final;
    virtual double g(double);
};

class B final : public A {
public:
    void f() override; // Error: f() is final in A.
};

class C : public B // Error: B is final.
{
// ...
};
```

The `override` specifier ensures that the method in the derived class correctly overrides the base class method. By explicitly stating this intent, any signature mismatches will trigger a compile-time error, thereby enhancing code safety and clarity.

```
class A {
    virtual void f();
    void g();
};

class B : public A {
    void f() const override; // Error: Signature does not match A::f().
    void f() override; // OK: Correctly overrides A::f().
    void g() override; // Error: g() is not virtual in A.
};
```

Tip: *Compiler Tip*

Although `override` is not mandatory, it is strongly recommended to catch potential errors during compilation; moreover it improves code safety and clarity.
The compiler option `-Wsuggest-override` can be used to generate warnings when the `override` specifier is missing.

3.5.3 Type Casting in Polymorphic Hierarchies

In polymorphic class hierarchies, it is often necessary to determine an object's actual type at runtime or safely convert between base and derived class pointers. C++ provides mechanisms such as **Run-Time Type Information (RTTI)** and `dynamic_cast` to achieve this.

Run-Time Type Identification (RTTI) and `typeid`

RTTI allows the program to determine the actual type of an object at runtime, which is particularly useful in polymorphic hierarchies. The `typeid` operator returns a type descriptor that can be inspected or compared.

```
#include <typeinfo>

class Base {
public:
    virtual void print() { std::cout << "Base class" << std::endl; }
};

class Derived : public Base {
public:
    void print() override { std::cout << "Derived class" << std::endl; }
};

// in main()
Base base;
Derived derived;
std::cout << "Type of base: " << typeid(base).name() << std::endl;
// Output: Type of base: 4Base
std::cout << "Type of derived: " << typeid(derived).name() << std::endl;
// Output: Type of derived: 7Derived
```

Since `typeid` queries the actual type of an object, it requires at least one `virtual` function in the base class. Otherwise, it only provides the static type determined at compile time.

Type Checking with `dynamic_cast`

The `dynamic_cast` operator ensures safe downcasting in polymorphic hierarchies. It performs a runtime check to verify whether a base class pointer can be converted to a pointer of a derived class.

```
double compute_area(const Shape& p) {
    auto ptr = dynamic_cast<const Square*>(&p);
    if (ptr != nullptr) {
        return ptr->get_side() * ptr->get_side(); // Safe only if p is a
            Square.
    } else {
        return p.area();
    }
}
```

If the conversion is not valid:

- For pointer conversions, `dynamic_cast` returns `nullptr` to signal a failed cast.
- For reference conversions, a failed `dynamic_cast` throws a `std::bad_cast` exception; ensure you handle this case using a `try-catch` block.

3.5.4 Aggregation vs. Composition with Polymorphic Objects

When a class aggregates a polymorphic object, a key design decision is whether it should take ownership of the object's lifetime.

```cpp
class Prism {
public:
    // 1) Aggregation: Takes an external object, does not manage its
       lifetime.
    Prism(Shape* s) : shape{s} {}

    // 2) Composition: Creates and manages the object's lifetime.
    void init_as_square(double side) {
        poly_ptr = new Square{side};
    }
    ~Prism() { delete poly_ptr; }

private:
    Shape* shape;  // Aggregation (does not own object)
    Shape* poly_ptr = nullptr; // Composition (owns object)
};
```

3.5.5 Best Practices in Polymorphic Hierarchies

Tip: ***Guidelines for Effective Use of Polymorphism***

- **Follow the Liskov Substitution Principle (LSP)**: Derived classes must be interchangeable with their base class without altering program behavior.
- **Use virtual destructors**: If a class is intended to be used polymorphically, its destructor should be declared as `virtual`.
- **Prefer composition over inheritance**: Avoid deep inheritance chains; favor composition when possible.
- **Avoid excessive downcasting**: Rely on virtual functions instead of frequent use of `dynamic_cast`.
- **Encapsulation**: Keep base class members `protected` or `private` and provide controlled access through public methods.

4 Advanced Topics on Functions

4.1 Callable Objects

A **callable object** is any entity that can be invoked using the function call operator `()`.
Callable objects in C++ include:

- **Functions** (free or member functions).
- **Function pointers**.
- **Member function pointers**.
- **Functors (function objects)**.
- **Lambda expressions** (introduced in C++11).
- **Function wrappers** (`std::function`).

4.1.1 Functions

In C++, functions can be categorized as **free functions** (standalone functions) or **member functions** (methods of a class). Additionally, functions can be classified as **normal functions** or **template functions**, including functions that employ automatic return type deduction.
For a free, non-template function, the typical declaration syntax is:

```
return_type function_name(...);

auto function_name(...) -> return_type;

auto function_name(...); // The compiler deduces the return type.
```

The first two forms are functionally equivalent, while the third allows the compiler to deduce the return type automatically.

Function Identifiers
A function in C++ is uniquely identified by:

- Its **name**, which includes its fully qualified name (e.g., `std::transform`).
- The number and type of its **parameters**.
- The presence of the **const** qualifier (for member functions).
- The type of the enclosing class (for member functions).

Warning: *Return Type and Function Identification*

The return type is **not** part of a function's identifier. Functions cannot be overloaded based solely on differing return types.

Why Use Free Functions?
Free functions represent pure mappings from inputs to outputs, adhering to the mathematical notion of a function, $f : U \to V$. Since they are generally **stateless** (except when using `static` variables), the same input always produces the same output.
Key advantages include:

- **Clarity:** They model functions in a mathematical sense without side effects.
- **Modularity:** They separate functionality from class definitions, enhancing code reusability.
- **Simplicity:** Their lack of state minimizes dependencies and simplifies debugging.

Returning Values from Functions

Returning a reference rather then a value avoids creating a copy of the object, which can be more efficient, especially for large objects.

```
std::ostream &operator<<(std::ostream &os, const MyClass &obj) {
    // ...
    return os;
}

std::cout << x << " concatenated with " << y; // Enables chaining.
```

Returning a reference is often used in operator overloading (e.g., `<<`) to enable chaining and in methods that modify an object and return it for further operations.

Warning:

Be cautious when returning references, as the object being referred to must outlive the reference. Returning a reference to a local variable, for example, leads to undefined behavior.

Default Parameters

Default values can be assigned to function parameters, reducing redundancy in function calls and simplifying the interface for users who only need the most common functionality.

```
std::vector<double> cross_prod(const std::vector<double> &a,
                               const std::vector<double> &b,
                               const unsigned int ndim = 2); // Default
// Usage:
a = cross_prod(c, d); // 'ndim' defaults to 2.
```

Observation: ***Default Parameters and Function Overloading***

We remember that function overloading allows multiple functions to share the same name, provided they have different signatures. Default parameters can be used to achieve the same effect without creating multiple functions.

4.1.2 Function Pointers

Function pointers allow passing functions as arguments and selecting functions dynamically at runtime, offering a level of indirection that enhances code modularity. They are particularly useful when the specific function to be executed is not known at compile time.

```
double integrand(double x);

using f_ptr = double (*)(double); // Function pointer type
double integrate(double a, double b, const f_ptr fun);

double I = integrate(0, 3.14, integrand); // Passing function as pointer
```

The name of the function is interpreted as a pointer to that function. However, you may precede it by `&`: `f_ptr f = &integrand`.

Function pointers are useful, but `std::function` offers greater flexibility and type safety as it can encapsulate any callable object (including function pointers, lambdas, and function objects) with a compatible signature.

Dynamic Function Selection

You can use function pointers to select and call functions at runtime based on runtime conditions.

```
int add(int a, int b) { return a + b; }
int subtract(int a, int b) { return a - b; }

int (*operation)(int, int);

if (user_input == "add") { // User input determines operation.
    operation = add;
} else {
    operation = subtract;
}

const int result = operation(10, 5); // Calls selected function.
```

Member Function Pointers

Member function pointers provide a way to call member functions (methods) of a class through a pointer. Unlike regular function pointers, member function pointers require an object instance to operate on because they are bound to a specific class. This allows you to call different implementations of a virtual function based on the object's actual type at runtime, or to store and invoke methods dynamically.

```
std::vector<Shape*> shapes = {new Circle(3.0), new Rectangle(2.0, 4.0)};

double (Shape::*area_fun)() const = &Shape::area;

for (const auto shape : shapes) {
    const double area = (shape->*area_fun)();
    std::cout << "Area: " << area << std::endl;
}
```

4.1.3 Functors (Function Objects)

A **functor** (function object) is a class that overloads the function call operator `operator()`. This makes instances of the class callable like functions. Functors can maintain state, which distinguishes them from simple functions, and are often used to pass operations to algorithms, providing more flexibility than function pointers or lambdas.

```
class Cube {
public:
    double operator()(const double &x) const { return x * x * x; }
};

Cube cube;
auto y = cube(3.4);   // Calls cube.operator()(3.4)
auto z = Cube{}(8.0); // Creates a temporary Cube object and
                      //     calls its operator()(8.0)
```

If `operator()` returns a `bool`, the functor is often referred to as a **predicate**. Declaring `operator()` as `const` ensures that calling the functor does not modify its internal state, which is good practice for functors that represent stateless operations.

One of the key advantages of functors is their ability to **maintain state across multiple calls**. This allows them to be used in situations where a regular function would not be sufficient. For example:

```
class Calculator {
public:
    int result = 0;

    class Add {
    public:
        Calculator& calc;
        Add(Calculator& c) : calc(c) {}
        void operator()(int x, int y) { calc.result = x + y; }
    };
};

// in main
Calculator calc;
Calculator::Add add(calc);
add(5, 3); // Calls add.operator()(5, 3), result stored in calc.result.
```

The `Add` functor stores the result in the `Calculator` object's `result` member. This behavior is not possible with regular functions without using global variables or other less desirable approaches. Functors are particularly useful with standard library algorithms like `std::transform` and `std::for_each`, where they can be used to apply a stateful operation to a range of elements.

Predefined Functors in the STL

The STL provides predefined functors under `<functional>`. These functors perform common operations like addition, subtraction, multiplication, division, and logical operations.

```
std::vector<int> in = {1, 2, 3, 4, 5};
std::vector<int> out;

std::transform(
    in.begin(), in.end(),      // Source
    std::back_inserter(out),   // Destination
    std::negate<int>()         // Negates each element
);
```

Using predefined functors reduces the need to write custom function objects for simple operations, leading to more concise and readable code.

Functor	Description
`plus<T>, minus<T>`	Addition/Subtraction (Binary)
`multiplies<T>, divides<T>`	Multiplication/Division (Binary)
`modulus<T>`	Modulus (Unary)
`negate<T>`	Negative (Unary)
`equal_to<T>, not_equal_to<T>`	(Non-)Equality Comparison (Binary)
`greater , less , greater_equal , less_equal`	Comparison (Binary)
`logical_and<T>, logical_or<T>, logical_not<T>`	Logical AND/OR/NOT (Binary)

Table 4.1: Some predefined functors in the STL. [**cplusplus**]

4.1.4 Lambda Functions

C++ supports a streamlined syntax for defining anonymous functions known as **lambda expressions**. These inline functions allow you to write short, self-contained functions directly where they are needed, eliminating the overhead of separately declaring and defining a function. Lambda expressions are analogous to Python's `lambda` functions or MATLAB's anonymous functions but integrate closely with C++'s type deduction and template mechanisms.

```
auto f = [](double x) { return 3 * x; }; // Lambda function.

auto y = f(9.0); // y = 27.0.
```

In these examples, the compiler deduces the return type automatically from the lambda's return statement, making the code both succinct and expressive.

Capture Specification

Lambda expressions can capture variables from their surrounding scope in various ways. This capture mechanism allows the lambda to access and use local variables without needing to pass them as parameters.

The available capture options include:

- `[]`: Captures no variables.
- `[&]`: Captures all variables from the enclosing scope by reference.
- `[=]`: Captures all variables from the enclosing scope by copy.
- `[y]`: Captures the variable `y` by copy.
- `[&y]`: Captures the variable `y` by reference.
- `[=, &x]`: Captures all variables by copy except for `x`, which is captured by reference.
- `[this]`: Captures the `this` pointer, thereby providing access to all members of the enclosing class.
- `[*this]`: Captures a copy of the entire enclosing object.

These capture options provide flexibility when designing lambdas, enabling you to control how data is accessed and modified within the lambda's body.

Using `[this]`

Capturing `this` allows a lambda to access and modify members of its enclosing class. This is particularly useful when working within member functions, as it enables concise operations over class variables using standard algorithms or other lambda-driven constructs.

```
class MyClass {
public:
    void compute(double a) {
        auto prod = [this, &a]() { x *= a; };
        std::for_each(v.begin(), v.end(), prod);
    }
private:
    double x = 1.0;
    std::vector<double> v;
};
```

Here, `compute()` uses the lambda `prod` that changes the member `x`. To be more explicit, you can write `this->x *= a;`

4.1.5 Function Wrappers (`std::function`)

A **function wrapper** is a polymorphic wrapper that can store *any* callable object, including free functions, function pointers, functors, and lambdas. The standard C++ library provides this capability through `std::function` (declared in `<functional>`). By using `std::function`, you can treat different callable objects as first-class citizens with a uniform interface.

```
#include <functional>
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::function<int(int,int)> func = add;  // Wraps a free function
    const int result = func(2, 3);
    std::cout << "Result: " << result << std::endl;  // Prints 5
    return 0;
}
```

Using `std::function` is extremely helpful when you need a generic, uniform way to store or pass around callable entities of different types.

Warning: *Performance Overhead*

Although `std::function` is very convenient, be aware that it introduces a small level of indirection (and hence some overhead), because the callable object is stored internally via type-erasure. In most cases, this overhead is negligible compared to the flexibility gained.

Function Wrappers and Polymorphism

The `std::function` wrapper is not limited to functions that share primitive argument types. It can also handle more complex scenarios, such as virtual member functions. The following example demonstrates how `std::function` can be used alongside polymorphic classes:

```
#include <functional>
#include <iostream>

class Shape {
public:
    virtual double area() const = 0;
    virtual ~Shape() = default;
};

class Circle : public Shape {
public:
    explicit Circle(double r) : radius(r) {}
    double area() const override {
        return 3.14159265359 * radius * radius;
    }
private:
    double radius;
};
```

```cpp
int main() {
    // A lambda that calls the virtual method area() on a Shape
    std::function<double(const Shape&)> compute_area =
        [](const Shape& s){ return s.area(); };

    Circle circle(5.0);
    std::cout << "Circle area: " << compute_area(circle) << std::endl;
    return 0;
}
```

Here, the lambda captures nothing but simply invokes `area()` on the passed `Shape`. Under the hood, `std::function` can store this lambda and call it later as needed.

A Vector of Functions

Because `std::function` can wrap any callable type with a given signature, we can store multiple, different callables in a single container:

```cpp
#include <functional>
#include <iostream>
#include <vector>

int func(int x, int y) { return x + y; }

class F2 {
public:
    int operator()(int x, int y) const {
        return x - y;
    }
};

int main() {
    std::vector<std::function<int(int,int)>> tasks;

    // Wrap a free function
    tasks.push_back(func);

    // Wrap a functor
    F2 func2;
    tasks.push_back(func2);

    // Wrap a lambda
    tasks.push_back([](int x, int y){ return x * y; });

    for (auto &t : tasks) {
        std::cout << t(3, 4) << std::endl;
    }
    return 0;
}
```

In this loop, each callable is invoked as `t(3, 4)`, printing:

$$\text{func}(3,4) = 7, \quad \text{func2}(3,4) = -1, \quad 3 \times 4 = 12.$$

`std::bind` and Function Adapters

The C++ standard library provides `std::bind` (also in `<functional>`) to create *adapted* or *partially applied* functions. It allows you to fix (bind) certain parameters of a function, producing a new callable that can be invoked with fewer parameters.

```
#include <functional>
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    // Create a new function that binds 'a' to 5
    std::function<int(int)> add5 =
        std::bind(add, 5, std::placeholders::_1);

    std::cout << add5(3) << std::endl; // Prints 8
    return 0;
}
```

Observation: *Modern Alternatives*

Although `std::bind` is powerful, modern C++ often prefers **lambdas** for readability. Still, `std::bind` is quite useful in complex scenarios or when you need to systematically reuse partially-applied arguments.

4.2 Generic Programming and Templates

Generic programming is a **paradigm** that aims to write code that is independent of any particular data type, making it reusable, type-safe, and efficient. C++ achieves generic programming primarily through *templates*, which allow you to write code that is instantiated for different data types as needed.

4.2.1 Overview of Generic Programming

Concept and Motivation.

Generic programming focuses on designing algorithms and data structures in a way that they can operate on a broad range of types. Rather than writing multiple versions of the same functionality for each type, you write a *single* version of an algorithm (e.g., `sort`) and let the compiler automatically generate the correct variant for an *int* array, a *double* array, a *std::string* array, and so on.

Key Benefits.

- **Reusability:** Write the code once and reuse it for multiple types.
- **Type Safety:** Many errors that involve type mismatches are caught at compile time.
- **Performance:** The compiler generates code specialized for each type, often comparable to handwritten type-specific code.
- **Expressiveness:** Templates reduce boilerplate and improve readability, as the type details are factored out.

Examples and Use Cases.

- **STL (Standard Template Library):** Provides generic containers (`std::vector`, `std::list`, …) and algorithms (`std::sort`, `std::accumulate`, …).
- **Function Templates:** Write one function that works for any type supporting the required operations (e.g., `operator+`).
- **Class Templates:** Implement a type-safe container or data structure (e.g., `std::stack`).
- **Custom Data Structures and Algorithms:** From binary trees to mathematical computations, templates give great flexibility.

4.2.2 Function Templates

Example: *Function Templates*

A function like `less_than` for multiple types leads to code duplication:

```
1 bool less_than(char x1, char x2) { return (x1 < x2); }
2 bool less_than(double x1, double x2) { return (x1 < x2); }
```

Since the logic remains unchanged, a **function template** generalizes the function:

```
1 template <typename T>
2 bool less_than(T x1, T x2) { return (x1 < x2); }
```

When calling `less_than(3.0, 2.5)`, the compiler generates `less_than<double>(double, double)` automatically. You can also specify the template argument explicitly: `less_than<int>(10, 5)`.

Defining and Using Function Templates

Function templates allow generic operations on different types without redundancy. Here's an example of an addition function:

```
1 template <typename T>
2 T add(T a, T b) { return a + b; }
3
4 int main() {
5     int i = add(5, 3);         // T deduced as int
6     double d = add(2.5, 3.7); // T deduced as double
7 }
```

Default Template Parameters

Templates support default parameters to simplify function calls:

```
1 template <typename T, typename U = double>
2 T multiply_and_add(T a, U b, T c) {
3     return a * b + c;
4 }
5
6 // T is int; U defaults to double
7 int result = multiply_and_add(5, 2.5, 3);
```

Template Specialization

Sometimes, a general template needs custom behavior for specific types:

```
template <typename T>
T max(T a, T b) { return (a > b) ? a : b; }

// Specialization for char: compare in uppercase
template <>
char max(char a, char b) {
    return (std::toupper(a) > std::toupper(b)) ? a : b;
}
```

Example: *Summing Elements in a Vector*

Templates can work with different types as long as necessary operators exist:

```
template <typename T>
T vector_sum(const std::vector<T>& vec) {
    T sum{};
    for (const auto &elem : vec) sum += elem;
    return sum;
}
```

This function sums numeric values or concatenates strings if `operator+` is defined.

Key Properties of Templates

- **Lazy Instantiation:** Templates are compiled only when instantiated.
- **Late Error Detection:** Compilation errors may arise only when the template is used.

Constant values as template parameters

You can give defaults to the rightmost parameters (this applies also).
A template parameter may also be an integral value.

```
template <int N, int M = 3>
constexpr int multiply() {
    return N * M;
}

constexpr int result1 = multiply<5>();    // calc. 5 * 3 at compile-time
constexpr int result2 = multiply<2, 7>(); // calc. 2 * 7 at compile-time
```

Only integral types can be used (e.g., integers, enumerations, pointers, ...).
`constexpr` can be applied to variables, functions, and constructors, to ensure that they are evaluated at compile time.

Advantages and Disadvantages of Generic Programming

- **Pros:** Code reuse, type-safety, efficiency, and cleaner code.
- **Cons:** Complex error messages, longer compile times, and difficult debugging in advanced cases.

4.2.3 Class Templates

Introduction

A **class template** generalizes an entire class definition for use with arbitrary data types. Instead of creating separate classes, such as `IntList`, `DoubleList`, or `StringList`, a single *blueprint* can adapt to any type. Standard library containers like `std::vector<T>` or `std::map<K,V>` are prime examples of class templates.

```
template <typename T>
class List {
public:
    // ...
private:
    T value;
    List* next;
};
```

In this example, `List` is a template that can hold objects of type `T`. The compiler will generate distinct `List` classes internally, depending on the specific `T` you use in your code.
When you want to create instances of a class template, you must *instantiate* it by providing the required type parameters:

```
List<int> list_int;       // List of int
List<double> list_double; // List of double
```

Here, `T` is deduced to be `int` for the first instantiation and `double` for the second. Each instantiation behaves as though you wrote a specialized `List` class for that type.

Class Template Specialization

Sometimes, the general-purpose implementation is not sufficient or needs distinct logic for a particular type. In these cases, you can *specialize* a class template:

```
template <typename T>
class Vector {
    // Generic version of Vector<T>
};

// Full specialization for std::string
template <>
class Vector<std::string> {
    // Specialized version for std::string
};
```

Whenever `Vector<std::string>` is instantiated, the specialized version is used instead of the generic one.

Partial Specialization

Partial specialization refines the template when some, but not all, of the template parameters are fixed. This is particularly useful when you want a custom implementation for a certain subset of parameters (e.g., arrays of size 1):

```cpp
// Generic template
template <typename T, int N>
class Array {
private:
    T data[N];
};

// Partial specialization for arrays of size 1.
template <typename T>
class Array<T, 1> {
private:
    T element; // No need to store an array for a single variable!
};

Array<int, 3> arr1;  // Generic template for arrays of size 3
Array<char, 1> arr2; // Partially specialized template if size is 1
```

Template Aliases

A **template alias** allows you to define a simpler or more meaningful name for a complex template. This improves readability and helps avoid repeated verbose declarations:

```cpp
template <typename T, int N>
class Array {
    T data[N];
};

// Alias for an Array of int
template <int N>
using IntArray = Array<int, N>;

IntArray<5> arr; // Creates an Array<int, 5>
```

Here, `IntArray<5>` is equivalent to `Array<int, 5>`, but shorter and more expressive in context.

Tip: ***Best Practices***

- `Use descriptive parameter names:`
  For clarity, prefer using `template <typename Key, typename Value>` instead of using `template <typename T, typename U>`, especially if the roles of those parameters are distinct.
- **Only specialize when necessary:**
  Overusing specializations can clutter code. Specialize class templates only if the type truly needs unique behavior.
- **Organize definitions sensibly:**
  Placing large definitions in a separate `.tpl.hpp` (or similar) file improves readability, but remember that templates must be visible where they're instantiated.

Example: *Templated Stack Class*

To see a more concrete scenario, consider a simple stack implementation where `T` represents the element type:

```
template <typename T>
class Stack {
public:
    void push(const T &value) {
        elements.push_back(value);
    }

    T pop() {
        if (elements.empty()) {
            std::cerr << "Stack is empty\n";
            std::exit(1);
        }
        T top = elements.back();
        elements.pop_back();
        return top;
    }

private:
    std::vector<T> elements;
};
```

This `Stack<T>` can store and manage elements of type `T`. For instance, you can instantiate `Stack<int>` or `Stack<std::string>` without rewriting any logic.

```
Stack<int> int_stack;
int_stack.push(10);
int_stack.push(20);
std::cout << int_stack.pop() << std::endl; // Prints 20
```

4.2.4 Notes on Code Organization

Template Instantiation and Linkage

The `compiler` produces the code corresponding to **function templates** and **class template members** that are instantiated in each translation unit. It means that all translation units that contain, for instance, an instruction of the type `std::vector<double> a;` produce the machine code corresponding to the default constructor of a `std::vector<double>`. If we then have `a.push_back(5.0)`, the code for the `push_back` method is produced, and so on. If the same is true in other compilation units, the same machine code is produced several times. It is the `linker` that eventually produces the executable by selecting only one instance.

The Core Problem

Template definitions need to be available at the point of instantiation. When a template is used with specific type arguments, the compiler needs to see the template definition to generate the code for that particular instantiation. Placing the template definition in a source file would make it unavailable for instantiation in other source files.

If you place template definitions in source files and use the template in multiple source files, you may encounter linker errors due to multiple definitions of the same template. **Placing the template definitions in a header file** ensures that the definition is available for all source files that include it, and the linker can consolidate the definitions as needed.

Possible File Organizations

1. **Leave everything in a header file.** However, if the functions/methods are long, it may be worthwhile, for the sake of clarity, to separate definitions from declarations. You can put declarations at the beginning of the file and only short definitions. Then, at the end of the file, add the long definitions for readability.
2. **Separate declarations `module.hpp` and definitions `module.tpl.hpp`** when templates are long and complex. Then add `#include "module.tpl.hpp"` at the end of `module.hpp` (before closing its header guard).
3. **Explicitly instantiate for a specific list of types.** Only in this case, definitions can go to a source file. But if you instantiate a template for other types not explicitly instantiated, the compiler will not have access to the definition, leading to linker errors.

Explicit Instantiation

Explicit instantiation tells the compiler to produce code for selected template instantiations in one translation unit. If a source file contains, for instance:

```
// In some .cpp file
template double func(const double&);
template class MyClass<double>;
template class MyClass<int>;
```

then the corresponding object file will contain the code corresponding to the template function `double func<T>(const T &)` with `T=double` and that of all methods of the template class `MyClass<T>` with `T=double` and `T=int`.

This can be useful to save compile time when debugging template classes (since the code for all class methods is generated).

4.2.5 Advanced Template Techniques and Concepts

Type Deduction and `auto`

Type deduction allows the compiler to determine the data type of variables and return values automatically.
Using `auto` can simplify code by letting the compiler deduce variable types:

```
template <typename T>
T add(T a, T b) { return a + b; }

int main() {
    auto x = add(5, 3);       // Deduces T as int
    auto y = add(2.5, 1.5);   // Deduces T as double
}
```

While `auto` is convenient, overusing it can harm readability. Aim for a balance where type deduction does not obscure intent.

Name Lookup and `this` in Templates

When working with class templates, the compiler resolves non-template names immediately, while names dependent on the template parameter are resolved only when the class is instantiated. This distinction can lead to issues in name lookup, particularly when inheriting from a base class template. Consider the following example:

Example: *Problem: Ambiguous Name Lookup*

```
void my_fun() { /* Global function */ }

template <typename T>
class Base {
public:
    void my_fun(); // Member function in Base<T>
};

template <typename T>
class Derived : public Base<T> {
public:
    void foo() {
        my_fun(); // Ambiguity: calls global my_fun(), not Base<T
           >::my_fun()
    }
};
```

Since `Base<T>` is a template, `my_fun()` is not immediately recognized as a member of `Base<T>`, and the compiler assumes it refers to the global function instead. To explicitly indicate that `my_fun()` belongs to the base class, we must use either:

- The `this->` pointer.
- A qualified name like `Base<T>::my_fun()`.

Using `this->my_fun()` ensures that the compiler correctly recognizes `my_fun()` as a member function of `Base<T>`.

```
template <typename T>
class Base {
public:
    void foo() {}
};

template <typename T>
class Derived : public Base<T> {
public:
    void bar() {
        this->foo();       // Explicitly accesses Base<T>::foo()
        Base<T>::foo();    // Alternative qualified name
    }
};
```

Using `this->foo()` ensures that the compiler correctly recognizes `foo()` as a member function of `Base<T>`. This technique is essential for avoiding name lookup ambiguities in template-based inheritance.

Template Template Parameters

Template template parameters let you pass an entire template as a parameter to another template. This technique is useful for creating flexible and reusable code that can work with a variety of template classes.

```
template <typename T, template <typename> class C = std::complex>
class MyClass {
private:
    C<T> a;
};

MyClass<double, std::vector> x; // 'a' is a std::vector<double>.

MyClass<int> x; // 'a' is a std::complex<int>.
```

This feature allows to write expressions like `std::vector<std::complex<double>>`.
In general, this technique helps when you want to provide a configurable `template` type, like a custom allocator or a specific internal container.

Template Metaprogramming and SFINAE

Template metaprogramming performs compile-time computations, enabling highly optimized or specialized code. **SFINAE** (*Substitution Failure Is Not An Error*) is a related rule that selectively enables or disables template overloads based on whether certain expressions are valid. This technique is often used to implement type traits or to enable/disable functions based on type properties.
The following example defines a type trait `has_print<T>` that determines whether a given type `T` has a member named `print`.

Example: *Type Traits with SFINAE*

```
template <typename T>
class has_print {
public:
    template <typename U>
    static std::true_type test(decltype(U::print)*);

    template <typename U>
    static std::false_type test(...);

    static constexpr bool value = decltype(test<T>(0))::value;
};

class MyType {
public:
    void print() {}
};

std::cout << std::boolalpha;
std::cout << has_print<MyType>::value << std::endl; // true
std::cout << has_print<int>::value << std::endl;    // false
```

The function `test` is overloaded:

- The first version takes `decltype(U::print)*`, which is valid only if `U` has a member `print`, selecting `std::true_type`.
- The second version is a fallback that catches all other cases, returning `std::false_type`.

By calling `test<T>(0)`, the compiler selects the appropriate overload, determining whether `T` has `print` at compile time.

Example: *Fibonacci with Template Metaprogramming*

```
template <int N>
class Fibonacci {
public:
    static constexpr int value = Fibonacci<N - 1>::value +
       Fibonacci<N - 2>::value;
};

template <>
class Fibonacci<0> {
public:
    static constexpr int value = 0;
};

template <>
class Fibonacci<1> {
public:
    static constexpr int value = 1;
};

constexpr int n = Fibonacci<10>::value; // Calc. at compile-time.
```

Variadic Templates

Variadic templates allow functions and classes to accept a variable number of arguments. The `...` syntax is used to define them. This feature is particularly useful for functions like `printf` or `std::make_tuple` that can accept an arbitrary number of arguments.

```
template <typename T>
T sum(T value) {
    return value;
}

template <typename T, typename... Args>
T sum(T first, Args... rest) {
    // Consume the first argument, then recurse over remaining arguments
    return first + sum(rest...);
}
```

This allows elegant handling of calls like `sum(1, 2, 3, 4.5, 6)` without writing multiple overloads.

CRTP (Curiously Recurring Template Pattern)

CRTP is a design pattern where a derived class inherits from a base class template with itself as the template argument. This pattern enables static polymorphism, allowing the base class to access and manipulate the derived class's members.

```
template <typename Derived>
class Shape {
public:
    double area() {
        return static_cast<Derived*>(this)->area();
    }
};

class Circle : public Shape<Circle> {
public:
    double area() {
        // Compute area of a circle.
    }
};

Circle c;
c.area();
```

CRTP allows the Shape class to know the interface of its derived class at compile time, enabling static (static) **compile-time polymorphism**, eliminating virtual function overhead for certain design scenarios.

Traits and Policy-Based Design

Traits classes (e.g., `std::is_integral<T>`) are used to encapsulate properties and behaviors of types. They are often used in policy-based design, where different behaviors are selected at compile time through template parameters.

Example: *Type Trait Check*

```
#include <type_traits>
#include <iostream>

template <typename T>
void process_type(T value) {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Integral type\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Floating-point type\n";
    } else {
        std::cout << "Other type\n";
    }
}
```

By using `if constexpr` (introduced in C++17), the compiler evaluates the condition at compile time, creating specialized code paths without runtime overhead.

5 Standard Template Library

The Standard Template Library (STL) is a collection of C++ template classes that provide general-purpose data structures and algorithms, including `std::vector`, `std::list`, `std::queue`, and `std::stack`. The STL consists of four main components:

- **Algorithms**: A collection of functions designed to operate on ranges of elements.
- **Containers**: Objects that store collections of other objects.
- **Function Objects**: Components that allow the creation of callable objects.
- **Iterators**: Objects that enable traversal of a container.

5.1 Containers

Containers are objects that store data. The STL provides several container classes, each supporting different operations. STL containers are categorized as follows:

- **Sequence Containers**: These containers maintain ordered collections of elements. The main sequence containers are `std::vector`, `std::list`, and `std::deque`.
- **Associative Containers**: These containers store elements in sorted order and support fast searching. The main associative containers are `std::set`, `std::multiset`, `std::map`, and `std::multimap`.
- **Container Adaptors**: These provide restricted access to elements by adapting existing containers. The primary container adaptors are extttstd::stack, extttstd::queue, and extttstd::priority_queue.
- **Special Containers**: These containers provide specialized functionality. Examples include `std::byte`, `std::pair`, `std::tuple`, `std::variant`, `std::optional` and `std::any`.

5.1.1 Sequence Containers

Sequence containers maintain ordered collections of elements, with element positions independent of their values. In `std::vector` and `std::array`, elements are stored contiguously in memory and can be accessed using an index `[]`.
Examples of sequence containers include `std::vector<T>`, `std::array<T,N>`, `std::deque<T>`, and `std::list<T>`.

Example:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = {1, 2, 3, 4, 5};
    for (int num : v) {
        std::cout << num << " ";
    }
    return 0;
}
```

5.1.2 Associative Containers

Associative containers store elements in sorted order and support fast searching. The main associative containers are `std::set`, `std::multiset`, `std::map`, and `std::multimap`.

- `std::set` is a container that stores unique elements in sorted order. Here, **key** and **value** are the same and thus used interchangeably.
- `std::multiset` is a container that stores multiple elements in sorted order.
- `std::map` is a container that stores key-value pairs in sorted order.
- `std::multimap` is a container that stores multiple key-value pairs in sorted order.

Example:

```cpp
#include <iostream>

int main() {
    std::map<std::string, int> m;
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;

    for (const auto& [key, value] : m) {
        std::cout << key << " => " << value << std::endl;
    }
    return 0;
}
```

They can be further divided in **ordered** and **unordered** associative containers. The former maintain elements in sorted order, while the latter do not. For the former, an ordering relation is required for the elements, which can be provided by a comparison function or by a comparison operator. Moreover, keys can be accessed read-only, but not modified. For the latter, a hashing function is required for the elements, which can be provided by a hash function or by a hash operator. Keys can be accessed and modified.

5.1.3 Container Adaptors

Container adaptors provide restricted access to elements by adapting existing containers. The primary container adaptors are `std::stack`, `std::queue`, and `std::priority_queue`.

- `std::stack` is a container that provides a LIFO (Last In, First Out) data structure.
- `std::queue` is a container that provides a FIFO (First In, First Out) data structure.
- `std::priority_queue` is a container that provides a priority queue data structure.

Example:

```cpp
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    q.push(1);
    q.push(2);
    q.push(3);

    while (!q.empty()) {
        std::cout << q.front() << " ";
        q.pop();
    }
    return 0;
}
```

5.1.4 Special Containers

Special containers provide specialized functionality. Examples include `std::byte`, `std::pair`, `std::tuple`, `std::variant`, `std::optional`, and `std::any`.

- `std::byte` is a container that stores byte values.
- `std::pair` is a container that stores a pair of elements.
- `std::tuple` is a container that stores a tuple of elements.
- `std::variant` is a container that stores a variant of elements.
- `std::optional` is a container that stores an optional value.
- `std::any` is a container that stores any type of value.

Example:

```cpp
#include <iostream>
#include <tuple>

int main() {
    std::tuple<int, float, std::string> t(1
    , 3.14, "Hello");
    std::cout << std::get<0>(t) << " ";
    std::cout << std::get<1>(t) << " ";

    return 0;
}
```

For further examples see here.

5.1.5 Iterators

Iterators are a generalization of **pointers** that allow a C++ program to work with different data structures (for example, containers and ranges (since C++20)) in a uniform manner. The iterator library provides definitions for iterators, as well as iterator traits, adaptors, and utility functions.

Definition:

An iterator is any object that allows iterating over a succession of elements, typically stored in a standard container. It can be dereferenced with the `*` operator, returning an element of the range, and incremented (moving to the next element) with the `++` operator.

Iterator category	Operations and storage requirement						
	write	read	increment		decrement	random access	contiguous storage
			without multiple passes	with multiple passes			
Iterator			Required				
OutputIterator	Required		Required				
InputIterator (mutable if supports write operation)		Required	Required				
ForwardIterator (also satisfies *InputIterator*)		Required	Required	Required			
BidirectionalIterator (also satisfies *ForwardIterator*)		Required	Required	Required	Required		
RandomAccessIterator (also satisfies *BidirectionalIterator*)		Required	Required	Required	Required	Required	
ContiguousIterator[1] (also satisfies *RandomAccessIterator*)		Required	Required	Required	Required	Required	Required

Figure 5.1: Iterators

Key Concepts of Iterators

- **Abstraction**: Iterators provide a way to access elements of a container without exposing its internal structure.
- **Uniform Interface**: They offer a consistent interface for traversing different types of containers (e.g., arrays, vectors, lists, etc.).
- **Categories**: Iterators are categorized based on their functionality. The main categories are:
  - **Input Iterators**: Can read elements in a sequence (forward-only, single-pass).
  - **Output Iterators**: Can write elements in a sequence (forward-only, single-pass).
  - **Forward Iterators**: Can read and write elements in a sequence (forward-only, multi-pass).
  - **Bidirectional Iterators**: Can move both forward and backward in a sequence.
  - **Random Access Iterators**: Can access any element in constant time (like pointers).

Common Iterator Operations

- `*iter`: Dereference the iterator to access the element it points to.
- `iter->member`: Access a member of the element the iterator points to.
- `++iter` / `iter++`: Move the iterator to the next element.

- `--iter` / `iter--`: Move the iterator to the previous element (for bidirectional and random access iterators).
- `iter1 == iter2` / `iter1 ≠ iter2`: Compare two iterators for equality.
- `iter + n` / `iter - n`: Move the iterator by `n` positions (for random access iterators).

Iterator Types in C++ Standard Library

- `begin()` and `end()`:
  - `begin()` returns an iterator to the first element of the container.
  - `end()` returns an iterator to one past the last element (used as a sentinel value).
- `const_iterator`: A read-only iterator that cannot modify the elements of the container.
- `reverse_iterator`: Iterates over the container in reverse order.

Example:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Using iterators to traverse the vector
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " "; // Dereference the iterator to
           access the element
    }
    std::cout << std::endl;

    // Using range-based for loop (internally uses iterators)
    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Iterator Categories in Practice

- **Random Access Iterators**: Supported by `std::vector`, `std::array`, and `std::deque`.
- **Bidirectional Iterators**: Supported by `std::list` and `std::set`.
- **Forward Iterators**: Supported by `std::forward_list`.
- **Input/Output Iterators**: Used in specific scenarios like reading from or writing to streams.

Custom Iterators

You can also define custom iterators for your own container classes by implementing the required operations (e.g., `++`, `*`, `==`, etc.).

5.1.6 size_type and size_t

They are used to represent the size of a container. `size_type` is a type defined by the container class, while `size_t` is a type defined by the C++ standard library. They are guaranteed to be unsigned integral types, but their sizes may vary depending on the platform.

Example:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};
    std::vector<int>::size_type vec_size = vec.size();
    std::size_t vec_size_t = vec.size();

    std::cout << "size_type: " << vec_size << std::endl;
    std::cout << "size_t: " << vec_size_t << std::endl;

    return 0;
}
```

5.2 Algorithms

The STL provides an extensive set of algorithms to operate on containers, or more precisely on **ranges**.

Warning:

C++20 has revised the concept of range and provides a new set of algorithms in the namespace `std::ranges`, with the same name as the old ones, but simpler to use and sometimes more powerful.

Why use STL Algorithms?

With standardized algorithms:

- You are more uniform with respect to different container types.
- The algorithm of the standard library may do certain optimizations if the contained elements have some characteristics.
- You have a parallel version for free…

5.2.1 Non-modifying Algorithms

Non-modifying algorithms do not change the contents of the container (the value in the range). They are used to perform operations like searching, counting, and comparing elements.

Example:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Find the first element greater than 3
    auto it = std::find_if(vec.begin(), vec.end(), [](int x) {
        return x > 3; });

    if (it != vec.end()) {
        std::cout << "First element greater than 3: " << *it << std
            ::endl;
    }

    // Check if all elements are positive
    bool all_positive = std::all_of(vec.begin(), vec.end(), [](int
        x) { return x > 0; });

    if (all_positive) {
        std::cout << "All elements are positive" << std::endl;
    }

    return 0;
}
```

5.2.2 Modifying Algorithms

Modifying algorithms change the contents of the container. They are used to perform operations like sorting, removing elements, and transforming elements.

Example:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {5, 2, 3, 4, 1};

    // Sort the elements in ascending order
    std::sort(vec.begin(), vec.end());

    // Remove the first element
    vec.erase(vec.begin());

    // Transform each element to its square
    std::transform(vec.begin(), vec.end(), vec.begin(), [](int x) {
        return x * x; });

    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

5.2.3 Inserters

Inserters are used to insert elements into a container. The main inserters are:

- `std::back_inserter`: Inserts elements at the end of a container.
- `std::front_inserter`: Inserts elements at the beginning of a container.
- `std::inserter`: Inserts elements at a specified position in a container.

Example:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec1 = {1, 2, 3};
    std::vector<int> vec2 = {4, 5, 6};

    // Insert elements from vec2 to vec1
    std::copy(vec2.begin(), vec2.end(), std::back_inserter(vec1));

    for (int val : vec1) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

5.2.4 Sorting Algorithms

Sorting algorithms are used to arrange elements in a specific order. They operate on a range to order it according to an ordering relation.

Example:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {5, 2, 3, 4, 1};

    // Sort the elements in ascending order
    std::sort(vec.begin(), vec.end());

    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

5.2.5 Min and Max

They are used to find the minimum and maximum elements in a range. The functions `std::min_element` and `std::max_element` return an iterator to the minimum and maximum elements, respectively.

Example:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> vec = {5, 2, 3, 4, 1};

    // Find the minimum and maximum elements
    auto min_it = std::min_element(vec.begin(), vec.end());
    auto max_it = std::max_element(vec.begin(), vec.end());

    std::cout << "Min element: " << *min_it << std::endl;
    std::cout << "Max element: " << *max_it << std::endl;

    return 0;
}
```

5.2.6 Numeric Algorithms

Numeric algorithms are used to perform numeric operations on a range of elements. The main numeric algorithms are:

- `std::accumulate`: Computes the sum of elements in a range.
- `std::inner_product`: Computes the inner product of two ranges.
- `std::partial_sum`: Computes the partial sum of elements in a range.
- `std::adjacent_difference`: Computes the differences between adjacent elements in a range.

Example:

```cpp
#include <iostream>
#include <vector>
#include <numeric>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Compute the sum of elements
    int sum = std::accumulate(vec.begin(), vec.end(), 0);

    // Compute the partial sum of elements
    std::vector<int> partial_sum(vec.size());
    std::partial_sum(vec.begin(), vec.end(), partial_sum.begin());

    for (int val : partial_sum) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

5.3 STL evolution

The STL has evolved over time, with new features and improvements introduced in each C++ standard. Here are some key changes in the STL from C++11 to C++20:

- **C++11**: Introduced move semantics, lambda expressions, and range-based for loops.
- **C++14**: Added generic lambdas, variable templates, and binary literals.
- **C++17**: Introduced parallel algorithms, `std::optional`, `std::variant`, and `std::any`.
- **C++20**: Added concepts, ranges, coroutines, and `std::span`.

For loop evolution

The range-based for loop was introduced in C++11 to simplify iteration over containers. It allows you to iterate over a range of elements without using iterators explicitly.

Example:

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // Using range-based for loop
    for (int val : vec) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

5.4 Smart Pointers

Definition: *RAII*

Resource Acquisition Is Initialization plays a significant role in C++. It essentially means that an object should be responsible for both the creation and destruction of the resources it owns.

Example: *RAII Compliant*

```
std::array<double, 10> p;  // var p creates and destroys the array
```

In C++, **smart pointers** are important tools to implement RAII.
Types of pointers:

- **Standard Pointers**: use them only to watch and operate on an object whose lfespan is independent of the one of the pointer.
- **Smart Pointers**: they control the lifespan of an object.
  - `std::unique_ptr`: manages a single object. The owned resource is destroyed when the pointer goes out of scope.
  - `std::shared_ptr`: manages a shared object. The owned resource is destroyed when the **last** shared pointer goes out of scope.
  - `std::weak_ptr`: observes a shared object without owning it. It is used to break circular references between `std::shared_ptr`s.

Example:

```
#include <iostream>

class MyClass {
public:
        set_polygon(std::unique_ptr<Polygon> p) {
            polygon = std::move(p);
        }
private:
        std::unique_ptr<Polygon> polygon;
};

std::unique_ptr<Polygon> create_polygon() {
    switch(t){
        case "Triangle":
            return std::make_unique<Triangle>();
        case "Rectangle":
            return std::make_unique<Rectangle>();
        default:
            return nullptr;
    }
}

MyClass obj;
obj.set_polygon(create_polygon("Triangle"));
```

5.4.1 std::unique_ptr

A `std::unique_ptr` serves as the unique owner of the object of type `T` it refers to. The object is destroyed automatically when the `std::unique_ptr` goes out of scope. It implements the `*` and `→` operators to access the object it owns, so it can be used as standard pointer, but can only be initialized through the constructor.
The default contructor produces a **null pointer**.

Warning: *Copying*

By definition, unique pointers cannot be **copied**, but their ownership can be transferred using the `std::move` function. The previous owner will be left with a null pointer.

5.4.2 std::shared_ptr

For instance you have several objects that refer to a resource (e.g., a matrix, a shape, ...) that is build dynamically (and maybe is a polymorphic object). You want to keep track of all the references in such a way that when (and only when) the last one gets destroyed the resource is also destroyed.
A `Shared Pointer` implements the semantics of *clean it up* when the resource is no longer needed.

Example:

```
#include <iostream>

class MyClass { ... };

class Preprocessor {
public:
    Preprocessor(std::shared_ptr<Data> &data, ...) : data(data) {}
private:
    std::shared_ptr<Data> data;
};

class NumericalSolver {
public:
    NumericalSolver(std::shared_ptr<Data> &data, ...) : data(data)
        {}
private:
    std::shared_ptr<Data> data;
};

int main() {
    std::shared_ptr<Data> data = std::make_shared<Data>();
    Preprocessor preprocessor(data, ...);
    preprocessor.preprocess();
    // shared\_data will still be used by other resources, hence it
        cannot be destroyed here.
    NumericalSolver solver(data, ...);
    solver.solve();
    return 0;
}
```

How a shared pointer works

The `std::shared_ptr` keeps track of the number of shared references to an object through reference counting. When the reference count reaches zero, the object is automatically deallocated, preventing memory leaks. It implements `*` and `→` operators to access the object it owns, so it can be used as a standard pointer. Moreover, it provides copy constructors and assignment operators.

5.4.3 std::weak_ptr

The `std::weak_ptr` is a smart pointer that holds a non-owning (weak) reference to an object managed by a `std::shared_ptr`. It is used to break circular references between shared pointers. It must also be converted to a `std::shared_ptr` to access the object it refers to.

Example:

```
#include <iostream>

std::shared_ptr<int> shared = std::make_shared<int>(42);
std::weak_ptr<int> weak = shared;  // Get pointer to data without
   taking ownership.
```

5.5 Move Semantics

Problem: how can I swap two objects in an efficient way?
Before C++11 one could have used a special method or a friend function.

Example:

```
#include <iostream>

void swap_with_move(Matrix& a, Matrix& b) {
    // swap rows and cols

    double* tmp = a.data; // save the pointer
    a.data = b.data // copy the pointer
    b.data = tmp; // copy the pointer
}
```

But this way it is not generalizable, since I cannot write a function template `swap_with_move<T>`, because I need to know how data is stored in `T` for each case.

Lvalues

Definition:

An **lvalue** (left value) is an expression that refers to a specific memory location and persists beyond a single expression.

- They have an **identifiable memory location**.
- They can be **modified** (unless they are `const`).

- They can be **bound to lvalue references** (`T&`).

Example:

```
int x = 10;  // 'x' is an lvalue
x = 20;      // Valid: lvalues can appear on the left side of an
   assignment

int& ref = x;  // Valid: lvalues can be bound to lvalue references
```

A `const` lvalue is still an lvalue but cannot be modified:

```
const int y = 5;
y = 10;  // Error: lvalue is read-only
```

Rvalues

Definition:

An **rvalue** (right value) is an expression that does **not persist beyond a single expression** and does not have a specific memory location that can be accessed.

- They are **temporary** and cannot be assigned to.
- They **cannot be bound to lvalue references** (`T&`), but they **can be bound to rvalue references** (`T&&`).

Example:

```
int x = 10;
int y = x + 5;  // 'x + 5' is an rvalue
```

C++11 introduced **rvalue references** to allow functions to take ownership of temporary values:

```
void foo(int&& a) {  // Accepts only rvalues
    std::cout << "Rvalue: " << a << std::endl;
}

foo(10);  // Valid: '10' is an rvalue
int x = 5;
foo(x);   // Error: 'x' is an lvalue
```

Lvalues vs. Rvalues in Function Overloading

Functions can be overloaded based on lvalue and rvalue references:

```
void process(int& x) {
    std::cout << "Lvalue reference" << std::endl;
}

void process(int&& x) {
```

```
    std::cout << "Rvalue reference" << std::endl;
}

int main() {
    int a = 10;
    process(a);   // Calls lvalue version
    process(20);  // Calls rvalue version
}
```

5.5.1 How Move Semantics is implemented

The following is the standard signature of a move operation for a class `Matrix`:

```
    Matrix(Matrix&&); // move constructor
    Matrix& operator=(Matrix&&); // move assignment operator
```

Example:

```
#include <iostream>

Matrix(Matrix&& rhs) : data(rhs.data), nr(rhs.nr), nc(rhs.nc) {
    rhs.data = nullptr;
    rhs.nr = 0;
    rhs.nc = 0;
}

Matrix& operator=(Matrix&& rhs) {
    delete[] this -> data; // release the resource
    data = rhs.data; // shallow copy
    // Fix rhs so it is a valid empty matrix
    rhs.data = nullptr;
    rhs.nr = 0;
    rhs.nc = 0;
}

int main() {
    Matrix foo();
    // ...
    Matrix a;
    a = foo(); // move assignment is called
    return 0;
}
```

Observation: *std::move*

The `std::move` function is used to convert an lvalue to an rvalue reference, allowing it to be moved instead of copied. It basically cast the value to an rvalue, making it available to be copied.

```
    template<class T>
    void swap(T& a, T& b) {
        T tmp = std::move(a);
```

```
        a = std::move(b);
        b = std::move(tmp);
    }
    // or simpler
    std::swap(a, b);
```

The solution is to force the move:

```
    class Foo{
    public:
        Foo(Matrix&& m) : my_m(std::move(m)) {}
        // ...
    private:
        Matrix my_m;
    }
```

Now, m is moved into my_m.
All standard containers support move semantic, and all standard algorithms are written so that if the contained type implements move semantics, the creation of unnecessary temporaries can be avoided. All containers also have a `swap()` method that performs swaps intelligently.

5.6 Exceptions

A **function** can be seen as a mapping from input data to output data. The conditions under which the input data is considered valid is called **precondition**, while the conditions under which the output data is considered valid is called **postcondition**. Failure to meet these conditions is considered a **failure** or **bug**.
An **invariant** is a condition that must always be true during the execution of a program. An object is considered to be in an **inconsistent state** if the invariants are not met.

Definition: *Exception*

An **exception** is an anomalous condition that disrupts the normal flow of a program's execution when left unhandled. It is not the result of incorrect coding but rather arises from challenging or unpredictable circumstances.

5.6.1 Run-time assertions

Example:

```
double calculate(double operand1, double operand2) {
    assert(operand2 != 0.0 && "Division by zero");
    return operand1 / operand2;
}
```

For improved efficiency, all assertions can be disabled by defining the `NDEBUG` macro:

```
    g++ -DNDEBUG -o program program.cpp
```

5.6.2 Compile-time assertions

Example:

```
template <typename T>
void process(T value) {
    static_assert(std::is_integral<T>::value, "T must be an
       integral type");
    // Process the value
}
```

If the condition is met, the error message is printed to the standard error and compilation will fail.

5.6.3 Exception handling in C++

The basic structure to handle exceptions is:

- Using the `throw` command to indicate that an exception has occurred. You can throw an object containing information about the exception.
- Employing the `try-catch` blocks to catch and handle exceptions. If an exception is not caught, it will propagate up the call stack and might lead to program termination.

Example:

```
int divide(int dividend, int divisor) {
    if (divisor == 0) {
        throw std::runtime_error("Division by zero");
    }
    return dividend / divisor;
}

try{
    const int result = divide(10, 0);
    std::cout << "Result: " << result << std::endl;
} catch (const std::runtime_error& e) {
    std::cerr << "Exception: " << e.what() << std::endl;
}
```

Standard exceptions examples are:

- `std::exception`: Base class for all standard exceptions.
- `std::runtime_error`: Exception used to report runtime errors.
- `std::logic_error`: Exception used to report errors in the program's logic.
- `std::invalid_argument`: Exception used to report invalid arguments.
- `std::out_of_range`: Exception used to report out-of-range errors.
- `std::bad_alloc`: Exception used to report memory allocation errors.
- `std::bad_cast`: Exception used to report casting errors.

Observation:

In situations where an algorithm's failure is one of its expected outcomes (e.g., the failure of convergence in an iterative method), returning a **status** rather than throwing an exception may be more suitable. Instead of terminating the program, a status variable is used to indicate the outcome, which can be checked by the caller. See also `std::terminate` , `std::abort` and `std::exit` .

Note, also, that **floating-point exceptions** do not result in program failure, instead they produce special numerical values like `NaN` or `Inf`, and the operations continue.

5.7 STL Utilities

5.7.1 I/O Streams

Input/Output (I/O) streams in C++ provide a convenient way to perform input and output operations, allowing you to work with various data sources and destinations, such as files, standard input/output, strings, and more.

The key components are:

- **iostream**: Provides the basic input/output operations. It is used for interacting with the standard input and output streams.
- **ifstream**: This class is used for reading data from files. You can open a file for input and read data from it.

```
std::ifstream file("input.txt");

if (file.is_open()) {
    std::string line;
    while (std::getline(file, line)) {
        std::cout << line << std::endl;
    }
    file.close();
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
```

- **ofstream**: This class is used for writing data to files. You can open a file for output and write data to it.

```
std::ofstream file("output.txt");

if (file.is_open()) {
    file << "Hello, World!" << std::endl;
    file.close();
} else {
    std::cerr << "Failed to open the file." << std::endl;
}
```

- **stringstream**: This class is used for reading and writing data to and from strings. It provides a way to work with strings as if they were streams.

```
        std::stringstream ss;
        ss << "Hello, World!";
        std::string str = ss.str();
        std::cout << str << std::endl;

        // Passing data from a string using std::stringstream.
        std::string data = "42 3.14 Hello";
        std::stringstream ss(data);
        int num;
        double d;
        std::string word;
        ss >> num >> d >> word;
```

5.7.2 Random Numbers

The capability of generating random numbers is essential not only for statistical purposes but also for internet communications. But an algorithm is deterministic. However, several techniques have been developed to generate pseudo-random numbers. They are not really random, but they show a low level of auto-correlation.

To generate them, you need the `<random>` header, and the chosen design is bsed on two types of objects:

- **Engines**: They are the source of randomness. They are used to generate random numbers.
- **Distributions**: They are used to transform the random numbers generated by the engine into a specific range.

Engines

Random number engines generate pseudo-random numbers using seed data as an entropy source. Several different classes of pseudo-random number generation algorithms are implemented as templates that can be customized.

For simplicity, the library provides predefined engines, such as `std::default_random_engine`, which balances efficiency and quality. There are also non-deterministic engines, like `std::random_device`, which generate non-deterministic random numbers based on hardware data.

Example:

```
std::default_random_engine engine;  // with default seed
std::default_random_engine engine(42);  // with seed 42
```

Distributions

Distributions are template classes that implement a call operator () to transform a random sequence into a specific distribution. You need to pass a random engine to the distribution to generate numbers according to the desired distribution. For example:

Example:

```
std::random_device rd;
std::default_random_engine engine(rd());
std::uniform_int_distribution<int> dist(1, 6);  // [1, 6]
int num = dist(engine);
```

5.7.3 Shuffling

In C++ you can shuffle a range of elements using the `std::shuffle` algorithm from the `<algorithm>` header. It rearranges the elements in the range `[first, last)` randomly using a random number generator.

Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::random_device rd;
std::mt19937 engine(rd());
std::shuffle(vec.begin(), vec.end(), engine);
```

5.7.4 Sampling

The `std::sample` algorithm from the `<algorithm>` header is used to sample elements from a range. It selects a random subset of elements from the input range and stores them in the output range.

Example:

```
std::vector<int> vec = {1, 2, 3, 4, 5};
std::vector<int> sample(3);
std::random_device rd;
std::mt19937 engine(rd());
std::sample(vec.begin(), vec.end(), sample.begin(), 3, engine);
```

5.7.5 Time Measuring

C++ provides three types of clock:

- `std::chrono::system_clock`: Represents the system-wide real time wall clock.
- `std::chrono::steady_clock`: Represents the monotonic clock that is not affected by system clock adjustments.
- `std::chrono::high_resolution_clock`: Represents the clock with the shortest tick period available on the system.

Example:

```
auto start = std::chrono::high_resolution_clock::now();
// Perform some operation
auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::
    milliseconds>(end - start);
```

5.7.6 Filesystem Utilities

The C++17 standard introduced the `<filesystem>` library, which provides facilities for performing operations on file systems and their components, such as paths, regular files, and directories.

Key Components

- `std::filesystem::path`: Represents a path in the filesystem.
- `std::filesystem::directory_entry`: Represents a directory entry.
- `std::filesystem::directory_iterator`: Iterates over the contents of a directory.
- `std::filesystem::recursive_directory_iterator`: Recursively iterates over the contents of a directory.

Common Operations

- `std::filesystem::exists`: Checks if a file or directory exists.
- `std::filesystem::create_directory`: Creates a new directory.
- `std::filesystem::remove`: Removes a file or directory.
- `std::filesystem::rename`: Renames a file or directory.
- `std::filesystem::copy`: Copies a file or directory.

Example:

```cpp
#include <iostream>
#include <filesystem>

int main() {
    std::filesystem::path p = "example.txt";

    // Check if the file exists
    if (std::filesystem::exists(p)) {
        std::cout << p << " exists." << std::endl;
    } else {
        std::cout << p << " does not exist." << std::endl;
    }

    // Create a new directory
    std::filesystem::create_directory("new_directory");

    // Remove a file
    std::filesystem::remove("example.txt");

    // Rename a file
    std::filesystem::rename("old_name.txt", "new_name.txt");

    // Copy a file
    std::filesystem::copy("source.txt", "destination.txt");

    return 0;
}
```

Iterating Over Directory Contents

You can use `std::filesystem::directory_iterator` to iterate over the contents of a directory.

Example:

```cpp
#include <iostream>
#include <filesystem>

int main() {
    std::filesystem::path dir = "some_directory";

    // Iterate over the contents of the directory
    for (const auto& entry : std::filesystem::directory_iterator(
        dir)) {
        std::cout << entry.path() << std::endl;
    }

    return 0;
}
```

Recursive Directory Iteration

You can use `std::filesystem::recursive_directory_iterator` to iterate over the contents of a directory recursively.

Example:

```cpp
#include <iostream>
#include <filesystem>

int main() {
    std::filesystem::path dir = "some_directory";

    // Recursively iterate over the contents of the directory
    for (const auto& entry : std::filesystem::
        recursive_directory_iterator(dir)) {
         std::cout << entry.path() << std::endl;
    }

    return 0;
}
```

6 Libraries

Definition:

A library is a collection of pre-written code or routines that can be reused by computer programs. These libraries typically contain functions, variables, classes, and procedures that perform common tasks, allowing developers to save time and effort by leveraging existing code rather than writing everything from scratch.

Why are useful?

- **Text Reusability**: Libraries provide a set of functionalities that can be used across multiple projects, reducing the need to write the same code over and over again.
- **Modularity**: Libraries promote modular programming by encapsulating specific functionalities into separate modules or components.
- **Abstraction**: Libraries abstract the underlying implementation details, allowing developers to use high-level interfaces without needing to understand the inner workings of the functions provided by the library.
- **Collaboration**: Communities of developers can share and collaborate on libraries, accelerating the development process. Many programming languages have centralized repositories or package managers to facilitate the distribution and installation of libraries.
- **Efficiency**: Libraries are often optimized and well-tested, providing efficient and reliable solutions for common programming tasks.

Libraries are composed by:

- **Header Files**: Header files contain declarations of functions, variables, and classes that are defined in the library. They provide the necessary information for the compiler to link the library with the program.
- **Source Files**: Source files contain the actual implementation of the functions, variables, and classes declared in the header files. They are compiled into object files that are linked together to create the final executable. They can be either static or shared.

Observation: ***Header-only libraries***

An exception are the libraries that contain only header files, which are known as header-only libraries.

Header files are only used in the development phase. In production, only **library files** are needed. Precompiled executables that just use shared libraries do not need header files to work. This is why certain software packages are divided into standard and development versions; only the latter contains the full set of header files.

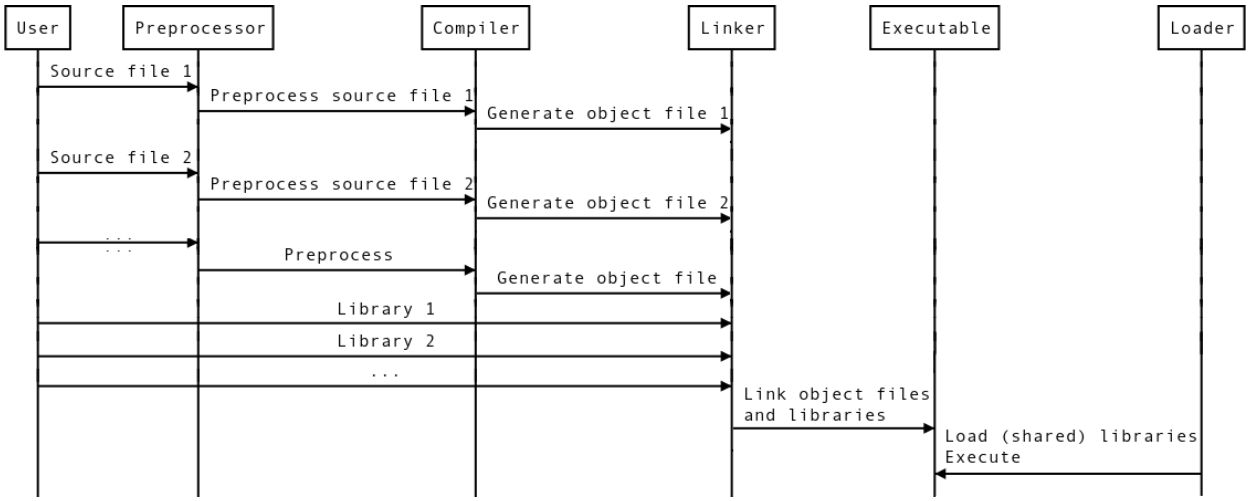

Figure 6.1: Build Process

Another distinction has to be made:

- **Static Libraries**: Static libraries are linked directly into the executable at compile time. Denoted with `.a` or `.lib` extension. When you build a program using a static library, a copy of the library's code is included in the final executable. This means that the resulting executable is independent of the original library file; it contains all the necessary code to run without relying on external library files.
- **Shared Libraries**: Shared libraries are loaded at runtime when the program is executed. Denoted with `.so` or `.dll` extension. When you build a program using a shared library, the library file is not included in the executable. Instead, the program references the shared library file, which is loaded into memory when the program is run. This allows multiple programs to share the same library file, reducing the overall memory footprint.

6.1 The build process

The preprocessing and compilation steps

```
1     g++ -Imylib/include -c main.cpp -o main.o
```

produce the main.o executable, it contains

```
1 $ nm -C main.o
2 0000000000000000 T main
3                  U my_fun
```

The `T` in the second column indicates that the function `main` is actually defined (resolved) by the library. While `myfun` is referenced but undefined. So, to produce a working executable, you have to specify to the linker another library or object file where it is defined.

Case 1: You can access *myfun*

- compile the object file implementing the function

```
1 g++ -c mylib.cpp
```

- link the application against that object file

```
g++ main.o mylib/mylib.o -o main
```

- now both `main` and `myfun` are resolved

```
$ nm -C main
    0000000000000000 T main
    0000000000000010 T my_fun
```

Case 2: The reality

In reality there are problems with this approach:

- Compilation takes time!
- Recompilation has to be done every time the library changes
- Actual implementation of the library is not available most of the times
- Dependency Hell

You can link in two ways:

```
    g++ main.o /path/to/mylib/libmylib.a -o main  # using library full
        path
```

```
    g++ main.o -L/path/to/mylib -lmylib -o main  # using -L and -l flags
```

Warning:

The order of the arguments is important! The linker processes the arguments from left to right, so the libraries should be specified after the object files that reference them.

6.2 Static Libraries

Static libraries are the oldest and most basic way of integrating third-party code. They are basically a collection of object files stored in a single archive. At the linking stage of the compilation processes, the symbols that are still unresolved are searched into the other object files indicated to the linker and in the indicated libraries, and eventually the corresponding code is inserted in the executable. Libraries result themselves from preprocessing and compiling their corresponding source codes.

```
    g++ -c mylib.cpp
    ar rcs libmylib.a mylib.o
```

Definition:

A static library is just an archive of source codes.

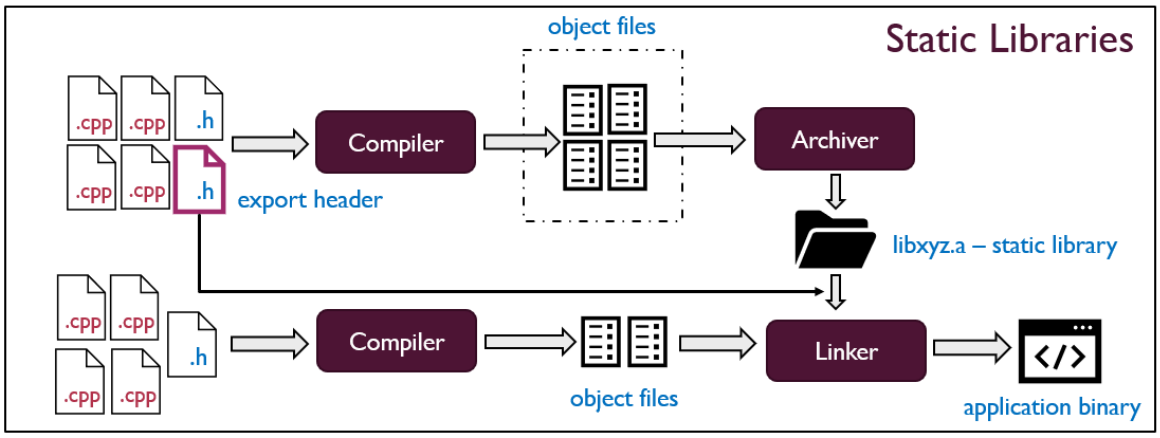

Figure 6.2: Static Library

Creating A Static Library

```
/my_library
├── include       # Header files
│   └── my_lib.hpp
├── src           # Source files
│   └── my_lib.cpp
├── build         # Object files and intermediate files
│   └── my_lib.o
└── lib           # Compiled library files
    └── libmy_lib.a   (static library) or libmy_lib.so (shared library)
```

Figure 6.3: Directory Structure

First create a folder (/my_library) with sub-folders /include (for header_files.hpp), /src (for source_files.cpp), /build (for compiled.o files) and /lib (for compiled_library_files.a).
Now insert declarations in header_files.hpp and definitions in source_files.cpp:

```
1    // chrislib.hpp
2    #pragma once
3
4    namespace chrislib {
5
6        void hello_world();  // Function declaration
7
8    }
```

```
1    // chrislib.cpp
2    #include "/path_to_library/include/chrislib.hpp"
3    #include <iostream>
```

```
4
5       namespace chrislib {
6
7           void hello_world() {
8               std::cout << "Hello from my private library!" << std::endl;
9           }
10
11      }
```

And then with terminal compile the source files:

```
1  g++ -c src/chrislib.cpp -o build/chrislib.o   ## compilation
2  ar rcs lib/libmy_lib.a build/my_lib.o   ## library files
```

Now you have the executables needed to use the library. It is time to create the project directory.

Using A Static Library

```
/my_project
├── include       # Header files (my_lib.hpp)
│   └── my_lib.hpp
├── lib           # Static or shared libraries (libmy_lib.a, libmy_lib.so)
│   └── libmy_lib.a
│   └── libmy_lib.so
├── src           # Source files (main.cpp, my_lib.cpp)
│   └── main.cpp
│   └── my_lib.cpp
├── build         # Object files
│   └── my_lib.o
└── Makefile      # Build automation file (optional)
```

Figure 6.4: Project Directory

```
1       cp -r ~/path_to_library/include ~/project   ## copying the entire
            include folder
2       cp -r ~/path_to_library/lib ~/project   ## copying the entire lib
            folder
```

Create now your main.cpp file and compile it.

```
1       // main.cpp
2       #include <iostream>
3       #include "/path_to_project/include/chrislib.hpp"
4
5       int main() {
6           std::cout << "Calling function from my private library!" << std
                ::endl;
7           chrislib::hello_world();  // Function from the static library
```

```
8           return 0;
9       }
```

```
1       g++ src/main.cpp -I./include -L./lib -lchrislib -o ./build/test_1
           ## -I for headers, -L for library files (executables)
```

Now just run the file (test_1):

```
1       christianfaccio@Christians-MacBook-Air build % ./test_1
2       Calling function from my private library!
3       Hello from my private library!
```

Key Flags:

- **-I** *(Include Directory)*: Specifies the directory to search for header files.
  `-I /path/to/include`
- **-L** *(Library Directory)*: Specifies the directory to search for static libraries.
  `-L /path/to/lib`
- **-l** *(Link Library)*: Links the static library (without the `lib` prefix and file extension).
  `-lchrislib`
- **-o** *(Output File)*: Specifies the output executable name.
  `-o my_program`
- **-std=c++17** *(C++ Standard)*: Specifies the C++ version to use (e.g., C++17).
  `-std=c++17`
- **-Wall** *(All Warnings)*: Enables all compiler warnings.
  `-Wall`
- **-g** *(Debug Information)*: Includes debugging information in the compiled binary.
  `-g`
- **-r** *(Relocatable Object File)*: Tells the linker to generate an object file that is relocatable (not yet fully linked).
  `-r`

6.3 Shared Libraries

Here:

- The **linker** ensures that symbols that are still unresolved are provided by the library.
- The corresponding code is not inserted, and the symbols remain unresolved.
- Instead, a reference to the library is stored in the executable for later use by the **loader**.

Warning:

Linker and loader are two different programs!

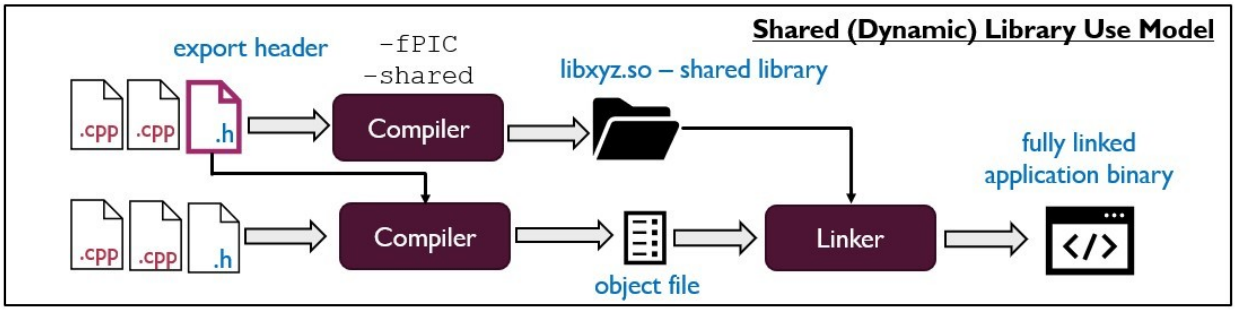

Figure 6.5: Shared Library

Creating A Shared Library

As before, create the header_files.hpp and source_files.cpp (I will use the same as before), putting them in a directory with the same structure as before.

```
1  g++ -fPIC -shared -Iinclude src/chrislib.cpp -o lib/libchrislib.so  ##
       you can even create the executable .o but it is an intermediary step
```

```
/Users/christianfaccio/chrislib/
    ├── include/                # Header files (e.g., chrislib.h)
    │   ├── chrislib.h
    ├── lib/                    # Compiled dynamic library
    │   ├── libchrislib.so       # The shared library
    ├── src/                    # Source files
    │   ├── chrislib.cpp
    ├── CMakeLists.txt          # Build configuration file
    └── ...                     # Other files if needed
```

Figure 6.6: Shared Library

Using A Shared Library

Go to the project directory, then

```
1  g++ -I../chrislib/include -L../chrislib/lib -lchrislib src/main.cpp -o
       build/my_program
```

```
1  export LD_LIBRARY_PATH=../chrislib/lib:$LD_LIBRARY_PATH
```

```
/Users/christianfaccio/project_test/
    ├── lib/                    # Folder to store the symlink or copy of libchri
    │   ├── libchrislib.so       # Symbolic link to /Users/christianfaccio/chris
    ├── src/                    # Source files of the project
    │   ├── main.cpp            # Entry point for your project
    ├── CMakeLists.txt          # Build configuration file
    ├── build/                  # Folder where the compiled program will be stor
    │   ├── my_program          # The compiled executable
    └── ...                     # Other files if needed
```

Figure 6.7: Shared Library

```
christianfaccio@Christians-MacBook-Air project_test % ./build/my_program
      ## run the program
Hello from chrislib!
```

Key Flags:

- `-fPIC`: Generate position-independent code (required for shared libraries).
- `-shared`: Create a shared library.

Linking and Compilation Flags

- **-I** *(Include Directory)*: Specifies the directory to search for header files.
  `-I /path/to/include`
- **-L** *(Library Directory)*: Specifies the directory to search for shared libraries.
  `-L /path/to/lib`
- **-l** *(Link Library)*: Links the shared library (without the `lib` prefix and file extension).
  `-lchrislib`
- **-o** *(Output File)*: Specifies the output executable name.
  `-o my_program`
- **-fPIC** *(Position Independent Code)*: Tells the compiler to generate position-independent code for shared libraries.
  `-fPIC`
- **-shared** *(Create Shared Library)*: Tells the compiler to create a shared library.
  `-shared`
- **-Wl,-rpath** *(Runtime Library Search Path)*: Specifies a runtime path for shared libraries.
  `-Wl,-rpath,/path/to/lib`
- **-std=c++17** *(C++ Standard)*: Specifies the C++ version to use (e.g., C++17).
  `-std=c++17`

- **-Wall** *(All Warnings)*: Enables all compiler warnings.

`-Wall`

- **-g** *(Debug Information)*: Includes debugging information in the compiled binary.

`-g`

6.4 Version Control

The version is an identifier typically represented by a sequence of numbers, indicating instances of a library with a common public interface and functionality. Naming scheme:

- **Link Name**: Used in the linking stage with the -lmylib option, of the form libmylib.so
- **soname (shared object name)**: Looked after by the loader, typically formed by the link name followed by the major version number, e.g., libmylib.so.3
- **Real Name**: The actual file storing the library with the full version number, e.g., libmylib.so.3.2.4

The ldd command is used (Linux/Mac) to display the shared libraries required by a program or a shared library. It shows the list of dynamic libraries (shared objects) that are linked to an executable or a shared library at runtime.

```
1 /Users/christianfaccio/project_test/lib/libchrislib.so:
```

```
1 build/libchrislib.so (compatibility version 0.0.0, current version
     0.0.0)
2 /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version
     1800.101.0)
3 /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version
      1351.0.0)
```

If there's a new release, placing the corresponding file in the /lib directory and resetting symbolic links, will make the program use the new release without recompiling (and this is what happens when, for example, you upgrade a package via apt or similar).

When working with different versions of libraries, dependency management becomes important to ensure that the correct versions of libraries are linked and used by your applications. This is especially crucial when there are multiple versions of libraries on your system, as using an incorrect version may lead to runtime errors or unexpected behavior.

- **Compatilbility version**: The version that a shared library guarantees compatibility with. This means that any program or library that links to this version will work as expected with the shared library
- **Current version**: The actual version of the library. It can evolve over time with bug fixes, features, or breaking changes

If you're using symbolic links (e.g., libchrislib.so pointing to a specific version like libchrislib.1.so), ls -l will show you where the symlink points.

```
1 ls -l /path/to/libchrislib.so
```

```
1 lrwxr-xr-x  1 christianfaccio  staff  52 Nov 14 23:32 /Users/
     christianfaccio/project_test/lib/libchrislib.so -> /Users/
     christianfaccio/chrislib/build/libchrislib.so
```

SONAME is a special name associated with shared libraries that helps the system identify which version of the library to load at runtime. It is used to manage compatibility between different versions of the same library. When a library is built and installed, the SONAME is embedded in the file as part of the dynamic linking process. This allows executables that rely on that library to load the correct version during runtime, based on the version specified in the SONAME.

For example, if you have a libchrislib.so.1 library (version 1) and you update it to libchrislib.so.2 (version 2), the SONAME might still reference libchrislib.so.1, ensuring older programs using version 1 of the library work as expected.

When creating a shared library, you typically specify the SONAME using the -Wl,-soname flag during compilation and linking. For example:

```
1 gcc -shared -o libchrislib.so.1.2.3 -Wl,-soname,libchrislib.so.1
    chrislib.o
```

This will ensure that the SONAME libchrislib.so.1 is embedded in the library file, and any application that links against libchrislib.so.1 will use this specific version or a compatible version.

Loading Phase

Linking and Loading phases are different. The loader has a different search strategy with respect to the linker. It looks in /lib , /usr/lib , and in all the directories contained in /etc/ld.conf or in files with the extension conf contained in the /etc/ld.conf.d/ directory.

7 Makefile and CMake

7.1 Large Project Structure

When working on a large project, it is essential to organize your repository in a way that makes it easy to manage and understand. A typical structure might look like this:

- **src/**: Contains all the source code files.
- **include/**: Contains all the header files.
- **lib/**: Contains external libraries.
- **build/**: Directory where the build system generates output files.
- **tests/**: Contains test code and test data.
- **docs/**: Contains documentation files.
- **CMakeLists.txt**: The main CMake configuration file.
- **README.md**: A file that provides an overview of the project.

This structure helps in maintaining a clean and organized codebase, making it easier for multiple developers to collaborate and for new developers to get up to speed quickly.

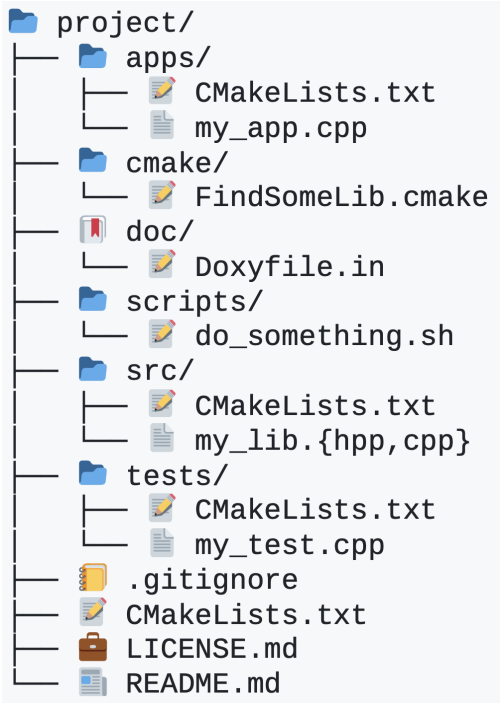

Figure 7.1: large Project Structure

7.2 Makefile

A `Makefile` can automate compilation and linking processes for your project. make efficiently determines the need to regenerate a target by checking its existence and the up- to-dateness of prerequisite files. This feature enables it to avoid unnecessary target regeneration. Make simplifies the installation of numerous libraries through a concise set of commands. A typical sequence for installing an open-source library involves using the following commands:

```
1      make  ##builds the library
2      make install   ##copies the library's headers, the libraries and the
           binaries to a user-specified folder
```

- **Target:** represents the desired output or action. It can be an executable, an object file, or a specific action like ”clean.”
- **Prerequisites:** are files or conditions that a target depends on. If any of the prerequisites have been modified more recently than the target, or if the target does not exist, the associated recipe is executed.
- **Recipes:** is a set of shell commands that are executed to build or update the target.

Considering a very simple case where we only have three files (math.hpp, math.cpp and main.cpp), we can write:

```
1  main.o: main.cpp math.hpp   ## for compilation
2      g++ std=c++17 -Wall main.cpp -c
3  math.o: math.cpp math.hpp   ## for compilation
4      g++ -std=c++17 -Wall math.cpp -c
5  main: main.o math.o   ## for linking
6      g++ main.o math.o -o main
7  clean:   ## for cleaning object files
8      rm -rf main *.o
```

A Makefile only redo the operations that contains files that have changed, and one can also call a single recipe instead of the entire file.
Use then variables for clarity and maintainability (used with $(...))

```
1  CXX=g++
2  CPPFLAGS=-I.   ## for the preprocessor of either C++ or C
3  CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror   ## specific for c++
       compiler
4
5  main.o: main.cpp math.hpp   ## for compilation
6      $(CXX) $(CPPFLAGS) $(CXXFLAGS) main.cpp -c
7  math.o: math.cpp math.hpp   ## for compilation
8      $(CXX) (CPPFLAGS) $(CXXFLAGS) math.cpp -c
9  main: main.o math.o   ## for linking
10     $(CXX) $(CXXFLAGS) main.o math.o -o main
11 clean:   ## for cleaning object files
12     rm -rf main *.o
13
14
15 ###### since the first two commands are the same we can summarize the
       command
16
```

```
CXX=g++
CPPFLAGS=-I.   ## for the preprocessor of either C++ or C
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror   ## specific for c++
    compiler

all=main  ## convention used to specify the final target of the project

%.o: %.cpp math.hpp
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) $< -c
main: $(OBJS)   ## for linking
    $(CXX) $(CXXFLAGS) $^$ -o $@
    ## $@ stands for the target file
    ## $^ expands to the list of all prerequisites (dependencies) of the
        current target
clean:   ## for cleaning object files
    rm -rf *.o

distcean:   ## convention to remove files generated during build process
    rm -rf main
```

Now we can introduce the dependencies:

```
    CXX=g++
CPPFLAGS=-I.   ## for the preprocessor of either C++ or C
CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror   ## specific for c++
    compiler

DEPS=math.hpp
SRCS=$(wildcard *.cpp)  ##List of all .cpp files
OBJS=$(SRCS:.cpp=.o)  ##Same, but replace .cpp with .o

all=main  ## convention used to specify the final target of the project

%.o: %.cpp $(DEPS)
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) $< -c
main: main.o math.o   ## for linking
    $(CXX) $(CXXFLAGS) $^$ -o $@
    ## $@ stands for the target file
    ## $^ expands to the list of all prerequisites (dependencies) of the
        current target
clean:   ## for cleaning object files
    rm -rf *.o

distcean:   ## convention to remove files generated during build process
    rm -rf main
```

To use this `Makefile`, simply type:

```
[language=bash]
make
```

To clean up:

```
[language=bash]
make clean
```

Example with dynamic library:

```
    ### Library Makefile
    CXX=g++
    CPPFLAGS=-I.
    CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror

    SRC=$(wildcard *.cpp)
    OBJ=$(SRC:.cpp=.o)
    OBJ_fPIC=$(SRC:.cpp=.fpic.o)
    DEPS=$(wildcard *.h)

    LIB_NAME_SHARED=libparser.so

    all: shared
    shared: $(LIB_NAME_SHARED)

    $(LIB_NAME_SHARED): $(OBJ_fPIC)
        g++ $(CXXFLAGS) -shared $^ -o $@

    %.fpic.o: %.cpp $(DEPS)
        $(CXX) -c -fPIC $(CPPFLAGS) $(CXXFLAGS) $^ -o $@

    %.o: %.cpp $(DEPS)
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@

    clean:
        rm -f *.o $(LIB_NAME_SHARED)
```

```
    ### Project Makefile
    CXX=g++
    CPPFLAGS=-Iparser/
    CXXFLAGS=-std=c++17 -Wall -Wpedantic -Werror

    LDFLAGS=-Wl,-rpath,parser/ -Lparser/ # For dynamic linking.
    LDLIBS=-lparser # Link the shared library libparser.so

    SRC=ex1.cpp
    OBJ=$(SRC:.cpp=.o)

    all: main

    main: $(OBJ)
        $(CXX) $(CXXFLAGS) $^ $(LDFLAGS) $(LDLIBS) -o $@

    %.o: %.cpp
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $< -o $@

    clean:
        rm -f *.o main
```

Then run in the terminal:

```
### In the library directory:
make

### In the project directory:
make

### Important thing is to insert the ex1.cpp in the project
    directory or update the directory in the Makefile
```

7.3 CMake

Introduction

CMake stands for "Cross-Platform Make." It is a **build-system generator**, meaning it creates the files (e.g., `Makefile`, Visual Studio project files) needed by your build system to compile and link your project. CMake abstracts away platform-specific build configurations, making it easier to maintain code that needs to run on multiple platforms.
It works the following way:

1. You write a `CMakeLists.txt` file that describes your project's configuration and structure.
2. You run CMake on the `CMakeLists.txt` file to generate the build system files (e.g. `Makefile` on Linux or .snl for Visual Studio).
3. You use the generated build system to compile and link your project.

CMakeLists.txt

Contains the configuration and structure of your project. It is a script that CMake uses to generate the build system files. It has the following structure:

```
cmake_minimum_required(VERSION 3.10)
project(MyProject)

add_executable(my_project_main.cpp)
```

Minimum Version

Here is the first line of every `CMakeLists.txt`, which is the required name of the file CMake looks for:

```
cmake_minimum_required(VERSION 3.10)
```

The version on CMake dictates the policies. Starting in CMake 3.12, this supports a range like `3.12...3.15`. This is useful when you want to use new features but still support older versions.

```
cmake_minimum_required(VERSION 3.12...3.15)
```

Setting a project

Every top-level CMake file will have this line:

```
project(MyProject VERSION 1.0
    DESCRIPTION "My Project"
    LANGUAGES CXX)
```

Strings are quoted, whitespace does not matte and the name of the prokect is the first argument. All the keywords are optional. The `version` sets a bunch of variables, like `MyProject_VERSION` and `PROJECT_VERSION`. The `LANGUAGES` keyword sets the languages that the project will use. This is useful for IDEs that support multiple languages.

Making an executable

```
add_executable(my_project_main my_project_main.cpp)
```

`my_project` is both the name of the executable file generate and the name of the CMake target created. The source file comes next and you can add more than one source file. CMake will only compile source file extensions. The headers will be ignored for most purposes; they are there only to be showed up in IDEs.

Making a library

```
add_library(my_library STATIC my_library.cpp)
```

`STATIC` is the type of library. It can be `SHARED` or `MODULE`. The source files are the same as for executables. Often you'll need to make a fictional target, i.e., one where nothing needs to be compiled, for example for header-only libraries. This is called an `INTERFACE library`, and the only difference is that it cannot be followed by filenames.

Targets

Now we've specified a target, we can set properties on it. CMake is all about targets and properties. An executable is a target, a library is a target. Your application is built as a collection of targets depending on each other.

```
target_include_directories(my_library PUBLIC include)
```

This sets the include directories for the target. The `PUBLIC` keyword means that the include directories will be propagated to any target that links to `my_library`. We can then chain targets:

```
add_library(my_library STATIC my_library.cpp)
target_link_libraries(my_project PUBLIC my_library)
```

This will link `my_project` to `my_library`. The `PUBLIC` keyword means that the link will be propagated to any target that links to `my_project`.
Targets can have include directories, linked libraries (or linked targets), compile options, compile definitions, compile features and more.

Variables

`Local variables` are used to store values that are used only in the current scope:

```
set(MY_VAR "some_file")
```

The names of the variables are case-sensitive and the values are strings. You access a variable by using `${}`. CMake has the concept of scope; you cna access the value of the variable after you set it as long as you are in the same scope. If you leave a function or a file in a sub directory, the variable will no longer be defined. You can set a variable in the scope immediately above your current one with `PARENT_SCOPE` at the end.
One can also set a list of values:

```
set(MY_LIST "value1" "value2" "value3")
```

which internally becomes a string with semicolons. You can access the values with `${MY_LIST}`. If you want to set a variable from the command line, CMake offers a variable cache. `Cache variables` are used to interact with the command line:

```
ì[language=C++]
set(MY_CACHE_VAR "VALUE" CACHE STRING "Description")

option(MY_OPTION "Set from command line" ON)
```

Then:

```
cmake /path/to/src/ \
-DMY_CACHE_VAR="some_value" \
-DMY_OPTION=OFF
```

`Environment variables` are used to interact with the environment:

```
# Read
message(STATUS $ENV{MY_ENV_VAR})

# Write
set(ENV{MY_ENV_VAR} "some_value")
```

But it is not recommended to use environment variables in CMake.

Properties

The other way to set properties is to use the `set_property` command:

```
set_property(TARGET my_library PROPERTY CXX_STANDARD 17)
```

This is like a variable, but it is attached to a target. The `PROPERTY` keyword is optional. The `CXX_STANDARD` is a property that sets the C++ standard for the target.

8 Optimization, Debugging, Profiling and Testing

8.1 Optimization

Definition:

Optimization **Code Optimization** is the process of enhancing a program's preformance, efficiency and resource utilization without changing its functionality. It involves improving execution speed, reducing memory usage and enhancing overall system responsiveness.

Techniques for optimization:

- **Algorithmic optimization**: changing the algorithm used to solve the problem.
- **Code optimization**: changing the code to make it more efficient.
- **Compiler optimization**: using compiler flags to optimize the code.
- **Hardware optimization**: using hardware features to optimize the code.
- **Profiling**: analyzing the code to identify bottlenecks and optimize them.

Loop unrolling

Loop unrolling is a technique used to optimize loops by reducing the overhead of loop control. It involves replicating the loop body multiple times to reduce the number of iterations. This can improve performance by reducing the number of loop control instructions and improving instruction-level parallelism.

```
1       for (int i = 0; i < n; ++i) {
2           for (int k = 0; k < 3; ++k) {
3               a[k] += b[k] * c[i];
4           }
5       }
6       // becomes
7       for (int i = 0; i < n; i += 3) {
8           a[0] += b[0] * c[i];
9           a[1] += b[1] * c[i];
10          a[2] += b[2] * c[i];
11      }
```

Prefetch also constant values inside the loop for further optimization:

```
1       for (int i = 0; i < n; i += 3) {
2           const int c0 = c[i];
3           a[0] += b[0] * c0;
4           a[1] += b[1] * c0;
5           a[2] += b[2] * c0;
```

```
    }
```

Avoid `if` inside nested loops

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (a[i][j] > 0) {
                b[i][j] = a[i][j];
            }
        }
    }
    // becomes
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            b[i][j] = std::max(a[i][j], 0);
        }
    }
```

Sum of vectors

```
    double sum1(double *data, const size_t &size) {
        double sum = 0;
        for (size_t i = 0; i < size; ++i) {
            sum += data[i];
        }
        return sum;
    }
    // vs
    double sum2(double *data, const size_t &size) {
        double sum{0}, sum1{0}, sum2{0}, sum3{0};
        size_t j;
        for (j = 0; j < size - 3; j += 4) {
            sum += data[j];
            sum1 += data[j + 1];
            sum2 += data[j + 2];
            sum3 += data[j + 3];
        }
        for (; j < size; ++j) {
            sum += data[j];
        }
        return sum + sum1 + sum2 + sum3;
    }
```

Here, `sum2` is 10 times faster than `sum1`. This is because of the loop unrolling technique.

Cache friendliness

Efficiency often depends on how variables are accessed in memory. Access variables contiguously for cache pre-fetching effectiveness. For example, if `mat` us a dynamic matrix organized row-wise:

```
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            sum += mat[i][j];
```

```
        }
    }
    // becomes
    for (int j = 0; j < m; ++j) {
        for (int i = 0; i < n; ++i) {
            sum += mat[i][j];
        }
    }
```

8.2 Debugging

Static Analysis analyzes the code without executing it. It can detect potential bugs, security vulnerabilities, and code smells. It can also provide insights into code complexity, maintainability, and performance. Static Analysis tools analyze source code by inspecting it for potential issues or vulnerabilities. An example is `cling`, an interactive C++ interpreter.

Debugging, instead, is the process of identifying and fixing bugs in a program. It involves tracing the execution of the program, inspecting variables, and analyzing the program's behavior to identify the root cause of the bug. Debuggers are software tools that enable developers to inspect, analyze and troubleshoot code during the development process. An example is `gdb`, the GNU Debugger. They differ in:

- **Timing**: Static analysis is done before the code is executed, while debugging is done during or after execution.
- **Focus**: Static analysis focuses on potential issues in the code, while debugging focuses on specific bugs that are causing problems.
- **Use cases**: Static analysis is used to prevent bugs and improve code quality, while debugging is used to fix bugs that have already occurred.
- **Automation**: Static analysis can be automated and integrated into the development process, while debugging is typically done manually.
- **Complementarity**: Static analysis and debugging are complementary techniques that can be used together to improve code quality and reliability.

8.3 Profiling

Definition:

Profiling is the process of analyzing a program's performance to identify bottlenecks and optimize its execution. It involves measuring various metrics such as CPU usage, memory usage, and execution time to identify areas of improvement. Profiling tools provide insights into how a program is performing and help developers optimize its performance.

A **profiler** in software development is a tool or set of tools designed to analyze the runtime behavior of a program. It provides detailed information about resource utilization, execution times, and function calls during the program's execution. Its main objectives are:

- **Performance analysis**, to know how much time a program spends in differnt functions
- **Resource usage**
- **Function call tracing**, to understand the flow of execution

`gprof` is a popular profiler for C and C++ programs. It generates a call graph showing the time spent in each function and the number of calls to each function. It can help identify bottlenecks and

optimize the code.

8.4 Testing

Verification ensures the correct implementation of a program, it tests individual components separately. **Validation**, instead, confirms the desired behavior, assessing if the code produces the intended outcome.
Exist different types of testing:

- **Unit testing**: Testing individual components or units of code in isolation.
- **Integration testing**: Testing the interaction between different components or units of code.
- **Regression testing**: Testing to ensure that new code changes do not introduce new bugs.

In C++, unit testing often uses frameworks like `Catch2` or `Google Test`. These frameworks provide tools for writing and running tests, as well as reporting results.

Example: *Using gtest*

```
#include <gtest/gtest.h>

int add(int a, int b) {
    return a + b;
}

TEST(AddTest, PositiveNumbers) {
    EXPECT_EQ(add(1, 2), 3);
}

TEST(AddTest, NegativeNumbers) {
    EXPECT_EQ(add(-1, -2), -3);
}

int main(int argc, char **argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Test-driven development (TDD)

TDD is a software development approach where tests are written before the actuaò code. The cycle is writing a test, implementing the code to pass the test, and refactoring the code. The advantages include better code quality, improved design, and faster development.

Continuous integration (CI)

CI is a software development practice where code changes are automatically tested and integrated into the main codebase. It helps identify bugs early, ensure code quality, and improve collaboration among developers. CI tools like `Jenkins` or `Travis CI` automate the testing and integration process.

Code coverage

Code coverage is a metric used in software testing to measure the extent to which source code is executed during the testing process. It provides insights into which parts of the codebase have been

exercised by the test suite and which parts remain untested. The key concepts are:

- **Lines of code**: The number of lines of code executed by the test suite. Since it is expressed in percentage, it is better to have it as close to 100% as possible.
- **Branches of code**: The number of branches of code executed by the test suite. It is important to test all possible branches to ensure code correctness.

9 Introduction to Python

9.1 Introduction

Python is a high-level, interpreted programming language. It is a general-purpose language that is designed to be easy to read and write. Python is a popular language for web development, scientific computing, and data analysis. It is also widely used in education and research. Why Python?

- Python is easy to learn and use.
- Python is a versatile language that can be used for a wide range of applications.
- Python has a large and active community of developers who contribute to the language and its libraries.
- Python has a rich ecosystem of libraries and tools that make it easy to work with data, build web applications, and more.
- Python is open source and free to use.
- Python is cross-platform, meaning it can run on Windows, macOS, and Linux.

Python Installation

To work wiht Python, you need to set up a development environment. Basic steps are:

- Download and install Python from the official website: https://www.python.org/downloads/
- Install an Integrated Development Environment (IDE) for Python. Some popular IDEs are PyCharm, Visual Studio Code, and Jupyter Notebook.
- Install the required libraries using the package manager pip.
- (Optional) Install a virtual environment to manage dependencies for different projects.

9.2 Built-in Data Types

Python typing is **dynamic**, meaning you do not have to declare the type of a variable when you create it. Python will automatically infer the type based on the value assigned to the variable. Python typing is also **strong**, such that you cannot merge different types of data together.

```
1 # Integer
2 x = 10
3 # String
4 name = "Alice"
5
6 'Alice' + x # TypeError: cannot concatenate 'str' and 'int' objects
```

Type Name	Type Category	Description	Example
int	Numeric	Integer numbers	`x = 10`
float	Numeric	Floating-point numbers	`y = 3.14`
str	Sequence	Textual data	`name = "Alice"`
list	Sequence	Ordered collection of items	`numbers = [1, 2, 3]`
tuple	Sequence	Immutable ordered collection of items	`point = (1, 2)`
dict	Mapping	Collection of key-value pairs	`person = {"name": "Alice", "age": 30}`
set	Set	Unordered collection of unique items	`unique_numbers = {1, 2, 3}`
bool	Boolean	True or False values	`is_valid = True`

Table 9.1: Python Built-in Data Types

9.2.1 Numeric, Boolean, Strings

- **Numeric Types** Exist three numeric types in Python: `int`, `float`, and `complex`. We can determine the type of an object with the `type()` function. Usual Arithmetic Operations are available.
- **Boolean Type** The boolean type is used to represent truth values. It has two possible values: `True` and `False`. Usual comparison operators are available. There are also the *boolean operators*, which evaluate to `True` or `False`. Python also has **bitwise operations** like AND (`&`), OR (`|`), XOR (`^`), NOT (`~`) and SHIFT (`<<`, `>>`).
- **Strings** They represent sequences of characters. Are created using single (`'`) or double (`"`) quotes. Strings are immutable, meaning that once they are created, they cannot be changed. Strings can be concatenated using the `+` operator. Strings can be formatted using the `format()` method or f-strings.

Example:

```
name = "Alice"

# Concatenation
greeting = "Hello, " + name

# Formatting
formatted_message = f"Hello, {name}"
```

9.2.2 Lists and Tuples

- **Lists** Lists are mutable, ordered collections of items. Elements can be added, removed or modified.

Example:

```
numbers = [1, 2, 3]

# Add an element
numbers.append(4)

# Remove an element
numbers.remove(2)

# Accessing an element
second_number = numbers[1]

# Slicing
first_two_numbers = numbers[:2]
```

- **Tuples** Tuples are immutable, ordered collections of items. Once created, they cannot be changed.

Example:

```
point = (1, 2)

# Unpacking
x, y = point

# Repetition
point = (0, 0) * 3

# Concatenation
points = (1, 2) + (3, 4)

# Slicing
first_point = points[:2]

# Accessing elements
x = points[0]
```

9.2.3 Dictionaries and Sets

- **Sets** Sets store an unordered list of unique items. Being unordered, sets do not record element position or oder of insertion and so do not support indexing.

Example:

```python
unique_numbers = {1, 2, 3}

# Add an element
unique_numbers.add(4)

# Remove an element
unique_numbers.remove(2)

{1, 2, 3} == {1, 2, 3} # True
[1, 2, 3] == [1, 2, 3] # False
```

- **Dictionaries** Dictionaries store a collection of key-value pairs. Keys are unique within a dictionary, while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Example:

```python
person = {"name": "Alice", "age": 30}

# Accessing a value
name = person["name"]

# Adding a key-value pair
person["city"] = "New York"

# Remove a key-value pair
del person["age"]
```

9.2.4 Casting

Python allows you to convert one data type to another. This process is known as **casting**. To cast a variable to a different type, you can use the built-in functions `int()`, `float()`, `str()`, `list()`, `tuple()`, `dict()`, and `set()`.

Example:

```python
# Casting to int
x = int(3.14)

# Casting to float
y = float(10)

# Casting to float
z = float("Hello") # ValueError: could not convert string to float:
    'Hello'
```

9.3 Control Structures

Conditional Statements

Conditional statements allow us to write programs where only certain blocks of code are executed depending on the state of the program.

Example:

```
x = 10

if x > 0:
    print("Positive")
elif x < 0:
    print("Negative")
else:
    print("Zero")
```

We can also write simple `if` statements *inline*, meaning in a single line, in this way:

```
x = 10
message = "Positive" if x > 0 else "Negative"
```

For Loops

For loops allow us to iterate over a sequence of items, such as a list, tuple, or string.

Example:

```
numbers = [1, 2, 3]

for number in numbers:
    print(number)
```

Range

The `range()` function generates a sequence of numbers. It can take up to three arguments: `start`, `stop`, and `step`.

Example:

```
for i in range(5):
    print(i)

for i in range(1, 5):
    print(i)

for i in range(1, 5, 2):
    print(i)
```

zip

The `zip()` function takes two or more sequences and pairs them element-wise. It returns a list of tuples.

Example:

```
names = ["Alice", "Bob", "Charlie"]
ages = [30, 25, 35]

for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

enumerate

The `enumerate()` function takes a sequence and returns an iterator that pairs each element with its index.

```
names = ["Alice", "Bob", "Charlie"]

for i, name in enumerate(names):
    print(f"{i}: {name}")
```

While Loops

While loops allow us to execute a block of code as long as a condition is true.

Example:

```
x = 0

while x < 5:
    print(x)
    x += 1
```

Break and Continue

The `break` statement is used to exit a loop prematurely. The `continue` statement is used to skip the rest of the code in a loop and move to the next iteration.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)

for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

9.4 Comprehensions

Comprehensions allow us to build lists/tuples/sets/dictionaries in one convenient, compact line of code.

```
words = ["apple", "banana", "cherry"]
letters = [word[0] for word in words] # List comprehension
```

```
# Multiple Comprehensions
[(i,j) for i in range(3) for j in range(4)]
```

```
numbers = [1, 2, 3, 4, 5]
squares = {number: number ** 2 for number in numbers} # Dictionary
    comprehension
```

9.5 Exceptions

When something goes wrong, we do not want our code to crash, but to **fail gracefully**. In Python we can handle exceptions using the `try` and `except` blocks.

Example:

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print("Division successful")
finally:
    print("End of program")
```

We can also write code that raises exceptions on purpose using the `raise` statement.

```
x = -1

if x < 0:
    raise ValueError("x must be positive")
```

This is useful when your function is complicated and would fail in a complicated way, with an error message. You can make the cause of the error much clearer to the user of the function.

9.6 Functions

Definition: *Function*

A **function** is a reusable piece of code that can accept input parameters, also known as *arguments*.

They begin with the `def` keyword, then the function name, arguments in parentheses and then a colon. The output is specified with the `return` keyword.

```
1 def greet(name):
2     return f"Hello, {name}"
```

When you create a variable inside a function, it is **local**, which means that it only exist inside the function.

Warning:

In Python, input arguments are passed by **reference**, meaning that if you modify the argument inside the function (and it is **mutable**), the change will be reflected outside the function.

Type Hints

Type hinting is a way to specify the type of a variable, function argument, or return value. It is not enforced by Python, but it can help you catch errors early and make your code easier to understand. It is just another level of documentation.

```
1 def greet(name: str) -> str:
2     return f"Hello, {name}"
```

Multiple return values

Python functions can return multiple values. This is done by separating the values with a comma.

```
1 def min_max(numbers):
2     return min(numbers), max(numbers)
```

Unpacking

In Python, the asterisk (`*`) is used to unpack a sequence into individual elements, i.e., it allows you to extract the elements from an iterable or the key-value pairs from a dictionary.

```
1     def add_numbers(a, b, c):
2         return a + b + c
3
4         numbers = [1, 2, 3]
5         result = add_numbers(*numbers)
6         print(results) # 6
```

```python
    # Unpacking in iterables
    first = [1, 2, 3]
    second = [4, 5, 6]
    combined = [*first, *second]
    print(combined) # [1, 2, 3, 4, 5, 6]

    # Unpacking in dictionary merging
    first = {"a": 1, "b": 2}
    second = {"c": 3, "d": 4}
    combined = {**first, **second}
    print(combined) # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

    # Unpacking in function definitions
    def example_function(a, b, *args, **kwargs):
        print(a)
        print(b)
        print(args)
        print(kwargs)
```

In Python, `*args` and `**kwargs` are used in function definitions to allow a variable number of arguments:

- `*args` is used to pass a variable number of non-keyword arguments to a function.

```python
def add_numbers(*args):
    return sum(args)
```

- `**kwargs` is used to pass a variable number of keyword arguments to a function.

```python
def print_person(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")
```

Lambda Functions

Lambda functions are small, anonymous functions that can have any number of arguments, but only one expression. They are defined using the `lambda` keyword.

```python
add = lambda x, y: x + y
result = add(1, 2)
```

Lambda functions are often used as arguments to higher-order functions, such as `map()`, `filter()`, and `sorted()`.

9.7 Docstrings

The **Docstrings** are used to document Python code. They are written between triple quotes and are used to describe what a function does, what arguments it takes, and what it returns.

```python
def greet(name):
    """
    This function greets the user by name.
```

```
4
5       Args:
6           name (str): The name of the user.
7
8       Returns:
9           str: A greeting message.
10      """
11      return f"Hello, {name}"
```

10 OOP in Python

10.1 Introduction

Classes in Python works in a similar way to classes in C++ seen before. However, Python classes are more flexible and powerful. In this chapter, we will see how to define classes in Python and how to use them.

```
class MyClass:
    def __init__(self, name):
        self.name = name

    def say_hello(self):
        print(f"Hello, {self.name}!")
```

Observation: `self`

Class methods have only one specific difference from ordinary functions: they must have an extra first name that has to be added to the beginning of the parameter list, but you do not give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object itself, and by convention, it is given the name `self`.

Also, the `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any initialization (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name. We do not explicitly call the `__init__` method. It gets called when we create a new instance of the class.

Attributes can be instance-specific or shared among all instances. The first ones are defined outside the `__init__` method, while the second ones are defined inside the class but outside any method.

```
class Dog:
    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance
```

Warning:

Changing class attributes affect all instances!

Besides regular methods, classes can have static methods and class methods. Static methods are methods that are not bound to an instance, while class methods are bound to the class itself.

- **Static methods** are defined using the `@staticmethod` decorator.

```python
class MyClass:
    @staticmethod
    def static_method():
        print("This is a static method")
```

- **Class methods** are defined using the `@classmethod` decorator.

```python
class MyClass:
    @classmethod
    def class_method(cls):
        print("This is a class method")
```

The `cls` parameter is the class itself.

Magic Methods

Magic methods are special methods that have double underscores at the beginning and at the end of their names. They are also known as dunder methods. They allow us to emulate built-in behavior within Python and can be extremely useful when we want to implement operator overloading.

```python
class MyClass:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return f"My name is {self.name}"
```

Observation:

Notes for C++ programmers:

- The `self` in Python is equivalent to the `this` pointer in C++.
- All class members (including the data members) are public and all the methods are virtual in Python.

10.2 Decorators

Decorators in Python offer a powerful way to enhance the functionality of functions or methods. They act as wrappers, allowing you to extend or modify the behavior of the original function. Decorators can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` is the same as calling

```python
    from_csv = classmethod(from_csv)
```

Definition:

Decorators are essentially functions that take another function as input, enhance its capabilities, and return a modified version of the original function.

```python
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_hello():
    print("Hello!")

say_hello = my_decorator(say_hello)
```

Or, with the `@` symbol:

```python
@my_decorator
def say_hello():
    print("Hello!")
```

This decorator, when applied to a function, surrounds the function call with additional actions. It can be applied to classes as well:

```python
def add_method(cls):
    def new_method(self):
        print("This is a new method")
    cls.new_method = new_method
    return cls

@add_method
class MyClass:
    def existing_method(self):
        print("This is an existing method")

obj = MyClass()
obj.new_method() # Prints "This is a new method"
```

Observation:

Python comes with built-in decorators like `@staticmethod` and `@classmethod`, which are implemented in C for efficiency.

Other than logging and timing/profiling, decorators can be used to add validation checks for input parameters and output cleanup to functions or methods, or to implement caching mechanisms, where the result of a function is stored for a specific set of inputs, and subsequent calls with the same inputs can return the cached result.

10.3 Inheritance and Polymorphism

Inheritance in Python enables classes to inherit methods and attributes from other classes. To use inheritance, we specify the base class names in a tuple following the class name in the class definition. Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of an instance in the subclass. This is very important to remember.

Warning:

Python does not automatically call the constructor of the base class, you have to call it explicitly!

In contrast, if we have not defined an `__init__` method in the subclass, the base class `__init__` method will be called automatically.

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        raise NotImplementedError("Subclass must implement abstract
            method")

class Dog(Animal):
    def __init__(self, name):
        super().__init__(name)

    def speak(self):
        return f"{self.name} says Woof!"
```

Getters

Since attributes can change, one can use the `@property` decorator to define a getter method. This method is called when the attribute is accessed.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def diameter(self):
        return 2 * self.radius

    @property
    def area(self):
        return 3.14159 * self.radius ** 2
```

Setters

Similarly, one can use the `@<attribute>.setter` decorator to define a setter method. This method is called when the attribute is assigned a new value.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def diameter(self):
        return 2 * self.radius

```

```
    @diameter.setter
    def diameter(self, diameter):
        self.radius = diameter / 2
```

This way we can ensure that the radius is always half the diameter.

Deleters

Finally, one can use the `@<attribute>.deleter` decorator to define a deleter method. This method is called when the attribute is deleted.

```
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def diameter(self):
        return 2 * self.radius

    @diameter.setter
    def diameter(self, diameter):
        self.radius = diameter / 2

    @diameter.deleter
    def diameter(self):
        del self.radius
```

10.4 Modules and Packages

In Python, the ability to reuse code is facilitated by modules. A module is a file with a `.py` extension that contains functions and variables. There are various methods to write modules, including using languages like C to create compiled modules. When importing a module, to enhance import performance, Python creates byte-compiled files `(__pycache__/filename.pyc)`. These files, platform-independent and located in the same directory as the corresponding `.py` files, speed up subsequent imports by storing preprocessed code.

You can import modules in your program to leverage their functionality. For instance, consider the `sys` module in the Python standard library. Below is an example:

```
import sys

print("Command line arguments: ", sys.argv)
```

You can selectively import variables from a module using the `from ... import ...` statement. However, it's generally advised to use the `import` statement to avoid potential name clashes and enhance readability.

```
    from math import pi
    print("The value of pi is", pi)
```

```
<some folder in sys.path>/
└── datascience/
    ├── __init__.py
    ├── preprocessing/
    │   ├── __init__.py
    │   ├── cleaning.py
    │   └── scaling.py
    └── analysis/
        ├── __init__.py
        ├── statistics.py
        └── visualization.py
```

Figure 10.1: Package Structure

Observation:

Every module has a `__name__` attribute that defines the name of the module. When the interpreter reads a source file, it sets the `__name__` variable to `"__main__"` if the module is being run as the main program. Otherwise, it sets the variable to the module's name.

You can create your own modules by writing a Python file and importing it in another file. To import a module, you can use the `import` statement followed by the module name. You can also import specific functions or variables from a module using the `from ... import ...` statement. Moreover, the `dif()` function lists all symbols defined in an object. For a module, it includes functions, classes, and variables. It can also be used without arguments to list names in the current module.

10.5 Packages

Packages are folders of modules with a special `__init__.py` file, indicating that the folder contains Python modules. They provide a hierarchical organization for modules.

Observation:

The `__init__.py` file in a Python package serves multiple purposes. It's executed when the package or module is imported, and it can contain initialization code, set package-level variables, or define what should be accessible when the package is imported using `from package import *`.

11 Integration with Python

11.1 Introduction

C++ and Python are powerful in their own right, but they excel in different areas. C++ is renowned for its performance and control over system resources, making it ideal for CPU-intensive tasks and systems programming. Python, on the other hand, is celebrated for its simplicity, readability, and vast ecosystem of libraries, especially in data science, machine learning, and web development.

- In research areas like machine learning, scientific computing, and data analysis, the need for processing speed and efficient resource management is critical.
- The industry often requires solutions that are both efficient and rapidly developed.

By integrating C++ with Python, you can create applications that harness the raw power of C++ and the versatility and ease-of-use of Python. Python, despite its popularity in these fields, often falls short in terms of performance. Knowledge of how to integrate C++ and Python equips with a highly valuable skill set.

Several libraries are available for this, each with its own set of advantages and drawbacks. We will use `pybind11`, a lightweight header-only library that exposes C++ types in Python and viceversa.

11.2 pybind11

11.2.1 Overview

pybind11 is a lightweight, header-only library that connects C++ types with Python. This tool is crucial for creating Python bindings of existing C++ code. Its design and functionality are similar to the Boost.Python library but with a focus on simplicity and minimalism. pybind11 stands out for its ability to avoid the complexities associated with Boost by leveraging C++11 features.

To install it on your system, you can use the following command:

- **pip**: `pip install pybind11`
- **conda**: `conda install -c conda-forge pybind11`
- **brew**: `brew install pybind11`

You can also include it as a submodule in your project:

```
1 git submodule add -b stable https://github.com/pybind/pybind11 extern/
      pybind11
2 git submodule update --init
```

This method assumes dependency placement in extern/. Remember that some servers might require the .git externsion. After setup, include extern/pybind11/include in your project, or employ pybind11's integration tools.

11.2.2 Basics

All pybind11 code is written in C++. The following lines must always be included in your code:

```cpp
#include <pybind11/pybind11.h>
namespace py = pybind11;
```

The first line includes the pybind11 library, while the second line creates an alias for the pybind11 namespace. This alias is used to simplify the code and make it more readable. In practice, implementation and binding code will generally be located in separate files.
The `PYBIND11_MODULE` macro is used to create a Python module. The first argument is the module name, while the second argument is the module's scope. The module name is the name of the Python module that will be created, while the scope is the C++ namespace that contains the functions to be exposed and is the main interface for creating bindings. Example:

```cpp
#include <pybind11/pybind11.h>
namespace py = pybind11;

int add(int i, int j) {
    return i + j;
}
PYBIND11_MODULE(example, m) {
    m.def("add", &add, "A function that adds two numbers");
}
```

Here we define a simple function that adds two numbers and bind it to a Python module named example. The function add is exposed to Python using the m.def() function. The first argument is the function name in Python, the second argument is the C++ function, and the third argument is the function's docstring.

Observation: *pybind11*

Notice how little code was needed to expose our function to Python: all details regarding the function's parameters and return value were automatically inferred using template metaprogramming. This overall approach and the used syntax are borrowed from Boost.Python, though the underlying implementation is very different.

Being it a header-only library, pybind11 does not require any additional linking or compilation steps. You can compile the code as you would any other C++ code. The resulting shared library can be imported into Python using the import statement. Compile the example in Linux with the following command:

```
g++ -O3 -Wall -shared -std=c++11 -fPIC
\$(python3 -m pybind11 --includes)
example.cpp -o example\$(python3-config --extension-suffix)
```

If you included pybind11 as a submodule, you can use the following command:

```
g++ -O3 -Wall -shared -std=c++11 -fPIC
-Iextern/pybind11/include
\$(python3-config --incldudes) Iextern/pybind11/include
example.cpp -o example\$(python3-config --extension-suffix)
```

This assumes that pybind11 has been installed with `pip` or `conda`, otherwise you can manually specify `-I <path-to-pybind11>/include` together with the Python includes path

`python3-config --includes`.

Warning: *On macOS*

the build command is almost the same but it also requires passing the `-undefined dynamic_lookup` flag so as to ignore missing symbols when building the module.

Building the C++ code will produce a binary module file that can be imported in Python with the `import` statement. The module name is the name of the shared library file without the extension. In this case, the module name is example. The shared library file is named example.so on Linux and example.dylib on macOS.

```
1 import example
2 print(example.add(1, 2)) #output: 3
```

With a simple modification, you can inform Pythn about the names of the arguments:

```
1 m.def("add", &add, "A function that adds two numbers",
2 py::arg("i"), py::arg("j"));
```

You can now call the function using keyword arguments:

```
1 import example
2 print(example.add(i=1, j=2)) #output: 3
```

Observation: *Documentation*

The docstring is automatically extracted from the C++ function and displayed in Python. This feature is useful for documenting the function's purpose and parameters. The docstring can be accessed in Python using the `__doc__` attribute.
help(example)
…
FUNCTIONS
add(…)
Signature: add(i: int, j: int) -> int
A function that adds two numbers

There is a shorthand notation

```
1 using namespace py::literals;
2 m.def("add", add, "Docstring", "i"_a, "j"_a=1);
```

The _a suffix is a user-defined literal that creates a py::arg object. This object is used to specify the argument's name and type. The shorthand notation is more concise and easier to read than the previous method. The second argument is optional and specifies the `default` value of the argument. If the argument is not provided, the default value is used.
`py::cast` is used to convert between Python and C++ types. The following example demonstrates how to convert a Python list to a C++ vector:

```cpp
std::vector<int> list_to_vector(py::list l) {
    std::vector<int> v;
    for (auto item : l) {
        v.push_back(py::cast<int>(item));
    }
    return v;
}
PYBIND11_MODULE(example, m) {
    m.def("list_to_vector", &list_to_vector, "Convert a Python list to a
        C++ vector");
}
```

To export variables, use the `attr` function. Built-in types and general objects are automatically converted when assigned as attributed, and can be explicitly converted using `py::cast`.

```cpp
int value = 42;
m.attr("value") = value;
```

```python
import example
print(example.value) #output: 42
```

11.2.3 Binding OO code

pybind11 supports object-oriented programming, allowing you to bind classes, methods, and attributes. The following example demonstrates how to bind a simple class:

```cpp
Pet(const std::string &name, int age) : name(name), age(age) {}
void set_name(const std::string &name_) { name = name_; }
void set_age(int age_) { age = age_; }
std::string get_name() const { return name; }
int get_age() const { return age; }
```

```cpp
PYBIND11_MODULE(example, m) {
    py::class_<Pet>(m, "Pet")
        .def(py::init<const std::string &, int>())
        .def("set_name", &Pet::set_name)
        .def("set_age", &Pet::set_age)
        .def("get_name", &Pet::get_name)
        .def("get_age", &Pet::get_age);
}
```

The `py::class_` function is used to bind a C++ class to Python. The first argument is the module, the second argument is the class name, and the third argument is the class type. The `def` function is used to bind class methods to Python. The first argument is the method name in Python, and the second argument is the C++ method.

```python
import example
pet = example.Pet("Tom", 5)
print(pet.get_name()) #output: Tom
print(pet.get_age()) #output: 5
pet.set_name("Jerry")
```

```
pet.set_age(3)
print(pet.get_name()) #output: Jerry
print(pet.get_age()) #output: 3
```

In the case above, the `print(pet)` statement will output the memory address of the object. To change this behavior, you can define a `__str__` method in the C++ class:

```
std::string __str__() const {
    return name + " is " + std::to_string(age) + " years old";
}
```

```
ì
.def("__str__", &Pet::__str__);
```

```
import example
pet = example.Pet("Tom", 5)
print(pet) #output: Tom is 5 years old
```

You can also expose the name field with the `def_readwrite` function:

```
.def_readwrite("name", &Pet::name)
```

```
import example
pet = example.Pet("Tom", 5)
print(pet.name) #output: Tom
pet.name = "Jerry"
print(pet.name) #output: Jerry
```

Dynamic attributes can be added to the class using the `def_property` function:

```
.def_property("description", &Pet::__str__, &Pet::set_name)
```

```
import example
pet = example.Pet("Tom", 5)
print(pet.description) #output: Tom is 5 years old
pet.description = "Jerry"
print(pet.get_name()) #output: Jerry
```

11.2.4 Inheritance and Polymorphism

pybind11 supports inheritance and polymorphism, allowing you to bind base classes and derived classes. The following example demonstrates how to bind a base class and a derived class:

```
class Animal {
public:
    virtual std::string speak() const {
        return "I am an animal";
    }
};
```

```
class Dog : public Animal {
public:
    std::string speak() const override {
        return "I am a dog";
    }
};
```

There are two ways to bind the derived class. The first method is to bind the base class and derived class separately, specifying the base class as an extra template parameter of the `class_`:

```
PYBIND11_MODULE(example, m) {
    py::class_<Animal>(m, "Animal")
        .def("speak", &Animal::speak);
    py::class_<Dog, Animal>(m, "Dog")
        .def(py::init<>());
        .def("speak", &Dog::speak);
}
```

```
import example
animal = example.Animal()
dog = example.Dog()
print(animal.speak()) #output: I am an animal
print(dog.speak()) #output: I am a dog
```

the second method is to assign a name to the previously bound `Pet class_` object and reference it when binding the Dog class:

```
py::class_<Pet> pet(m, "Pet");
pet.def(\dots);
py::class_<Dog>(m, "Dog", pet)
    .def(py::init<>());
    .def("speak", &Dog::speak);
```

```
import example
animal = example.Pet()
dog = example.Dog()
print(animal.speak()) #output: I am an animal
print(dog.speak()) #output: I am a dog
```