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 1, 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.

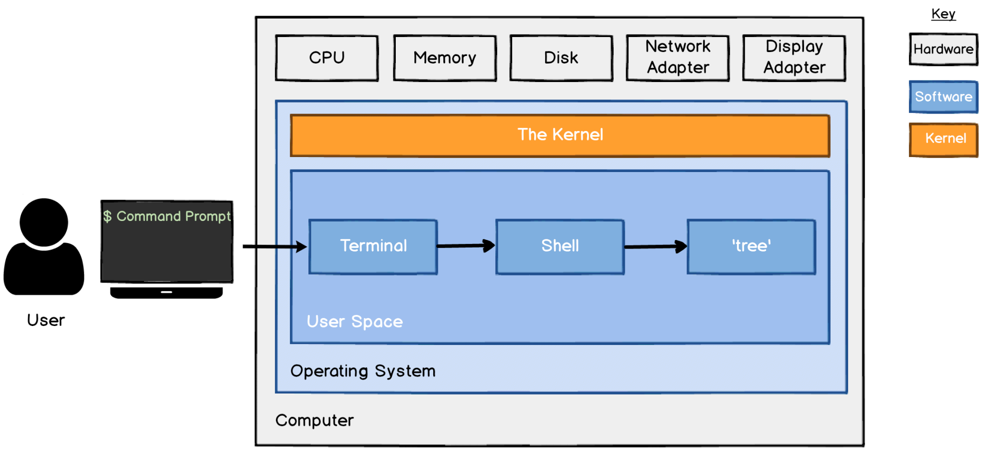

Figure 1.1: The Shell

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: *Sum Function*

An example function that prints the sum of two numbers:

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

1.2.6 Additional Shell Commands

More Commands

- `cat` : Stands for *Concatenate*. Reads and outputs the contents of files. It can read multiple files and concatenate their content.
- `wc` : Short for *Word Count*. Provides statistics like newline count, word count, and byte count for a list of files.
- `grep` : Stands for *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.
    Example: `sudo apt update && sudo apt upgrade`.
  - `||` : 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

- `tr` : Stands for *Translate*. Performs character transformations.
  - Example: `echo "abc" | tr [a-z] [A-Z]` converts lowercase to uppercase.
  - Example: `echo "123abc" | tr -d [:digit:]` removes digits.
- `sed` : A *stream editor* for text processing.
  - Example: `echo 'UNIX is great' | sed 's/UNIX/Linux/'` replaces ’UNIX’ with ’Linux’.
  - Example: `echo "1\n2\n3" | sed "2d"` deletes the second line.
- `cut` : Extracts specific sections from lines of text.
  - Example: `cut -b 1-3 file.txt` extracts the first three bytes of each line.
  - Example: `echo "1,2,3" | cut -d "," -f 1` retrieves the first column.
- `find` : Searches for files based on conditions.
  Example: `find . -type d -name "*lib*"` searches for directories containing ”lib”.
- `locate` : Faster alternative to `find`, relying on a pre-built database. Update the database with `updatedb`.
  Example: `locate -i foo` finds items containing ”foo”, ignoring case.

Quoting in Shell

Quoting affects how strings and variables are interpreted:

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

Example: *Quoting*

```
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:

Output conversion

```
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.

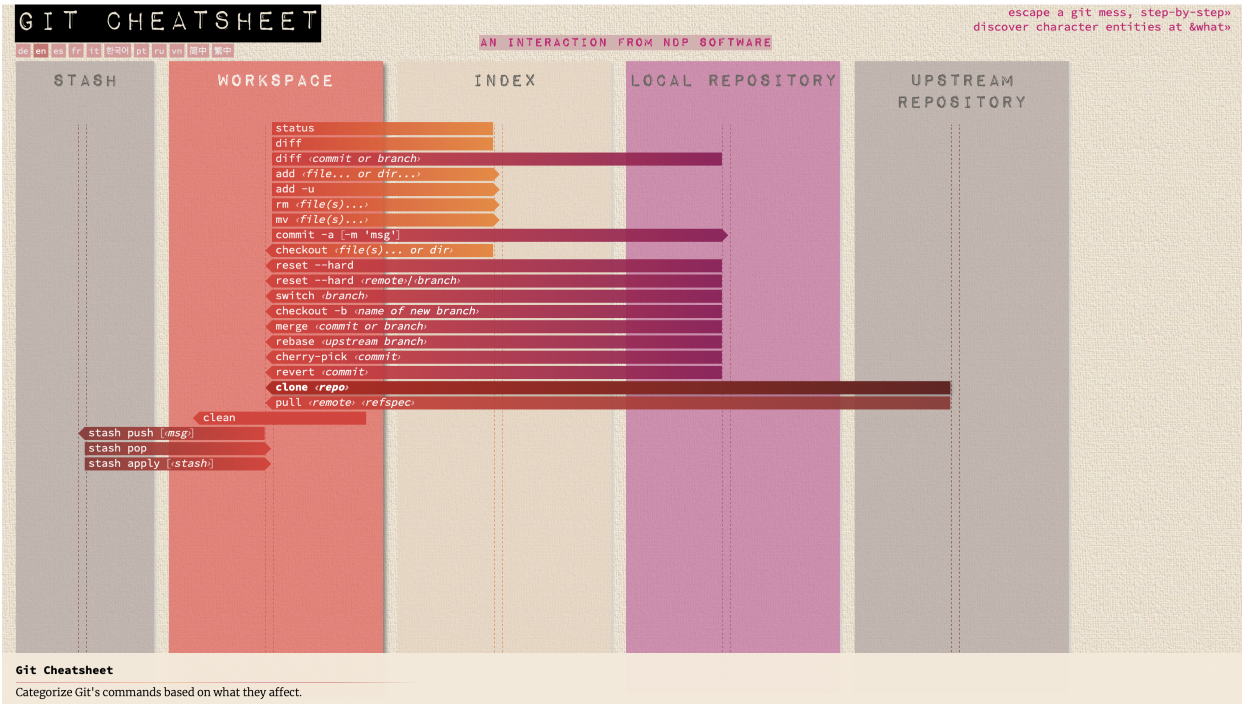

Figure 1.2: Git Cheatsheet

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 latest commit of the branch you are switching to. When you merge branches, Git combines the

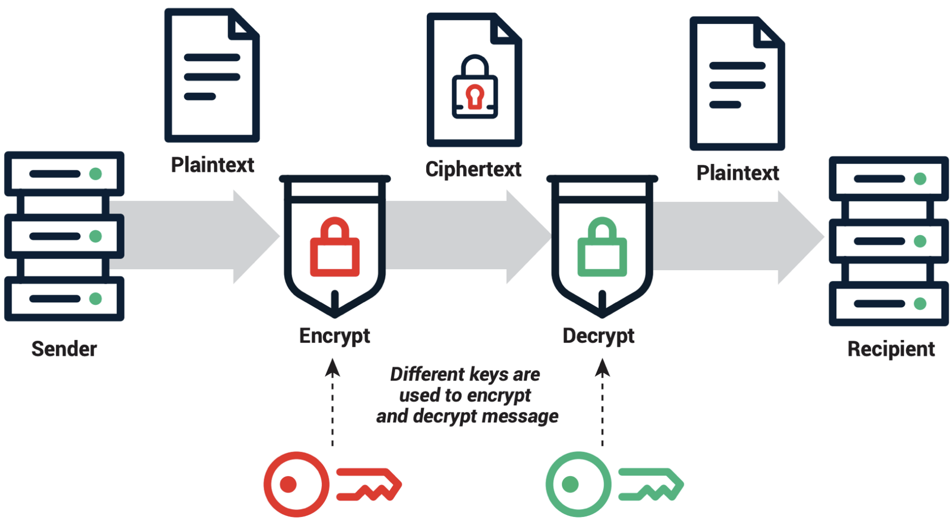

Figure 1.3: SSH

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.

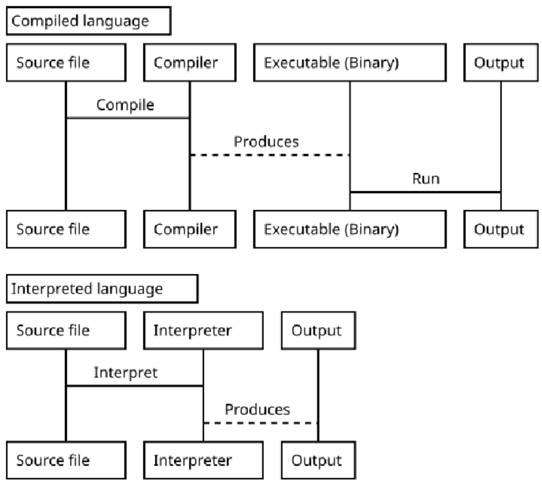

Figure 2.1: Compiled vs. Interpreted Languages

2.2.2 The Build Process

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

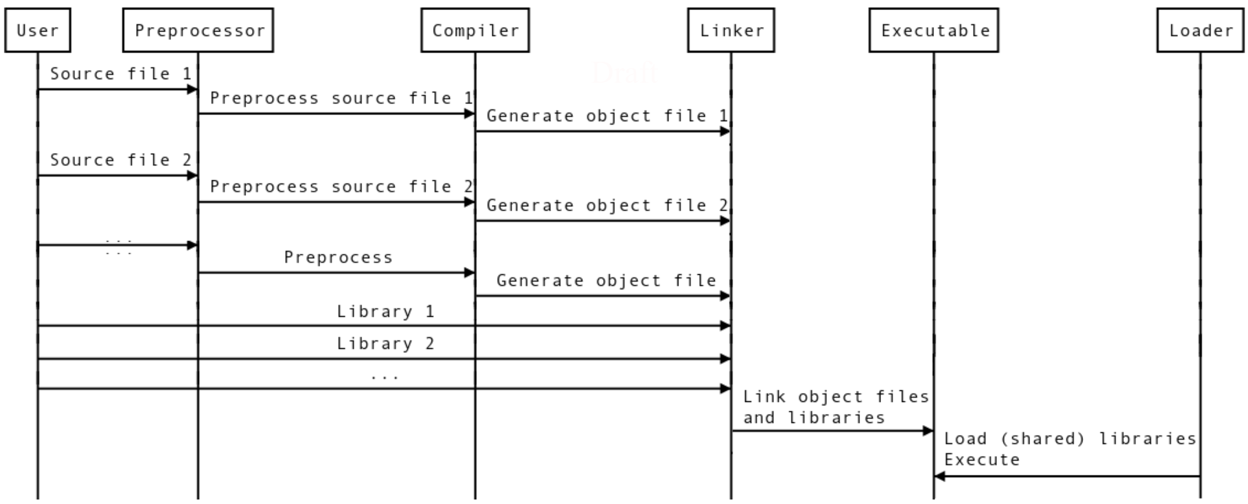

Figure 2.2: The Build Process

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.

Compiling a C++ File

```
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.

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.
```

2.4 Fundamental Types

C++ provides various built-in 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++

Integer Numbers

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

Example:

Integer Types

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

Floating-Point Numbers

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

Example:

Integer Types

```
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.

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.

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.
}
```

2.4.1 Representation of `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**. For example:
  - On a 32-bit system: `0x00000000`
  - On a 64-bit system: `0x0000000000000000`

- Example in C/C++:

Example: *Representation of NULL*

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

2.4.2 Representation of `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`:
  - **Quiet NaN (qNaN)**: Used for undefined or unrepresentable results. It has a leading `1` in the significand.
  - **Signaling NaN (sNaN)**: Used to trigger exceptions in certain operations. It has a leading `0` in the significand.
- Example in C/C++:

Example: *Representation of NaN*

```
#include <cmath>
#include <cstdio>

int main() {
    double result = 0.0 / 0.0; // Results in NaN
    if (isnan(result)) {
        printf("Result is NaN\n");
    }
    return 0;
}
```

2.4.3 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 (e.g., `0x00000000`)	Exponent filled with 1s and non-zero significand
Context	Memory addresses, pointers	Floating-point arithmetic

Characters and Strings

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

Example: ***Working with Characters and Strings***

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

Example: ***Boolean***

```cpp
bool is_happy = true;
bool is_sad = false;

if (is_happy) {
    std::cout << "You are happy!" << std::endl;
}
```

```cpp
if (-1.5) // true
if (0)    // false
```

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

Example: ***Initialization and Type Aliases***

Variable Initialization:

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

Type Aliases:

```cpp
        using integer = int;     // Alias for int.
        integer count = 100;

        typedef float distance; // Alias for float.
        distance meters = 250.5;
```

2.4.4 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.

Example: *The auto Keyword*

```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:

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

Explicit conversion:

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

2.5 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.

2.5.1 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***

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

2.5.2 Heap Memory

The **heap** is used for **dynamic memory 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 because it involves more 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***

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

2.5.3 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

2.5.4 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.

2.5.5 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***

```cpp
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.
```

2.5.6 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***

```cpp
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.

2.6 Condition Statements

2.6.1 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.

Example: ***If-Else Statement***

```cpp
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;
}
```

2.6.2 Switch Statements

The `switch` statement is used to select one of many code blocks to be executed. It evaluates an expression and compares it with case values. If a match is found, the corresponding block is executed.

Example: *Switch Statement*

```
int day = 3;
switch (day) {
    case 1:
        std::cout << "Monday" << std::endl;
        break;
    case 2:
        std::cout << "Tuesday" << std::endl;
        break;
    default:
        std::cout << "Invalid day" << std::endl;
}
```

2.6.3 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.

Example: *For Loop*

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

2.6.4 While Loop

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

Example: *While Loop*

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

2.7 Functions and operators

2.7.1 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.

Example: ***Function Definition***

```cpp
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.

Example: ***Function Parameters***

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

Warning: ***const correctedness***

const correctedness is a valuable practice for writing safe and maintainable code. It ensures that functions do not modify the input parameters unless explicitly intended. Use `const` for parameters that should not be modified, it indicates read-only data and functions.

- **Prevents unintended modifications**: helps avoid accidental data modifications, enhancing code safety.
- **Self-documenting code**: makes code more self-documenting by indicating the intent of data usage.
- **Compiler optimizations**: allows the compiler to perform certain optimizations, as it knows that const data won't change.

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

2.7.2 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.

Example: ***Increment and Decrement Operators***

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

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.

Example: ***Function Overloading***

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

2.8 User-defined types

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

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

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

```cpp
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.

```cpp
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).


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

2.9 Declarations and Definitions

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

Example: *Declaration and Definition*

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

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

2.10 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*

```cpp
// module.h (Header file)
#ifndef MODULE_H
#define MODULE_H

void foo();

#endif

// module.cpp (Source file)
#include "module.h"

void foo() {
    std::cout << "Hello, world!" << std::endl;
}
```

2.10.1 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***

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

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

2.11 The Build toolchain in practice

2.11.1 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
```

2.11.2 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.

2.11.3 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.

2.11.4 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
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.

3.1 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`.

3.1.1 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;
}
```

3.1.2 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.

3.1.3 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;
}
```

3.1.4 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.

3.1.5 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: without multiple passes	increment: with multiple passes	decrement	random access	contiguous storage
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 3.1: 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.).

3.1.6 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;
}
```

3.2 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…

3.2.1 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;
}
```

3.2.2 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;
}
```


3.2.3 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;
}
```

3.2.4 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;
}
```

3.2.5 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;
}
```

3.2.6 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;
}
```

3.3 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;
}
```

4 CMake

4.1 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.

4.2 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)
```

4.2.1 Minimum Version

Here is the first line of every `CMakeLists.txt`, which is the required name of the file CMake looks for:

CMakeLists.txt

```
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.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12...3.15)
```

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.2.5 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.

4.2.6 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:

```
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.

4.2.7 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.