UniTs - University of Trieste

Faculty of Data Science and Artificial Intelligence

Department of mathematics informatics and geosciences

Advanced Programming

Lecturer:
Prof. Pasquale Claudio Africa

Author:
Christian Faccio

January 21, 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.

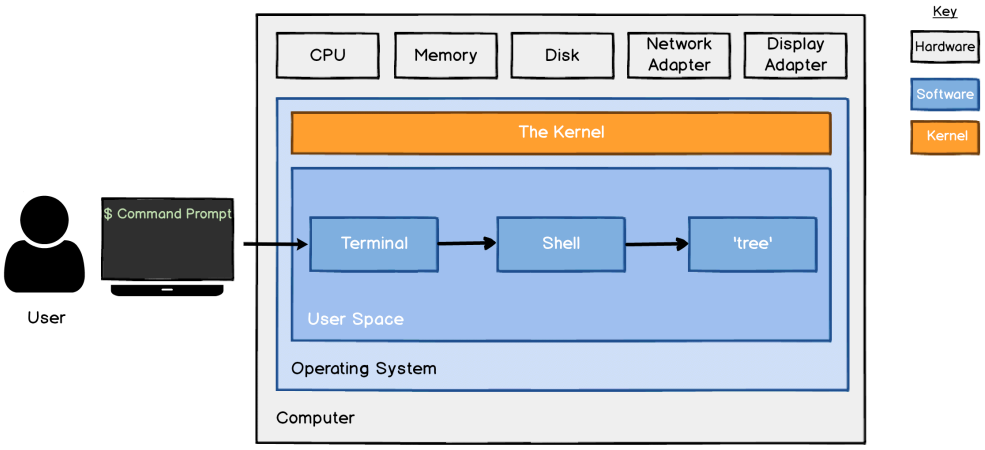

Figure 1.1: The Shell

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:

Assigning a Value to a Variable

```
A=1
```

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

Displaying a Variable's Value

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

Exporting a 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.

Bash Shebang

```
#!/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 Syntax

```
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:

```
function 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).

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.

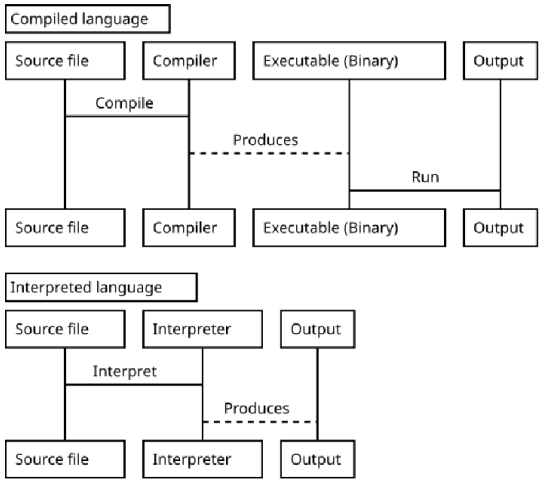

Figure 2.1: Compiled vs. Interpreted Languages

2.2.2 The Build Process

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

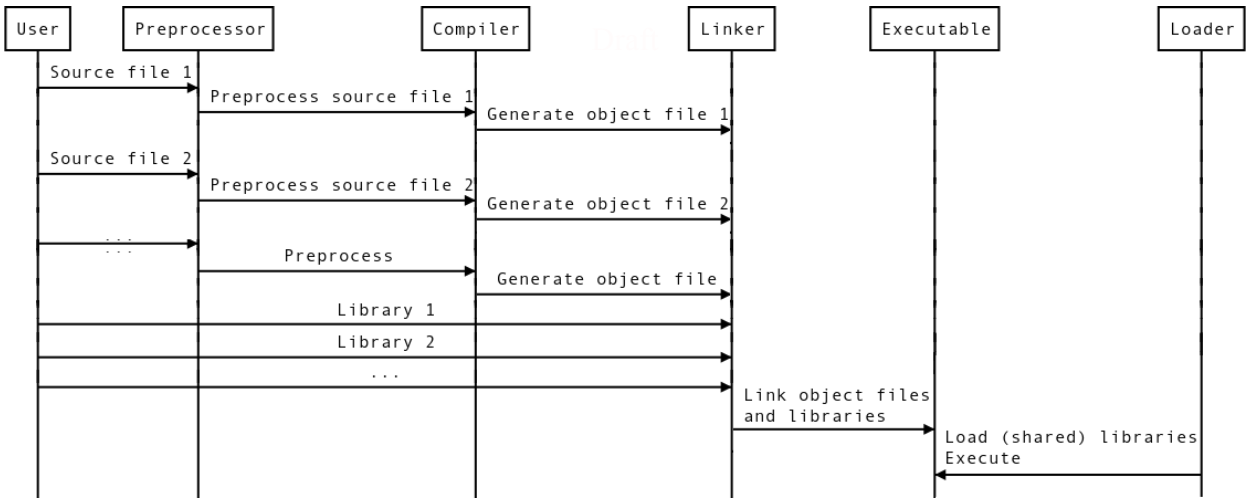

Figure 2.2: The Build Process

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:

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

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

Preprocessed source code:

```
// 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)
`bool`	1
(`unsigned`) `char`	1
(`unsigned`) `short`	2
(`unsigned`) `int`	4
(`unsigned`) `long`	4 or 8
(`unsigned`) `long long`	8
`float`	4
`double`	8
`long double`	8, 12, or 16

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

```
1 int age = 30;
2 short population = 32000;
3 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

```
1 float pi = 3.14;
2 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.
}
```

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;
```

2.4.1 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:

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

Explicit conversion:

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

2.5 Memory Management

2.5.1 Heap vs. Stack

Programs utilize two primary memory regions for storing data: the stack and the heap.

The **Stack Memory** is used for storing function call frames, local variables, and control information. It is:

- Allocated and deallocated automatically.
- Suitable for small, short-lived variables.
- Limited in size and operates in a last-in, first-out (LIFO) manner.

The **Heap Memory** is used for dynamic memory allocation, which is:

- Manually managed through `new` and `delete`.
- Suitable for objects with varying sizes or extended lifetimes.
- Larger in size than the stack but slower due to manual management.

Example: *Stack vs. Heap*

```
1 int stack_var = 42;          // Stored on the stack.
2 int* heap_ptr = new int(42); // Stored on the heap.
3
4 // Cleanup for heap memory.
5 delete heap_ptr;
6 heap_ptr = nullptr;
```

2.5.2 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.
```

2.5.3 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.

To be continued...

2.6 The Build toolchain in practice

2.6.1 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++`.

Example: *Preprocessor and Compiler*

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
```

2.6.2 Linker

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

Example: *Linker*

```
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.

2.6.3 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.

2.6.4 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
Makefile

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.

5 Integration with Python

5.1 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.

5.2 pybind11

5.2.1 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:

bash
```
git submodule add -b stable https://github.com/pybind/pybind11
    extern/pybind11
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.

5.2.2 Basics

All pybind11 code is written in C++. The following lines must always be included in your code:

C++

```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:

C++

```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:

terminal

```
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:

terminal

```
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.

python

```
import example
print(example.add(1, 2)) #output: 3
```

With a simple modification, you can inform Pythn about the names of the arguments:

C++

```
m.def("add", &add, "A function that adds two numbers",
py::arg("i"), py::arg("j"));
```

You can now call the function using keyword arguments:

python

```
import example
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

C++
```cpp
using namespace py::literals;
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:

C++
```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`.

C++
```cpp
int value = 42;
m.attr("value") = value;
```

python
```python
import example
print(example.value) #output: 42
```

5.2.3 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:

C++

```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; }
```

C++

```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

```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:

C++

```cpp
std::string __str__() const {
    return name + " is " + std::to_string(age) + " years old";
}
```

C++

```cpp
.def("__str__", &Pet::__str__);
```

python

```python
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:

C++

```cpp
.def_readwrite("name", &Pet::name)
```

python

```python
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:

C++

```cpp
.def_property("description", &Pet::__str__, &Pet::set_name)
```

python

```python
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
```

5.2.4 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:

C++

```cpp
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_`:

C++

```cpp
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);
}
```

python

```python
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:

C++

```cpp
py::class_<Pet> pet(m, "Pet");
pet.def(\dots);
py::class_<Dog>(m, "Dog", pet)
    .def(py::init<>());
    .def("speak", &Dog::speak);
```

python

```python
import example
animal = example.Pet()
dog = example.Dog()
print(animal.speak()) #output: I am an animal
print(dog.speak()) #output: I am a dog
```