Course: C++ language

Ali ZAINOUL

for Keyce Academy
May 20, 2023

Table of contents

Language Overview

- C++ programming language is an extension of the C programming language, which was developed by **Bjarne Stroustrup** as an enhancement to C with added features for object-oriented programming.
- C++ inherits the popularity of C and is widely used in various industries and domains. Some key reasons for using C++ include:
  - C++ is a powerful and expressive language.
  - C++ supports both procedural and object-oriented programming paradigms.
  - C++ provides high performance and efficient execution.
  - C++ is compatible with multiple operating systems and computer platforms.
- **A C++ program or C++ source code is typically written in one or more text files with the extension ".cpp" for source files and ".hpp" for header files.**

Language Overview

- C++ was developed as an extension of the C programming language, which itself was invented to write the operating system **UNIX**.
- C++ inherits some features from its predecessor, **C language**, which was invented in the 1970s.
- The C++ language was formalized by the **International Organization for Standardization** (ISO) in the year 1998.
- The **UNIX OS was initially written in C in 1973,** and C++ has become widely used in various domains since then.
- C++ is one of the most widely used and **popular** system programming languages.
- Many Linux Operating Systems and a multitude of industrial applications are written in C++.

Usages of C++ (Part 1)

C++ is a versatile programming language that finds applications in various domains. Its rich set of features and powerful capabilities make it suitable for the following use cases:

- **System Programming:** C++ is commonly used for system programming tasks, such as developing operating systems, device drivers, embedded systems, and firmware. Its low-level control and direct memory manipulation features make it well-suited for these applications.
- **Game Development:** C++ is widely used in the game development industry. Its high performance, efficient memory management, and access to low-level hardware interfaces make it an optimal choice.

Usages of C++ (Part 2)

- **Application Development:** C++ is used to build a wide range of applications, including desktop applications, scientific simulations, financial systems, CAD/CAM software, and performance-critical applications. Its ability to balance performance and productivity allows developers to create robust and efficient applications.
- **High-Performance Computing (HPC):** C++ is extensively used in the field of high-performance computing. It enables developers to leverage parallel processing, utilize multi-core architectures, and optimize code for computational-intensive tasks, such as simulations, data analysis, and scientific computing.

Usages of C++ (Part 3)

- **Graphics and Multimedia:** C++ is a popular choice for graphics programming, computer vision, image processing, and multimedia applications. Libraries like OpenGL and DirectX provide C++ bindings, allowing developers to create visually appealing and interactive graphical applications.
- **Networking and Communications:** C++ is utilized in the development of network protocols, network programming, and communication systems. Its ability to efficiently handle low-level socket programming, implement networking libraries, and build server applications makes it suitable for network-related tasks.

Usages of C++ (Part 4)

- **Library Development:** C++ is often used to create libraries and frameworks that can be utilized by other developers in their projects. Many widely used libraries and APIs, such as Boost, Qt, and the Standard Template Library (STL), are written in C++.

These are just a few examples of the numerous applications where C++ excels due to its performance, flexibility, and ability to handle low-level operations. Its widespread usage across different industries and domains showcases its versatility as a programming language.

Environment setup

In order to program in C++, you need the following:

- Either: (1): **a Text Editor** and (2): a **C++ compiler** using command line terminal
- Either: an **Integrated Development Environment** (IDE).

Environment setup

If you choose the first option: text editor plus a C++ compiler, you can choose from the following:

- Text editor: one can choose either Atom (consuming a lot of memory), Sublime Text, Visual Studio, gEdit, VIM, Notepad++.
- for C++ compiler: It all depends on your Operating System (OS)
  - **Linux / Unix based systems** (most recommanded): we recommand installing the most stable version of Linux called Ubuntu. Once it is done, one may install **GNU Compiler Collection** (**GCC**) by following these command lines:

```
sudo apt update
sudo apt install build-essential
sudo apt-get install manpages-dev
gcc –version
```

Environment setup

- for C compiler: It all depends on your Operating System (OS)
  - **Linux / Unix based systems**
    - The first command updates the system.
    - The second one installs a bunch of new packages including **gcc** (for C), **g++** (for C++) and make (for both languages).
    - The third command installs the manual pages about using GNU/Linux for development.
    - The last one is used to check your version of GCC.
  - **Macos based systems**

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install gcc
gcc –version
```

    - The first command installs Homebrew (The Missing Package Manager for macOS)
    - The second one tells Homebrew to install GCC
    - The last one checks which version of gcc is installed.

Environment setup

- for C compiler: It all depends on your Operating System (OS)
  - **Windows based systems** One could follow the steps in this website: Installing GCC using command line on Windows

Environment setup

If one has chosen to use the second option (i.e. installing an Integrated Development Environment), one may install one of the following:

- Visual Studio Code
- Code::Block
- Codelite
- NetBeans

Program Structure (Part 1)

A C++ program consists of various components, including:

- **Preprocessor Directives:** The preprocessor commands, starting with a hash symbol (#), are used to include header files, define constants, and perform macro substitutions before the code is compiled.
- **Functions:** Functions in C++ are blocks of code that perform specific tasks. They can be defined and called within the program to achieve modular and reusable code.
- **Variables:** C++ allows the declaration and use of variables to store and manipulate data. Variables must be declared before they are used and can have different types, such as integers, floating-point numbers, characters, or user-defined types.

Program Structure (Part 2)

A C++ program also includes the following components:

- **Statements:** C++ programs consist of a series of statements that define actions to be executed. Statements can include assignments, conditionals, loops, function calls, and more.
- **Expressions:** Expressions are combinations of variables, constants, and operators that produce a value when evaluated. They can be used for calculations, comparisons, and other operations.
- **Comments:** Comments in C++ are used to document and explain the code. They are not executed and are ignored by the compiler.

Creating a new project

To create a C++ program, open a text editor or your favorite IDE, create a new file, and name it "**helloCWorld.cpp**". Copy and paste the code below:

```
// This is a comment
/*
This is a comment on many
lines.
*/
#include <iostream>
using namespace std;

int main()
{
// This is my first C++ program
cout « "Hello, C++ World!" « endl;
return 0;
}
```

Code Analysis

Let’s analyze the ‘helloCWorld.cpp‘ code above:

- The first line is a comment, which is not considered as code and will be ignored by the compiler.
- The second block represents a comment spanning multiple lines, just like in the previous code.
- The ‘#include <iostream>‘ statement is used to include the ‘iostream‘ library, which provides input/output functionality in C++.
- In C++, we use the ‘cout‘ statement from the ‘iostream‘ library to output text. The line ‘cout « ”Hello, C++ World!” « endl;‘ will print the text ”Hello, C++ World!” to the console. The ‘endl‘ is used to insert a newline character after the text.

Compiling and Executing the Program

Let's look at how to save the source code in a file, compile it, and run it:

- **Open a text editor** or your **favorite IDE** and **copy-paste the 'HelloCWorld.cpp' code** above. Save the file in a directory as **'HelloCWorld.cpp'**.
- **Open a command prompt** or terminal and **navigate to the directory** where your file is saved.
- Type "**g++ -o myHelloProgram HelloCWorld.cpp**" and **press enter** to compile your code.
- **If there are no errors in your code**, the command prompt will move to the next line, and an executable named '**myHelloProgram**' will be generated.
- To **run the program**, type "**./myHelloProgram**" and **press enter**.
- You will see **"Hello, C++ World!"** printed on the screen.

Compiling and executing the program

```
$ cd my_directory_where_is_my_file
$ g++ -o myHelloProgram helloCWorld.cpp
$ ./myHelloProgram
Hello, C++ World!
%
```

Basic Syntax

- This section gives details about the basic syntax of the C++ language, including tokens, keywords, identifiers, etc.
- We saw a basic structure of a C++ program (helloCWorld.cpp), so it will be easy to understand the basic building blocks of the C++ language.

Basic Syntax of a C++ Program (Part 1)

In C++, a program consists of a collection of statements that are written in a specific syntax. Here are some key elements of the basic syntax in C++:

- **Functions**: A C++ program starts with a special function called the `main()` function, which serves as the entry point of the program. It is where the execution begins and ends.
- **Statements**: C++ programs are composed of statements that perform specific actions. Each statement typically ends with a semicolon (;) to indicate the end of the statement.
- **Variables**: Variables are used to store and manipulate data in C++. Before using a variable, you need to declare it by specifying its type and a unique name. For example, `int age;` declares an integer variable called `age`.

Understanding these elements of the basic syntax in C++ is crucial for writing correct and efficient code. They provide a foundation for

Basic Syntax of a C++ Program (Part 1)

In C++, a program consists of a collection of statements that are written in a specific syntax. Here are some key elements of the basic syntax in C++:

- **Functions**: A C++ program starts with a special function called the `main()` function, which serves as the entry point of the program. It is where the execution begins and ends.
- **Statements**: C++ programs are composed of statements that perform specific actions. Each statement typically ends with a semicolon (;) to indicate the end of the statement.

Basic Syntax of a C++ Program (Part 2)

In addition to functions and statements, C++ also includes other important components in its basic syntax:

- **Variables**: Variables are used to store and manipulate data in C++. Before using a variable, you need to declare it by specifying its type and a unique name. For example, `int age;` declares an integer variable called `age`.
- **Data Types**: C++ provides various data types to represent different kinds of values, such as integers (`int`), floating-point numbers (`float`), characters (`char`), and more. Each data type has a specific range and behavior.

Basic Syntax of a C++ Program (Part 3)

Apart from functions, statements, variables, and data types, C++ also includes the following important components in its basic syntax:

- **Comments**: Comments in C++ are used to add explanatory or descriptive text within the code. They are ignored by the compiler and do not affect the program's execution. Comments can be single-line (`//`) or multi-line (`/* ... */`).
- **Control Structures**: C++ supports control structures like `if` statements, `for` loops, `while` loops, and `switch` statements to control the flow of execution in a program based on certain conditions.

Understanding these components of the basic syntax in C++ is important for writing well-structured and maintainable code. They allow you to add clarity and flexibility to your programs by incorporating comments and controlling program flow.

Tokens in C++

- A C++ program consists of various **tokens**.
- A token can be a **keyword**, an **identifier**, a **constant**, a **string literal**, or a **symbol**.
- Tokens are the fundamental units of the language and are used to build meaningful instructions and expressions in C++.

Tokens in C++

- The tokens in the given C++ statement are:

```
cout
«
”Hello, C++ World! \n”
«
;
```

Tokens in C++

- The tokens above consist of:
  - The **cout** keyword, which is a standard C++ object used for outputting data.
  - The **«** operator, which is the insertion operator used to insert data into the output stream.
  - The string literal **"Hello, C++ World! \n"**, which represents the text to be printed.
  - Another **«** operator to continue inserting data.
  - The semicolon **;** to terminate the statement.

Semicolons ; in C++

- In C++, a semicolon (;) is also a statement terminator. Each line of code should end with a semicolon to indicate the completion of a statement.
- Let's consider the following example:

```
cout « ”Hello, C++ World! \n” ;
return 0 ;
```

- In the code above, we have two statements. The first statement uses the **cout** object to output the text ”Hello, C++ World!” to the console. The second statement is the **return** statement, which returns the value 0 from the **main** function.

Comments in C++ - Single-line

- Single-line comments in C++ start with // and extend until the end of the line. They are used to add explanatory or descriptive information to the code.
- Example:

```
// This is a comment on a single line
```

Comments in C++ - Multi-line

- Multi-line comments in C++ start with /* and end with */. They can span multiple lines and are used to provide more detailed explanations or to temporarily disable code.
- Example:

```
/* This is a comment that can
span multiple lines until the closing */
```

Identifiers in C++ - Naming Rules

- In C++, an identifier is a name used to identify variables, functions, classes, and other entities.
- The naming rules for identifiers in C++ are as follows:
  - An identifier must start with a letter (a-z, A-Z) or an underscore (_).
  - After the first character, an identifier can contain letters, digits (0-9), and underscores.
  - C++ is case-sensitive, so ”myVariable” and ”myvariable” are treated as two different identifiers.
- Special characters such as @, $, and % are not allowed in identifiers.

Identifiers in C++ - Examples

- Examples of valid identifiers in C++:
  - Variable: **myVariable, age, num1**
  - Function: **calculateSum, printMessage**
  - Class: **Student, Circle, Rectangle**
- Examples of invalid identifiers in C++:
  - **3numbers** (starts with a digit)
  - **my@variable** (contains special character @)
  - **my-variable** (contains hyphen)

Keywords in C++

- There is **reserved words** that must not be used as constant or variable or as any other identifier name. The **reserved words** are reserved to the language itself. The non-exhaustive list is below:

alignas	alignof	and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	char8_t	char16_t	char32_t
class	compl	concept	const	consteval	constexpr	constinit	const_cast
continue	co_await	co_return	co_yield	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit	export	extern	false
float	for	friend	goto	if	inline	int	long
mutable	namespace	new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	reflexpr	register	reinterpret_cast
requires	return	short	signed	sizeof	static	static_assert	static_cast
struct	switch	synchronized	template	this	thread_local	throw	true
try	typedef	typeid	typename	union	unsigned	using	virtual
void	volatile	wchar_t	while	xor	xor_eq		

Figure: Reserved keywords

Whitespaces in C++

- A line containing only whitespaces, possibly within (or not) a comment, is known as a **blank line**, and a C++ compiler totally **ignores it**.
- **Whitespace** is the term used in C++ to describe **blanks**, **tabs**, **newline characters**, and **comments**.
- Whitespace separates one part of a statement from another one.

Whitespaces in C++

- Whitespace enables the compiler to identify where one element in a statement, such as **int**, ends and the next element begins. Therefore, in the following statement: **int mark;**
- There must be at least one whitespace character (usually a space) between **int** and the identifier **mark** in order for the compiler to be able to distinguish them. Otherwise, the compiler will identify **intmark** which may be an identifier on its own...

Data Types in C++

In C++, data types refer to the fact that every variable or function has its own type, either basic or generic (user-defined) and/or derived types. The type of a variable will determine how much space it must occupy in order to store it into memory and how the bit pattern will arise. So, as stated, there are four types of variables in C++, classified as follows:

- **Basic types:** arithmetic types; either **integer** ones (natural numbers and integers in mathematics) or **floating-point** ones (real numbers).
- **Enumerated types:** arithmetic types too, used to define variables only assignable by certain **discrete integer values**.

Data Types in C++

- **Void type:** the **void** type indicates that no value is assigned. (think of void in the real world)
- **Derived types: derived** types may be pointers, arrays, structures, classes (in any Object-Oriented Programming languages), union types, and functions. (the list is not exhaustive).

Remark: In C++, there are two aggregate types: arrays and structures. Aggregate types are collections of scalar values.

Category	Type	Size (bits)	Range	Description
Basic Types	bool	1	true/false	Boolean value
	char	8	-128 to 127	Character
	wchar_t	16 or 32	Platform-dependent	Wide character
	int	32 or 64	Platform-dependent	Integer
	unsigned int	32 or 64	0 to 4294967295	Unsigned Integer
	short	16	-32768 to 32767	Short Integer

- The basic types in C++ include bool, char, wchar_t, int, unsigned int, and short.
- Each type has a specific size in bits and a range of values it can represent.

Category	Type	Size (bits)	Range	Description
Basic Types	unsigned short	16	0 to 65535	Unsigned Short Integer
	unsigned long	32 or 64	0 to 18446744073709551615	Unsigned Long Integer
	unsigned long long	64 or more	0 to 18446744073709551615	Unsigned Long Long Integer

- The basic unsigned types in C++ include unsigned short, unsigned long, and unsigned long long.
- Each type has a specific size in bits and a range of values it can represent.

Category	Type	Size (bits)	Range	Description
Derived Types	pointers	-	-	Store memory addresses
	arrays	-	-	Collection of elements of same type
	structures	-	-	Group of related variables

- Derived types in C++ include pointers, arrays, and structures.
- Pointers store memory addresses, arrays represent collections of elements of the same type, and structures group related variables together.

Category	Type	Size (bits)	Range	Description
Derived Types	classes	-	-	Blueprint for creating objects
	unions	-	-	Store different types in same memory space
	functions	-	-	Blocks of code with a name

- Derived types in C++ also include classes, unions, and functions.
- Classes serve as blueprints for creating objects, unions store different types in the same memory space, and functions are blocks of code with a name.

Data types

- In order to get the exact size of a type or a variable on a particular operating system with a particular compiler, you can use the sizeof operator. The expression sizeof(type) yields the storage size of the object or type in **bytes**.

Remark: be aware of the fact that the sizeof operator will return slightly different values depending on the architecture system, OS, compiler etc.

Data types: Format Specifiers in C++ (Part 1)

Format specifiers in C++ are used during input and output operations. They help specify the type of data in a variable when scanning a variable using `std::cin` or printing a variable using `std::cout`.

In C++, format specifiers are similar to those in C, but with some differences and additional features. They are used with stream extraction and insertion operators (>> and <<) to handle different data types and formatting options.

Data types: Format Specifiers in C++ (Part 2)

C++ provides various format specifiers for different data types and formatting options. Some commonly used format specifiers include:

- `%d` for integers
- `%f` for floating-point numbers
- `%c` for characters
- `%s` for strings
- `%p` for pointers
- `%x` or `%X` for hexadecimal values

It's important to use the correct format specifier to ensure proper input and output operations. Format specifiers can be combined with additional flags, field widths, and precision specifiers to control the formatting of the data.

Data types: Format Specifiers in C++ (Part 3)

Here are some examples of using format specifiers in C++:

Example 1: To output a floating-point number with 2 decimal places

```
double value = 3.14159;
std::cout << "The value is: " << std::fixed <<
std::setprecision(2) << value;
```

Example 2: To format a string with a specified width

```
std::string name = "John";
std::cout << std::setw(10) << std::left << name;
```

Explanation of Examples

- **Example 1:** In this example, we have a double variable value with the value 3.14159. We use std::fixed to set the output format to fixed-point notation and std::setprecision(2) to specify 2 decimal places. The output will display ”The value is: 3.14”.
- **Example 2:** In this example, we have a string variable `name` with the value ”John”. We use std::setw(10)’ to set the output width to 10 characters and `std::left` to align the string to the left within the specified width. The output will display ”John ” with additional spaces to fill the remaining width.

These examples demonstrate how format specifiers can be used to control the formatting of data during output operations in C++.

Data types: the void type

The **void type** is an important concept built-in type and specifies that no value is available. It is used in three particular situations:

- **The return_type of a function is of type void**. Function returns as void. There are various functions in C which do not return value hence they are returning **void type**. Example:

```
void draw_line() {
        std::cout « ”- – — - – — — - – — – -” « std::endl;
}
```

Data types: The void type (Part 1)

- **Function arguments as void**: Functions in C++ taking arguments list as void are simply functions with unknown parameters, whereas in C++, `f(void)` is exactly the same as `f()`.

Data types: The void type (Part 2)

- **Pointers to void**: A pointer of type **void *** represents the address of an object, but **not its type**. For example, the `new` operator in C++ can allocate memory and return a pointer to void which can be **casted** to any data type. For instance:

```
void *ptr = new int(10); // Allocate memory for an integer
int *intPtr = static_cast<int*>(ptr); // Cast void pointer to int pointer
delete intPtr; // Deallocate the memory
```

- **Remark:** The void type is an important concept in C++ for handling and manipulating memory areas, pointers, and casting. The only object that can be declared with the type specifier void is a pointer.

Variables in C++

Definition: A **variable** is a name given to a storage area that a program may manipulate.

- Each variable in C++ has:
  - **a specific type**, which determines the size and layout of the variable's memory;
  - **the range of values** that can be stored within that memory;
  - and **the set of operations** that can be performed on the variable.

Variables in C++

- The name of a variable in C++ must follow these two rules:
  - **It must begin** with either a letter or an underscore.
  - The variable can be composed of alphanumeric values and/or the underscore character. For example, _1, _myVar, a_var, var, and var123_L are all valid examples.
  - C++ programming language also allows for the definition of various other types of variables such as Enumeration, Pointer, Array, Structure, Union, etc. However, in this discussion, we will focus on basic variables.

  Remark: C/C++ are case-sensitive, so upper and lowercase letters are distinct. For instance, myVar is **not** the same as myvar.

Variables

- In the following frames, we will highlight the difference between the **declaration**, the **definition** and the **initialization** of a variable, ***AN IMPORTANT CONCEPT*** in programming languages, and especially in C/C++ ones.

Variables
Variable **Declaration** in C/C++

Definition: Declaration of a variable is generally an introduction to a new memory allocated and highlighted by an identifier with a given type (built-in, basic or user-defined).

- The three properties of declaration are:
  - Memory space creation occurs at the same time of declaration itself.
  - Variables declared ***but not defined or initialized*** may have garbage values (random value of the given type).
  - Variables ***CANNOT*** be used before declaration.

Variables
Variable **Declaration** in C/C++

- It is done by the following:

```
type variable_list;
```

- The **type *must*** be a **valid C++ data type** including: char, int, float, double, bool or any user-defined object, etc.

- **variable_list** may consist of **one** or **more** identifier names separated by commas. Some valid declarations are shown below:

Variables
Variable **Declaration** in C/C++

```
// This is a simple comment and will be ignored by the compiler
int a; // Declaration of variable a with int type
char b, c; // Declaration of distinct variables b and c with char type
float d, e, f; // Declaration of distinct variables d, e and f with float type
double _d; // Declaration of the variable _d with double type.
```

Remark: the same is applied to functions, one may **declare** a function and **define** it later. E.g: void foo(); is a declaration of the function foo. And without giving it any definition, the code can compile provided that the function isn't used elsewhere in your program. (which is quite useless...).

Variables
Variable **Definition** in C++

Definition: Definition of a variable is assigning a value to a variable, typically with the **assignment operator =**. It is the fact of assigning a ***valid*** value to the previously declared variable.

```
int a, b; // Declaration of distinct variables a and b of type int
a = 1; // Definition of variable a
b = a + 1; // Definition of variable b as the value of variable a + 1 (=2 here)
```

Variables
Variable **Initialization** in C++

Definition: Initialization of a variable is simply the fact of assigning (defining) the value at the time of declaration.

For example:

```
float f = 0.3;
```

is declaring a variable with identifier f, of type float and defining / assigning to it the value 0.3.

Function **Declaration** and **Definition**

A **Function Declaration** as seen above has the signature:

```
return_type name_of_my_function(args if any);
```

Whereas a **Function Definition** is given by:

```
return_type name_of_my_function(args if any) {
        function_instructions;
}
```

Function Declaration and Definition in C++

A complete example is as follows:

```
// Function Declaration
int my_function();
int main() {
        /*
       Function call. The Definition may come:
       - just after the Declaration in the same source program,
       - or be part of another source program that must be compiled before we use the function.
       */
       // Function Call
       my_function();
}
// Function Definition
int my_function() return (-123);
// The function simply returns the integer value -123. The parentheses are here for the sake of exactness.
```

Notion of recursivity: recursive functions

Definitions:

- An algorithm is called **recursive** if it is defined by it-self.
- In the same manner, a function is called **recursive** if it references itself.

Rvalues and Lvalues in C++ (Part 1)

In C++, expressions can be categorized as **rvalues** or **lvalues**.

- **Lvalues** represent objects with a specific memory location, such as variables or named objects.
  Example:

  ```
  int x = 5; // Here, x is an lvalue.
  ```

- **Rvalues** refer to temporary objects that do not have a specific memory location.
  Example:

  ```
  int sum = 2 + 3; // Here, 2 + 3 is an rvalue.
  ```

- Understanding the distinction between rvalues and lvalues is crucial for C++ programming.

Rvalues and Lvalues in C++ (Part 2)

In C++, rvalues and lvalues have different properties and behaviors.

Lvalues:

- They have a specific memory location.
- They can be assigned to and used as references.

Rvalues:

- They do not have a specific memory location.
- They cannot be assigned to or used as references directly.

Understanding these concepts is essential for effective C++ programming.

Rvalues and Lvalues in C++ (Part 3)

In addition to basic rvalues and lvalues, C++ introduces **rvalue references** and **move semantics**.

Rvalue reference:

- An rvalue reference can bind to an rvalue and potentially modify it.

Move semantics:

- Move semantics optimize the transfer of resources between objects, improving efficiency.

By leveraging these concepts, C++ provides powerful tools for efficient programming.

Rvalues and Lvalues in C++ (Summary)

- **Lvalues** represent objects with a specific memory location and can be assigned to.
- **Rvalues** refer to temporary objects without a specific memory location.
- C++ introduces **rvalue references** and **move semantics** to optimize temporary object handling.
- Rvalue references enable binding and potential modification of rvalues.
- Move semantics facilitate efficient resource transfer between objects.

More examples on Lvalues and Rvalues in C++

- Continuing with examples of lvalues and rvalues:

  - Example 1: Passing an lvalue as a function argument
    - `int square(int x) { return x * x; }`
    - Here, `x` is an lvalue that is passed as an argument to the `square` function.
  - Example 2: Using an rvalue reference
    - `int&& rvalueRef = 10;`
    - Here, `rvalueRef` is an rvalue reference that can bind to an rvalue.

Constants and literals

Definition: constants refers to fixed values that a program may not alter during its execution. These fixed values are called literals.

- Constants can be of any of the basic data types:
  - an **integer** constant;
  - a **floating** constant;
  - a **character** constant;
  - a **string** literal;
  - an **enumeration** constant.

Number systems

We distinguish four common number systems used in everyday communication protocol. We detail them as following:

- **Binary system**, working on base **2**;
- **Octal system**, working on base **8**;
- **Decimal system**, working on base **10**;
- **Hexadecimal system**, working on base **16**.

Number systems

Here is a picture illustrating the equivalence between the systems:

Decimal	Binary	Octal	Hexadecimal
0	0000	0	0
1	0001	1	1
2	0010	2	2
3	0011	3	3
4	0100	4	4
5	0101	5	5
6	0110	6	6
7	0111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Figure: Number systems

Constants and literals
Integer Literals

An integer literal may be a *decimal*, *octal*, or *hexadecimal* constant. A prefix specifies the base or radix:

- **0x** or **0X** for **hexadecimal** (base **16**);
- **0** for **octal** (base **8**);
- **nothing** for **decimal** (base **10**).

An example of integer literals:

Constants and literals
Float Literals

A floating-point literal is composed of:

- an integer part;
- a decimal point;
- a fractional part;
- an exponent part.

Floating point literals may be represented either in **decimal** form or **exponential** form.

Constants and literals
Float Literals

- **Decimal form**: must include the decimal point, the exponent, or both;
- **Exponential form**: one must include the integer part, the fractional part, or both. While the signed exponent is introduced by lowercase or uppercase **e** (**e** or **E**).

Some examples from IBM website:

floating-point constant	Value
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100000
7.321E-3	0.007321
3.2E+4	32000
0.5e-6	0.0000005
0.45	0.45
6.e10	60000000000

Figure: float literals

Constants and literals
Character constants

Definition: Character literals are enclosed in single quotes and can be stored in a simple variable of char type.

- a character literal can be a plain character (e.g. char a = 'a' ;) ;
- an escape sequence '\t' (we will come to this later) ;
- an universal character (we will come to this later).

Constants and literals

Character constants

Certain characters in C, when preceded by a backslash, have a special meaning. A non-exhaustive list is below:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

Figure: Escape Sequences

Constants and literals
String literals

Definition: String literals or constants are enclosed in double quotes "".
A string is simply an array of character literals: plain characters, escape sequences, and/or universal characters.

Constants and literals
String literals

- We may want to split a long string into multiple ones using literals by separating them using whitespaces. One example may be the following:

```
”Hello world!”
”Hello \
world!”
”Hello ” ”w” ”orld!”
```

Constants and literals
Defining Constants

There are two ways in order to define constants in C and C++ :

- with the **#define** preprocessor directive;
- with the **const** keyword. Here are the two ways:

  - #define my_Identifier my_Value (e.g. #define PI 3.14)
  - const type my_Identifier = my_Value; (e.g. const float PI = 3.14...)

Remark: no semi-colon needed for preprocessor constants. **Note:** it is recommanded / a good practice to define constants in CAPITAL letters.

Storage Classes in C++

Definition: In C++ programming language, a **Storage Class** defines the scope (visibility) and lifetime of variables and functions in a C++ program. These specifiers (a.k.a storage classes) precede the type that they modify.

C++ also has four main storage classes: **auto**, **extern**, **static**, and **register**. The storage class for a variable is specified as follows:

```
storageClass variableDataType variableName;
```

These storage classes provide different behaviors and properties for variables and functions in a C++ program.

Operators

Definition: In mathematics and computer programming, an **operator** is a symbol or character that tells the compiler to perform specific mathematical or logical processes.

Every programming language has its own built-on operators, however the operators presented here are common to many programming languages. C provides these built-in operators: **Arithmetic** operators, **Relational** operators, **Logical** operators, **Logical** operators, **Bitwise** operators, **Assignment** operators and **Miscellaneous** operators (Misc).

Operators
Truth Table

An important concept about manipulating operators is the truth table. Assume that we have two propositions (P) and (Q). Then:

P	Q	P AND Q	P OR Q	P XOR Q	NOT P	NOT Q
0	0	0	0	0	1	1
0	1	0	1	1	1	0
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Operators
Arithmetic Operators

The following table explicits the C built-in Arithmetic Operators.
This example shows these arithmetic operators in a C program.

Operator	Meaning
+	Adds the Right Hand Side (RHS) operand value to the Left Hand Side (LHS) operand value (a+b)
−	Subtracts the RHS operand value from the LHS operand value (a-b)
*	Multiplies the LHS and the RHS operand values together (a*b)
/	Divides the LHS operand value by the RHS operand value (a/b)
%	Modulus returns the remainder of an euclidian division of the LHS by the RHS (a%b)
++	Increments an integer value by one (++a **preincrément**) or (a++ **postincrément**)
−−	Decrements an integer value by one (–a **predécrement**) or (a– **postdecrément**)

Operators
Relational Operators

The following table explicits the C built-in relational operators.
This example shows these Relational Operators in a C program.

Operator	Meaning
==	Evaluated to true only if the two operands are equal
!=	Evaluated to true only if the two operands aren't equal
>	Evaluated to true only if LHS (Left Hand Side) operand value is strictly bigger than the Right Hand Side (RHS)
<	Evaluated to true only if Right Hand Side (RHS) operand value is strictly bigger than the Left Hand Side (LHS)
>=	Evaluated to true only if LHS operand value is strictly bigger OR equal than the RHS
<=	Evaluated to true only if RHS operand value is strictly bigger OR equal than the LHS

Operators
Logical Operators

The following table explicits the C built-in logical operators. **This example** shows these Logical Operators in a C program.

Operator	Meaning
&&	Logical AND operator. The condition is true only if the two operands aren't zeroes.
||	Logical OR operator. The condition is true only if one of the operands isn't zero.
!	Logical NOT operator. It reverses the logic state. If condition is true, then logical NOT operator will make it false, and vice-versa.

Operators
Bitwise Operators

The following table explicits the C built-in Bitwise operators.
This example shows these Bitwise Operators in a C program.

Operator	Meaning
&	Bitwise binary AND Operator, evaluates to TRUE only if the two bits are the TRUE.
|	Bitwise binary OR Operator, evaluates to TRUE only if one of the bits is TRUE.
^	Bitwise binary XOR operator. evaluates to TRUE only if a bit is TRUE in one operand but not both.
~	Bitwise unary NOT operator (or Complement Operator). Complement Operator simply flips the bits (same logic as NOT).
«	Bitwise binary Left Shift Operator. The LHS operands value is moved left by the number of bits specified by the RHS operand.
»	Bitwise binary Right Shift Operator. The LHS operands value is moved right by the number of bits specified by the RHS operand.

Operators
Assignment Operators

The following table explicits the C built-in Assignment operators. **This example** shows these Assignement Operators in a C program.

Operator	Meaning
=	Assignment Operator. It assigns values from RHS operands to LHS operand.
+=	Add AND Assignment Operator. It adds RHS operand to the LHS operand and assign the result to the LHS operand.
-=	Substract AND Assignment Operator. It substracts RHS operand to the LHS operand and assign the result to the LHS operand.
*=	Multiply AND Assignment Operator. It multiply RHS operand to the LHS operand and assign the result to the LHS operand.
/=	Divide AND assignment Operator. It divides LHS operand with the RHS operand and assign the result to LHS operand
%=	Modulus AND Assignment Operator. Assigns remainder of the Euclidian Division of LHS operand by the RHS operand to LHS operand.
«=	Left Shift AND Assignment Operator.
»=	Right Shift AND Assignment Operator.
&=	Bitwise AND Assignment Operator.
^=	Bitwise Exclusive OR, AND Assignment Operator.
|=	Bitwise Inclusive OR, AND Assignment Operator.

Operators
Miscellaneous Operators

The following table explicits the C built-in Miscellaneous Operators.
This example shows these Miscellaneous operators in a C program.

Operator	Meaning
sizeof()	It returns simply the size of a variable in bytes.
&	It returns the address of a variable.
*	Pointer operator.
? :	Ternary Operator. Conditional Expression.

Operators
Precedence and **priority** in C++

Operator precedence decides the grouping of terms in an expression. It will affects how an expression is evaluated. Certain operators have higher precedence than others.

- **The rule is**: within an expression, higher precedence operators will be evaluated first.
- In the table below, operators with the highest precedence appear at the top, whereas those with the lowest precedence appear at the bottom.

Operators
Precedence and **priority** in C++

Operator	Classification	Associativity
() [] -> . ++ –	Postfix	Left to right
+ - ! ~ ++ – (type)* & sizeof	Unary	Right to left
*/ %	Multiplicative	Left to right
+-	Additive	Left to right
« »	Shift	Left to right
< <= > >=	Relational	Left to right
== !=	Equality	Left to right
&	Bitwise AND	Left to right
^	Bitwise XOR	Left to right
|	Bitwise OR	Left to right
&&	Logical AND	Left to right
||	Logical OR	Left to right
?:	Conditional	Right to left
= += -= *= /= %= »= «= &= ^= |=	Assignement	Right to left
,	Comma	Left to right

Alternate and Repeating Conditional Structures & Loops

- In algorithms and programming, we call a **conditional structure** the set of instructions that test whether a condition is true or not. There are three of them:
  - The conditional structure **if**:

```
if (condition) {
      my_instructions;
}
```

Alternate and Repeating Conditional Structures & Loops

- **Continuation:**
  - The Conditional Structure **if ...else** :

```
 if (condition) {
       my_instructions;
}
else {
       my_other_instructions;
}
```

Alternate and Repeating Conditional Structures & Loops

- **Continuation:**
  - The conditional structure **if ...elif ...else** :

```
 if (condition_1) {
      my_instructions_1;
}
elif (condition_2) {
      my_instructions_2;
}
else {
      my_other_instructions;
}
```

Alternate and Repeating Conditional Structures & Loops

- Furthermore, an **iterative conditional structure** allows the same series of instructions to be executed several times (iterations). The **while** (while) instruction executes the instruction blocks as long as the while condition is true. The same principle for the **for** loop (for), the two structures are detailed below:
  - The iterative Conditional Structure **while**:

```
while (condition)
     my_instructions;
```

For Loop in C++ (Index-based)

- The for loop in C++ can be used in an index-based manner to iterate over a range of values or elements of an array.
- Here's an example of using a for loop in an index-based approach:

```
for (int i = 0; i < n; i++) {
    // Access element at index i
    // Perform operations
}
```

- In this approach, a loop counter `i` is initialized to the starting index, and the loop continues until `i < n` is false, where `n` represents the size or length of the range or array.

For Loop in C++ (Index-based)

- The loop counter is incremented or decremented at the end of each iteration.
- You can use the loop counter `i` to access elements at different indices and perform operations within the loop body.
- This approach is useful when working with specific elements of an array or a defined range of values.

Index-based For Loop Examples

- Example 1:

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

- Example 2:

```
int numbers[] = {1, 2, 3, 4, 5};
int length = sizeof(numbers) / sizeof(numbers[0]);
for (int i = 0; i < length; i++) {
        std::cout << "Number: " << numbers[i] << std::endl;
}
```

For Loop in C++ (Element-based)

- The for loop in C++ can also be used in an element-based manner to directly iterate over elements of a container or a range-based sequence.
- Here's an example of using a for loop in an element-based approach:

```
for (const auto& element : container) {
    // Access element
    // Perform operations
}
```

- In this approach, the loop iterates over each element in the `container` (such as an array, vector, or any range-based sequence).

For Loop in C++ (Element-based)

- The loop variable `element` represents the current element being processed in each iteration.
- You can directly access and work with the elements within the loop body without the need for an explicit index.
- This approach is more convenient when you want to iterate over all elements in a container or a sequence without managing the loop counter.

Element-based For Loop Examples

- Example 1:

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (const auto& num : numbers) {
        std::cout << "Number: " << num << std::endl;
}
```

- Example 2:

```
std::string str = "Hello, World!";
for (const auto& ch : str) {
        std::cout << "Character: " << ch << std::endl;
}
```

Alternate and Repeating Conditional Structures & Loops

- **Continuation**:
  - The iterative conditional structure **do ...while**:

```
do {
     my_instructions;
} while(my_condition);
```

- **Note**: a difference lies between the **while** loop and the **do while** loop: in the **while** loop the check condition executes before the first iteration of the loop, while **do while**, checks the condition **after** the execution of the instructions inside the loop.

Scopes

Definition: A **scope** in any programming language is the area where a function or variable is visible and accessible. Whereas outside the same scope the function or variable is not accessible.

There are three regions where variables may be declared in C++:

- Inside a function or a block which is called **local variables**;
- Outside of all functions which is called **global variables**;
- In the definition of function parameters which is called **formal parameters**.

Scopes

Here are three code source examples of local, global variables and formal parameters:

- **Local Variables example;**
- **Global Variables** example;
- **Formal Parameters example.**

Scopes: behaviour of initializing local and global variables

When a **Local Variable** is **declared**, it is ***not initialized*** by the compiler. Whereas **Global Variables** are **initialized automatically** when declared:

Data Type	Default Value when declared
int	0
char	'\0'
float	0.0 or simply 0
double	0.0 or simply 0
Pointer	NULL

Important note: It is highly recommanded to **initialize** variables properly. Otherwise, the variable not initialized will have a garbage value already available at it's memory location, and it may lead to an unexpected behaviour and/or results.

Arrays

- An array is a collection of data elements of the same type (generally) that satisfies the following conditions:
  - A collection of data that have the **same type** (bool, int, double, string...)
  - Items are stored **contiguously** in memory (image 5)
  - The **size of the array** must be known when declaring the array.
  - Considering an array of size *N*, the **index of the first element** of the array is 0, while the **index of the last element** is $N-1$. (in most programming languages: C/C++, Python etc.)
  - The elements of an array are **accessible** from their **position**, hence their **index**.

Example of array representation
in memory of an array of ints

Memory Location	400	404	408	412	416	420
Index	0	1	2	3	4	5
Data	18	10	13	12	19	9

Figure: arrayExample

Arrays

- Advantages of arrays:
  - **Code optimization:** the complexity to add or delete an element at the end of the array, access to an element *via* its index, add an element or delete it is in $O(1)$. The **sequential** (linear) search for an element in an array has a complexity of $O(n)$.
  - **Ease of access:** direct access to an element *via* its index (e.g: myArray[3]), iterating over the elements of an array using a loop, simplified sorting.
- Drawbacks of arrays:
  - The constraint of having to declare its size prior to **compile time** (what is called a **static array**, we will come back to this), and that the size of the array does not grow during **runtime**.
  - Inserting and deleting items can be expensive as items have to be managed according to the newly allocated memory area (Example: wanting to add an item in the middle of an array).

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n]$: creates an array of size *n* (*n* elements) of type textbftypename with random values. **Example: int a**$[3]$ produces:

```
Index: 0 | address: 0x7ffee253c4c0 | value: -497826512
Index: 1 | address: 0x7ffee253c4c4 | value: 32766
Index: 2 | address: 0x7ffee253c4c8 | value: 225226960
```

  - **typename myArray**[n]=$\{v_1, ..., v_n\}$: creates an array of size *n* of type **typename** with values initialized to v_1 until v_n. **Example: int a**$[3] = \{1, 4, 9\}$ creates an array of 3 elements initialized to 1, 4 and 9 respectively.

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n] = \{v, ..., v\}$: creates an array of size *n* of type **typename** with all values initialized to *v*. **Example: int a**$[5] = \{1, 1, 1, 1, 1\}$ creates an array of 5 elements, all initialized to 1.

```
Index: 0| address: 0x7ffeea95b510| value: 1
Index: 1| address: 0x7ffeea95b514| value: 1
Index: 2| address: 0x7ffeea95b518| value: 1
Index: 3| address: 0x7ffeea95b51c| value: 1
Index: 4| address: 0x7ffeea95b520| value: 1
```

Arrays

- Declaration of an array in C/C++:
  - **typename myArray**$[n] = \{\}$: creates an array of size *n* of type **typename** with all values initialized to 0. **Example: int a**$[4] = \{\}$ creates an array of 4 elements, all initialized to 0.

```
Index: 0| address: 0x7ffee3257510| value: 0
Index: 1| address: 0x7ffee3257514| value: 0
Index: 2| address: 0x7ffee3257518| value: 0
Index: 3| address: 0x7ffee325751c| value: 0
```

- **typename myArray**$[n] = \{v\}$: creates an array of size *n* of type **typename** with the first value initialized to *v*, the rest to 0 .

Pointers
Introduction to Pointers

Definition: Compile time is the period when the soure program code (such as C, C++, C#, Java, or Python) is converted to binary code.

Definition: Run time is the period of time when a program is running and generally occurs after compile time.

Pointers
Introduction to Pointers

Definition: A **memory address** or **address** is the location of where a variable is stored on the computer. An address is always a non-negative integer. The address of a variable can be accessed using ampersand (&) operator, it denotes the address of the variable.

Definition: **Memory allocation** is the process of reserving (virtual or physical) **computer space** or **location** for a computer program to run.

Pointers
Introduction to Pointers

- Two main types of memory allocation exists:

Definition: Static Memory Allocation at Compile Time (referred also to Compile Time Memory Allocation). It occurs when declaring a variable, the compiler allocates automatically a memory location for it.

Definition: Dynamic Memory Allocation at Run Time (referred also to Run Time Memory Allocation). It occurs when memory can be allocated for data variables after the program begins execution.

Pointers in C++ (Part 1)

Definition: Pointers, or pointer variables, are special variables in C++ that are used to store addresses. A pointer is a variable that holds the memory address of another variable as its value.

One should distinguish two types of defining a pointer in C++:

- **Defining a pointer at the stack**: In C++, when declaring a pointer that points to a local variable or function, it is stored on the stack. The stack is a region of memory used for local variables and function call information. You can learn more about how function calls resolve in a C++ program, including the processing of local variables and pointers to local variables, in this article.

Pointers in C++ (Part 2)

- **Defining a pointer at the heap**: In C++, you can declare a pointer dynamically using the `new` operator or other built-in functions like `malloc()`, `calloc()`, or `realloc()` provided by the standard library. This allows you to allocate memory dynamically on the heap. The heap is a region of memory used for dynamic memory allocation.

Pointers

Assume we have the declaration of a variable of type **typename** and name **var**:

```
// Declaration:
typename var;
```

Here we are declaring a pointer of type **typename *** and name **pvar**:

```
// Declaration:
typename * pvar;
```

Pointers

- Some important notes:
  - **pvar** is simply a convention in order to be consistent about the naming of our variables. Here we are startit the name pvar with letter p in order to say that it is a pointer, pointing on variable var. Hence **pvar** as the name of our pointer.
  - **typename** have to be one of the built-in types: char, int, float, double.

Important note: Let's now point the pointer **pvar** to variable **var**:

```
// Definition:
pvar = &var;
```

Pointers

- An important fact about pointers is:
  - ***pvar** refers to the value of **var**.
  - **pvar** refers to the address of variable **var**: (**&var**).
  - **&pvar** refers to the address of the pointer (which is simply a variable that owns an address).

Pointers

Let's now explain with a real example how it is done:

```
 // Declaration of a variable f
float f;
// Initializing f to 1.2
f = 1.2;
// Declaration of pointer pf
float * pf;
// Setting the pointer pf to the address of f
pf = &f;
// Now pf is pointing to f.
```

Pointers

One can define directly a pointer as follows:

```
 // Declaration of a variable f
float f;
// Initializing f to 1.2
f = 1.2;
// Definition of pointer pf
// Declaring pointer pf and setting it to the address of f
float * pf = &f;
// Now pf is pointing to f.
```

A more real example is given in this example: **pointersExample.c**

Pointer Arithmetic

- Pointer arithmetic is a feature in C++ that allows performing arithmetic operations on pointers.
- When you perform arithmetic operations on pointers, they are scaled based on the size of the data type they point to.
- Incrementing a pointer by one moves it to the next memory location of the corresponding data type.
- Similarly, decrementing a pointer by one moves it to the previous memory location.
- Pointer arithmetic is particularly useful when working with arrays and iterating over elements using pointers.
- However, it's important to be cautious with pointer arithmetic to avoid accessing invalid memory locations or causing undefined behavior.

Pointer Arithmetic Example

- Let's consider an example of pointer arithmetic and arrays.
- Suppose we have an integer array `arr` with five elements.
- We can define a pointer `ptr` that points to the first element of the array.
- By using pointer arithmetic, we can iterate over the array and print its elements as follows:

```
int* ptr = arr;
for (int i = 0; i < 5; i++) {
  cout << *(ptr + i) << " ";
}
```

- This code snippet demonstrates how we can access the elements of the array using pointer arithmetic, incrementing the pointer by the

Pointers and Arrays

- In C++, arrays and pointers have a close relationship.
- In most cases, the name of an array behaves like a constant pointer that points to the first element of the array.
- You can use pointers to access elements of an array using pointer arithmetic.
- For example, given an array `arr`, you can access its elements using a pointer as `*(arr + i)`, where `i` is the index.
- Pointers provide a way to iterate over arrays, manipulate their elements, and perform various operations efficiently.
- However, it's important to handle pointers and arrays carefully to avoid accessing out-of-bounds memory or causing undefined behavior.

Pointers and Structures

- Pointers can also be used with structures in C++.
- Pointers to structures allow dynamic allocation and manipulation of structure objects.
- You can allocate memory for a structure dynamically using the `new` operator or other appropriate memory allocation functions.
- Accessing members of a structure using a pointer is done using the arrow operator (`->`).
- For example, if `ptr` is a pointer to a structure `st`, you can access a member variable `member` as `ptr->member`.
- Pointers to structures are particularly useful when working with dynamically allocated data or passing large structures to functions efficiently.

Strings

Definition: A **string** in C is an array of characters which is terminated by a null character.

- The following **Definition** creates a string named **myString** that consists of the word ”Programming”.
- In order to hold the terminated character (null character) at the end of the array, the size of the character array which contains the string **must be** one character more than the number of characters in the word ”Programming”.

```
char myString[12] = {'P','r','o','g','r','a','m','m','i','n','g','\0'};
```

Strings

- One may declare the same string by following the rules of array initialization (**literal string it will be**), and without writing the null terminated character, the following example highlight the how:

```
char myString[] = "Programming";
```

Here is the memory representation of the previously defined character array "Programming":

Figure: String Representation in Memory

Strings

- The header file **#include<string.h>** contains several functions that can be performed into strings, and string literals. Here is a non-exhaustive list:

  - **strcpy(s1, s2);** It will copies string s2 into string s1.
  - **strcat(s1, s2);** It concatenates string s2 onto the end of string s1.
  - **strlen(s);** It returns the length of the string s.
  - **strcmp(s1, s2);** It returns 0 if s1 and s2 contains the same characters; a negative number if s1<s2 and a positive one if s1>s2. More details may be found in **this link**.
  - **strchr(s, c);** It returns a pointer to the first occurrence of character c in string s.
  - **strstr(s1, s2);** It returns a pointer to the first occurrence of string s2 in string s1.

Structures

Definition: In C language, a **struct** or structure, is a collection of variables that may be of different types, under a single name.

We define a structure as follow:

```
struct structTag {
        typename_1 var_1;
        typename_2 var_2;
        ...
        typename_n var_n;
} structTypeTag;
```

Differences Between `struct` in C and C++

C Struct	C++ Struct
Members are public by default	Members are private by default
Cannot have member functions	Can have member functions
Cannot have constructors or destructors	Can have constructors and destructors
Does not support inheritance	Supports inheritance
Does not support access specifiers	Supports access specifiers (public, private, protected)
Cannot contain static member variables	Can contain static member variables
Size is not guaranteed to be the same across different compilers	Size is guaranteed to be the same as `long` in C++

Example: Student Structure (Part 1)

- Consider a scenario where you need to store information about students, such as their name, roll number, and age.
- You can use a structure in C++ to represent this data in a single entity.

Example: Student Structure (Part 2)

- Here's an example of defining a structure named `Student` with three member variables:

```
struct Student {
    std::string name;
    int rollNumber;
    int age;
};
```

- In this example, we define a structure named `Student` with three member variables: `name` of type `std::string`, `rollNumber` of type `int`, and `age` of type `int`.

Example: Rectangle Structure (Part 1)

- Structures can also be used to represent more complex entities, such as geometric shapes.
- Let's consider an example of defining a structure named `Rectangle` to represent a rectangle with its width and height.

Example: Rectangle Structure (Part 2)

- In this example, we define a structure named `Rectangle` with two member variables: `width` and `height`, both of type `double`.

```
struct Rectangle {
    double width;
    double height;
};
```

- You can create objects of the `Rectangle` structure to represent different rectangles and access its member variables to calculate their area or perform other operations.

Example: Employee Structure with Methods (Part 1)

- Structures in C++ can also contain member functions, allowing you to define methods that operate on the structure's data.
- Let's consider an example of an 'Employee' structure with member variables 'name' and 'salary', along with methods to set and display the employee's information.

Example: Employee Structure with Methods (Part 2)

```
struct Employee {
    std::string name;
    double salary;
    void setInfo(const std::string& empName, double empSalary) {
        name = empName;
        salary = empSalary;
    }
    void displayInfo() const {
        std::cout << "Name: " << name << ", Salary: " << salary << std::endl;
    }
};
```

- In this example, we define an `Employee` structure with member variables `name` and `salary`. The `setInfo()` method sets the employee's information, and the `displayInfo()` method displays the employee's name and salary.

/* Fundamental Principles of Object-Oriented Programming (OOP) */

Fundamental Principles of OOP

Definition: Object-Oriented Programming (OOP) is one of the fundamental principles in computer programming, with origins dating back to the 1970s with the Simula and Smalltalk languages, but the principle quickly took off with the creation of the C++ language, which is an extension of the C language, with the aim of making software more robust.

Fundamental Principles of OOP (continued)

A useful mnemonic device for the **six fundamental principles** of Object-Oriented Programming is **"ACOPIE"**.

- **A***bstraction*
- **C***lass*
- **O***bject*
- **P***olymorphism*
- **I***nheritance*
- **E***ncapsulation*

Object

- In computer science, an object is a self-contained symbolic container that holds information and functions/methods related to a subject, manipulated in a program. The subject is often something tangible belonging to the real world.

Class

- The class is a special data structure in object-oriented programming languages.
- It describes the internal structure of data and defines the methods that will apply to objects of the same family (the same class) or type.
- It provides methods for creating objects whose representation will therefore be that given by the generating class. The objects are then said to be instances of the class. This is why an object's attributes are also called instance variables and messages are called instance operations or instance methods.

Encapsulation

- Encapsulation is the act of grouping data and methods that manipulate them into a single entity called a *capsule*.
- It allows for organized code, restricting access to certain portions from outside the capsule, and having robust code.

Abstraction

- Abstraction is one of the key concepts in object-oriented programming languages. Its main purpose is to manage complexity by hiding unnecessary details from the user.
- Abstraction is the process of representing a real-life object as a computer model.
- This essentially involves extracting relevant variables attached to the objects we want to manipulate and placing them into a suitable computer model.

Inheritance

- It is a mechanism for transmitting all the methods of a so-called ”mother” class to another called ”daughter” and so on.

Polymorphism

- The name of polymorphism comes from Greek and means ”many forms.” This characteristic is one of the essential concepts of object-oriented programming. While inheritance concerns classes (and their hierarchy), polymorphism is related to object methods.

Virtual Functions in C++

- Virtual functions are used in C++ to achieve runtime polymorphism, also known as dynamic polymorphism.
- When a derived class overrides a virtual function from its base class, the derived class's implementation is called instead of the base class's implementation.
- This allows different objects of the same base class to exhibit different behaviors based on their actual derived types.

Virtual Functions Example

- Consider a base class called ‘Shape‘ with a virtual function called ‘area()‘.
- We create two derived classes, ‘Rectangle‘ and ‘Circle‘, that override the ‘area()‘ function with their own implementations.
- We can then create objects of both classes and call the ‘area()‘ function on them, which will invoke the appropriate implementation based on the actual object type.

Friend Functions in C++

- Friend functions in C++ are non-member functions that have access to the private and protected members of a class.
- They can be useful for providing external functions with privileged access to a class's private data.
- Friend functions are not part of the class and do not have the 'this' pointer, as they are not associated with any particular object.

Friend Functions Example

- Suppose we have a class called ‘Car‘ with private member variables such as ‘make‘, ‘model‘, and ‘year‘.
- We can declare a friend function called ‘displayCarInfo()‘ that can access these private members and display the car’s information.
- This allows us to keep the member variables private while still providing a way to access and display the information externally.

Types of Polymorphism in C++

- Polymorphism refers to the ability of objects of different classes to respond to the same function call.
- In C++, there are two main types of polymorphism: compile-time polymorphism and runtime polymorphism.
- Compile-time polymorphism is achieved through function overloading and operator overloading.
- Runtime polymorphism is achieved through virtual functions and is applicable when the actual object type is determined during runtime.

Compile-time Polymorphism Example

- Function overloading is a form of compile-time polymorphism where multiple functions with the same name but different parameter lists are defined.
- For example, we can have an ‘add()‘ function that can perform addition for different data types like integers, floating-point numbers, or even strings.
- The compiler selects the appropriate function based on the arguments passed at compile time.

Runtime Polymorphism Example

- Runtime polymorphism is achieved through virtual functions and is useful when we want to perform different actions based on the actual object type.
- Consider a base class ‘Animal‘ with a virtual function ‘makeSound()‘. Derived classes like ‘Cat‘ and ‘Dog‘ override this function with their specific sound implementations.
- By declaring a pointer or reference of type ‘Animal‘ and assigning objects of different derived classes to it, we can call the ‘makeSound()‘ function, which will invoke the appropriate implementation based on the actual object type.

Complete Example

A complete example is _Shape Project.