**OBJECT-ORIENTED PROGRAMMING**

CLASSES AND OBJECTS

Object Oriented Language offers **3 high-level features** for programming solutions:

* **Encapsulation** – wrapping of **data members and member functions**. In other words, **Encapsulation** couples an object's data to the logic of its class. The class’s member functions or methods define its logic. Typically, an object's data remains private to its class. Privacy helps preserve the object's integrity.
* **Inheritance -** Inheritance is the relationship between classes where one class inherits the entire structure of another class. The base class defines the structure to be inherited by derived classes or sub-classes.
* **Polymorphism -** Polymorphism supports the multiplicity of behaviors for an identifier in an inheritance hierarchy. Multiple behaviors are associated with the same identifier. (Doing one thing in different ways).

These behaviors are distinguished through an **object's dynamic type**. The language selects the behavior for the identifier that is appropriate to the dynamic type. The dynamic type of an object may change throughout the identifier's lifetime. As the type changes, the behavior associated with the identifier may change.

**What is the dynamic type of an object?**

In C++, the dynamic type of an object refers to **the actual type of the object during runtime**, as opposed to its static type, which is known at compile time.

When we create an object in C++, we typically declare its type.

**For example,** we might create an object of type `**Animal**`. But during the program's execution, if that object is actually **pointing to** a more specific type, like **`Dog` or `Cat`,** then the **dynamic type of the object is `Dog` or `Cat`, not just `Animal`.**

When we call a function on a pointer or reference to the base class that actually points to an object of a derived class, **it's the dynamic type** that **determines which version of the function is called.**

So, in short, dynamic typing in C++ is about **determining the actual type of an object during runtime**, **which can be different from the type declared at compile time.**

**What is compile time and run time?**

Compile time and run-time are two distinct phases in the lifecycle of a program:

1**. Compile Time:**

* Definition: Compile time refers to the period during which the **source code of a program is translated into machine code** (or some intermediate form) **by a compiler.**
* Activities: During compile time, the compiler checks the **syntax and semantics of the code**, translates it into machine code, and performs various optimizations.
* Errors: Errors detected during compile time are **typically syntax errors, type errors,** or other issues that prevent the program from being translated into machine code.
* Output: The output of the compilation process is **typically an executable file**, library, or some other form of binary code that can be executed by the computer.

2. **Run Time**:

* Definition: Run time refers to the period during which a **program is executing on a computer system.**
* Activities: During run time, the **program's instructions are being executed by the CPU, and data is manipulated according to the program's logic**.
* Errors: Errors detected during run time are typically runtime errors, such as **division by zero, dereferencing a null pointer, or accessing out-of-bounds memory.**
* Output: The output of the program's execution may include **results, side effects, or errors** that occur while the program is running.

MODULARITY

In programming, modules are like **building blocks of code**. They **define classes** (blueprints for objects) and **their implementations** (how those objects work). Each module is **stored in its own file or group of files (generally .cpp and .h)**. We combine these individual modules to create a **complex software application.**

Each module is compiled (translated into machine code) separately, making it easier to manage and update your code.

**To Sum up**, modularity in programming is about breaking down your code into smaller, manageable parts.

BUILDING BLOCKS

The common building blocks of an object-oriented language include:

* **Values -** Represent **data or information** that can be manipulated or processed. They can be **numbers**, **strings**, or **other data types.**
* **Objects** – Instances of classes that encapsulate data (attributes) and behavior (methods or functions).
* **Variables** - Memory spaces in which we can store values, and the value can change during the program's execution.
* **References** - **Aliases** or **alternative** **names** for objects or variables already existing in memory.
* **Functions** – **Blocks of reusable code** that perform a **specific task** when called or invoked.
* **Types** –Define the **kind of data** a variable can hold or the **blueprint for creating objects** (classes).
* **Class Members** – **Data members (attributes)** and **member functions (methods)** of a class.
* **Templates** –Generic code structures that allow for the **creation of functions** or **classes** that work with **any data type.**
* **Namespaces** –Scopes used to **organize code elements** and **prevent naming conflicts.**

Relationship between **Objects, Variables, and Values**

An **object** **occupies a region of memory, holds a value,** and **may have a name**, but does not necessarily have a name.

A **variable** is a **named object**. It **occupies a region of memory**, **holds a value,** and **has a name**.

The **value** that an object holds is the **contents of the memory region allocated to the object** and **defines its current state**.

TYPES (OBJECTS)

The **type of an object** relates the object to **its underlying implementation** and **identifies the operations that the object can perform**.

TYPE CATEGORIES

* **Fundamental types:** correspond directly to the hardware facilities
* **Built-in types:** reflect the capabilities of the hardware facilities directly and efficiently
* **User-defined types:**
* **concrete types:** their representation is part of their definition and is known
* **abstract types:** their representation is not part of their definition and is unknown

DECLARATIONS

A declaration **associates an entity/name with a type**, telling the compiler **how to interpret the entity's identifier**. The entity may be a **variable, an object or a function.** The name is visible within the part of the program. That part of the program is called the **name's scope.**

SCOPE

**The scope of a name may be any one of:**

* **local scope:** the name has been declared within a **function** or **code block**
* **class scope:** the name has been declared as a **member of a class**
* **namespace scope:** the name has been declared as a **member of a named block**
* **global scope:** the name has **not been declared in any one of the above scopes**

**Global Scope can be of 2 types as well**

* **Internal Linkage:** Also known as "**static global scope**," this refers to names declared outside of any function or class, at the top level of a file. Names with file scope are accessible throughout the entire file in which they are declared.

**Eg: static int x = 10;**

* **External Linkage:** This refers to names declared at the global scope that can be accessed across multiple files in a program. Names with **external linkage** are typically declared using the extern keyword, and their definitions can be spread across multiple files. They serve as points of communication between different parts of a program.

**Eg:**

// Header.h

**extern int y;** // Declaration

// File1.cpp

#include "Header.h"

int y = 20; // Definition