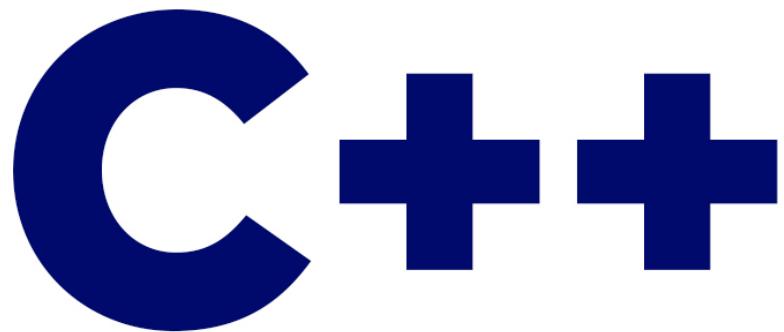


The Ultimate



Part 1: The Fundamentals

Mosh Hamedani
codewithmosh.com



Hi! I am Mosh Hamedani. I'm a software engineer with over 20 years of experience and I've taught millions of people how to code and become professional software engineers through my YouTube channel and coding school (Code with Mosh).

This PDF is part of my **Ultimate C++ course** where you will learn everything you need to know from the absolute basics to more advanced concepts. You can find the full course on my website.

<https://codewithmosh.com>

<https://www.youtube.com/c/programmingwithmosh>

<https://twitter.com/moshhamedani>

<https://www.facebook.com/programmingwithmosh/>

Table of Content

Getting Started.....	4
The Basics.....	6
Data Types.....	8
Decision Making.....	11
Loops.....	15
Functions.....	17

Getting Started

Terms

C++ Standard Library	Integrated Development Environment
Compiling	Machine code
Console application	Statement
Function	Syntax
Graphical User Interface (GUI)	Terminal

Summary

- C++ is one of the oldest yet most popular programming languages in the world due to its performance and efficiency.
- It's often used in building performance-critical applications, video games (especially with Unreal Engine), servers, operating systems, etc.
- To learn C++, you need to learn the syntax (grammar) of the language as well as C++ Standard Library, which is a collection of pre-written C++ code for solving common problems.
- To write C++ applications, we often use an Integrated Development Environment (IDE). The most popular IDEs are MS Visual Studio, XCode, and CLion.
- To run C++ applications, first we have to compile our C++ code to machine code.
- The main() function is the starting point of a C++ program.

Your First C++ Program

```
#include <iostream>

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

The Basics

Terms

Camel case	Operator
Comment	Pascal case
Constant	Snake case
Directive	Standard input stream
Expression	Standard output stream
Hungarian notation	Stream extraction operator
Mathematical expression	Stream insertion operator
Operand	Variable

Summary

- We use *variables* to temporarily store data in the computer's memory.
- To declare a variable, we should specify its type and give it a meaningful name.
- We should initialize variables before using them. Using an uninitialized variable can lead to unexpected behavior in our programs since these variables hold garbage values.
- Unlike variables, the value of *constants* don't change.
- The common naming conventions used in C++ applications are: **PascalCase**, **camelCase**, and **snake_case**.
- An *expression* is a piece of code that produces a value. A mathematical (arithmetic) expression consists of an operator (+, -, *, /, %) and two operands.
- Multiplication and division operators have a higher priority than addition and subtraction operators. So, they're applied first.

- We can use parentheses to change the order of operators.
- We use **cout** (pronounced sea-out) to write characters to the *Standard Output Stream* which represents the *terminal* or *console* window.
- We use **cin** (pronounced sea-in) to read data from the *Standard Input Stream* which represents the keyboard.
- We use the *Stream Insertion Operator* (`<<`) to write data to a stream.
- We use the *Stream Extraction operator* (`>>`) to read data from a stream.
- The *Standard Template Library* (STL) consists of several files each containing functions for different purposes.
- To use functions in the Standard Library, we should include the corresponding files using the **#include directive**.
- Using *comments* we can explain what cannot be expressed in code. This includes why's, how's, and any assumptions we made while writing code.

```
// Declaring a variable
int number = 1;

// Declaring a constant
const double pi = 3.14;

// Mathematical expressions
int x = 10 + 3;

// Writing to the console
cout << "x = " << x;

// Reading from the console
cin >> number;
```

Data Types

Terms

Array	Floating-point number
Binary system	Hexadecimal system
Boolean values	Index
Casting	Overflow
Characters	Run-time error
Compile-time error	Stream manipulator
Data type	String
Decimal system	Underflow

Summary

- C++ has several built-in *data types* for storing *integers* (whole numbers), *floating-point numbers* (numbers with a decimal point), characters, and Boolean values (true/false).
- Floating-point numbers are interpreted as **double** by default. To represent a **float**, we have to add the F suffix to our numbers (eg 1.2F).
- Whole numbers are interpreted as **int** by default. To represent a **long**, we have to use the **L** suffix (eg 10L).
- Using the **auto** keyword, we can let the compiler infer the type of a variable based on its initial value.
- Numbers can be represented using the *decimal*, *binary*, and *hexadecimal systems*.
- If we store a value larger or smaller than a data type's limits, *overflowing* or *underflowing* occurs.

- Using the **sizeof()** function, we can see the number of bytes taken by a data type.
- We can use *stream manipulators* to format data sent to a stream. The most common manipulators are **setw**, **fixed**, **setprecision**, **boolalpha**, **left**, and **right**.
- The Boolean **false** is represented as 0. Any non-zero number is interpreted as the Boolean **true**.
- In C++, characters should be surrounded with single quotes.
- Characters are internally represented as numbers.
- A *string* is a sequence of characters and should be surrounded by double quotes.
- We use *arrays* to store a sequence of items (eg numbers, characters, etc).
- Array elements can be accessed using an *index*. The index of the first element in an array is 0.
- When we store a smaller value in a larger data type, the value gets automatically *cast* (converted to) the larger type. When storing a large value in a smaller data type, we have to explicit cast the value.
- C-style casting involves prefixing a variable with the target data type in parentheses. In C++, we use the **static_cast** operator.
- C++ casting is safer because conversion problems can be caught at the *compile-time*. With C-style casting, we won't know about conversion issues until the *run-time*.

```
// Data types
double price = 9.99;
float interestRate = 3.67F;
long fileSize = 90000L;
char letter = 'a';
string name = "Mosh";
bool isValid = true;
auto years = 5;

// Number systems
int x = 255;
int y = 0b111111;
int z = 0xFF;

// Data types size and limits
int bytes = sizeof(int);
int min = numeric_limits<int>::min();
int max = numeric_limits<int>::max();

// Arrays
int numbers[] = { 1, 2, 3 };
cout << numbers[0];

// C-style casting
double a = 2.0;
int b = (int) a;

// C++ casting
int c = static_cast<int>(a);
```

Decision Making

Terms

Boolean expression
Comparison operators
Conditional operator
If statement
Logical operators
Nesting if statements
Switch statement

Summary

- We use *comparison operators* to compare values.
- A *Boolean expression* is a piece of code that produces a Boolean value.
- With *Logical operators* we can combine Boolean expressions and represent more complex conditions.
- With the *logical AND operator* (`&&`) both operands should be true. If either of them is false, the result will be false.
- With the *logical OR operator* (`||`), the result is true if either of the operands is true.
- The *logical NOT operator* (`!`) reverses a Boolean value.
- Using *if* and *switch statements*, we can control the logic of our programs.
- An if statement can have zero or more **else if** clauses for evaluating additional conditions.

- An if statement can optionally have an **else** clause.
- We can code an if statement within another. This is called *nesting* if statements.
- Using the *conditional operator* we can simplify many of the if / else statements.
- We use **switch** statements to compare a variable against different values.
- A switch block often has two or more *case labels* and optionally a *default label*.
- Case labels should be terminated with a **break** statement; otherwise, the control moves to the following case label.
- Switch statements are not as flexible as if statements but sometimes they can make our code easier to read.

```
// Comparison operators
bool a = 10 > 5;
bool b = 10 == 10;
bool c = 10 != 5;
```

```
// Logical operators
bool d = a && b; // Logical AND
bool e = a || b; // Logical OR
bool f = !a; // Logical NOT
```

```
if (temperature < 60) {
    // ...
}
else if (temperature < 90) {
    // ...
}
else {
    // ...
}
```

```
// Conditional operator
double commission = (sales < 10'000) ? .05 : .1;
```

```
switch (menu) {  
    case 1:  
        // ...  
        break;  
    case 2:  
        // ...  
        break;  
    default:  
        // ...  
}
```

Loops

Terms

Break statement	Iteration
Continue statement	Loops
Do-while statement	Loop variable
For statement	Nested loop
Infinite loop	While statement

Summary

- We use *loops* to repeat a set of statements.
- In C++, we have four types of loops: **for** loops, range-based **for** loops, **while** loops, and **do-while** loops.
- **For** loops are useful when we know ahead of time how many times we want to repeat something.
- Range-based **for** loops are useful when iterating over a list of items (eg an array or a string).
- **While** and **do-while** loops are often used when we don't know ahead of time how many times we need to repeat something.
- Using the **break** statement we can break out of a loop.
- Using the **continue** statement we can skip an iteration.

```
// For loop
for (int i = 0; i < 5; i++)
    cout << i << endl;

// Same algorithm using a while loop
int i = 0;
while (i < 5) {
    cout << i << endl;
    i++;
}

// Same algorithm using a do-while loop
i = 0;
do {
    cout << i << endl;
    i++;
} while (i < 5);

// Range-based for loop
int numbers[] = { 1, 2, 3 };
for (int number: numbers)
    cout << number << endl;
```

Functions

Terms

Debugging	Function signature
Functions	Global variables
Function arguments	Invoking a function
Function declaration	Header files
Function definition	Local variables
Function parameters	Namespaces
Function prototype	Overloading functions

Summary

- A *function* is a group of one or more statements that perform a task. Each function should have a clear responsibility. It should do one and only one thing.
- A function can have zero or more *parameters*
- *Arguments* are the values passed to a function.
- To *call* (or *invoke*) a function, we type its name, followed by parenthesis, and the arguments (if any).
- Function parameters can have a default value. This way, we don't have to provide arguments for them.
- The *signature* of a function includes the function name, and the number, order, and type of parameters.

- *Overloading* a function means creating another variation with a different signature. By overloading functions, we can call our functions in different ways.
- Arguments of a function can be passed by value or reference. When passed by value, they get copied to the parameters of the function.
- To pass an argument by a reference, we should add an & after the parameter type.
- *Local variables* are only accessible within the function in which they are defined. *Global variables* are accessible to all functions.
- Global variables can lead to hard-to-detect bugs and should be avoided as much as possible.
- A *function declaration* (also called a *function prototype*) tells the compiler about the existence of a function with a given signature. A *function definition* (or implementation) provides the actual body (or code) for the function.
- As our programs grow in more complexity, it becomes critical to split our code into separate files.
- A *header file* ends with “.h” or “.hpp” extension and consists of function declarations and constants. We can import header files using the **#include** directive.
- An *implementation file* ends with “.cpp” extension and consists of function definitions.
- Using *namespaces* we can prevent name collisions in our programs.
- *Debugging* is a technique for executing a program line by line and identifying potential errors.

```
// Defining functions
void greet(string name) {
    cout << "Hello " << name;
}

string fullName(string firstName, string lastName) {
    return firstName + " " + lastName;
}

// Parameters with a default value
double calculateTax(double income, double taxRate = .3) {
    return income * taxRate;
}

// Overloading functions
void greet(string name) {

}

void greet(string title, string name) {

}

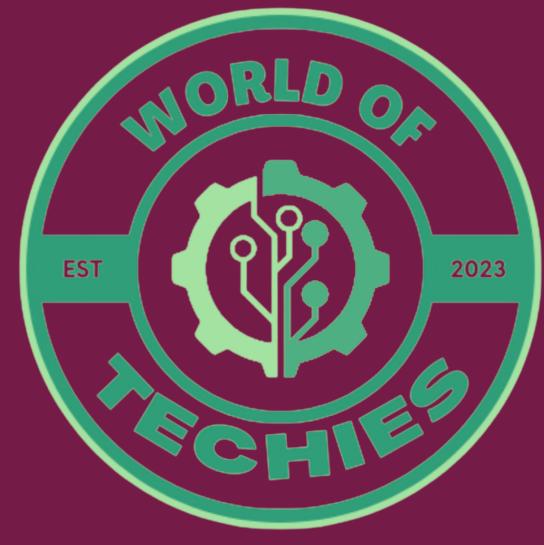
// Reference parameters
void increase(double& number) {
    number++;
}
```

```
// Function declaration
void greet(string name);

// Defining a namespace
namespace messaging {
    void greet(string name) {}
}

// Using a namespace
using namespace messaging;
// or
using messaging::greet;
```

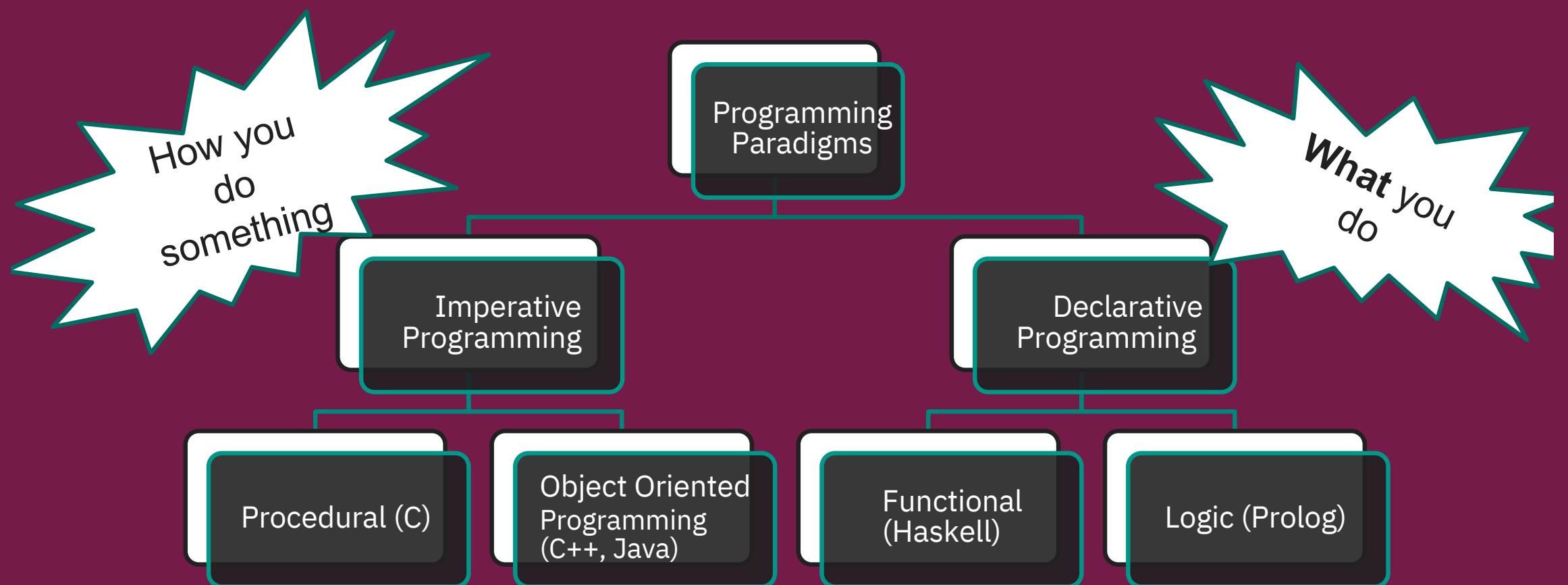
OOPS in C++



AnkitPangasa



PROGRAMMING PARADIGM

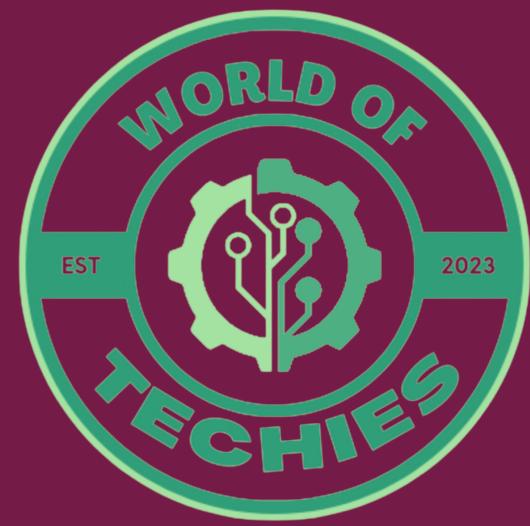
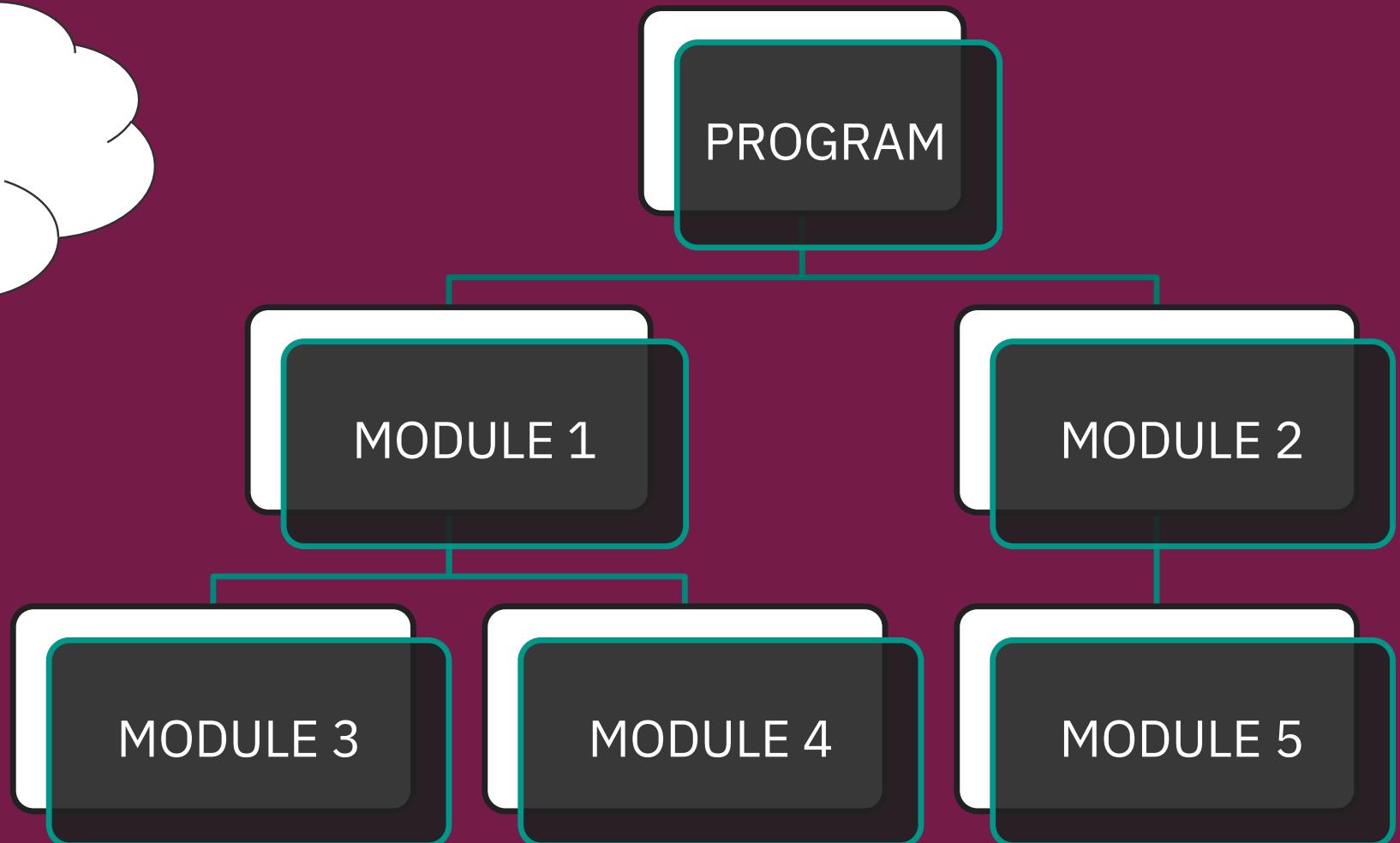


AnkitPangasa



STRUCTURED/ PROCEDURAL LANGUAGE

**TOP-DOWN
APPROACH**



AnkitPangasa

SWIPE

EXAMPLES OF STRUCTURED LANGUAGE

COBOL

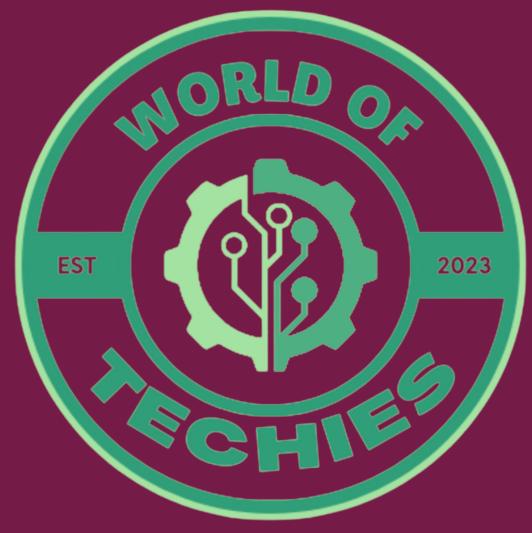
BASIC

C

PASCAL

ALGOL

FORTRAN



AnkitPangasa

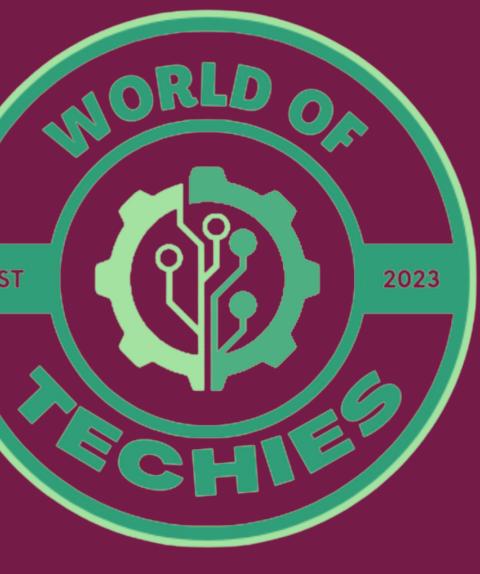
SWIPE

BOTTOM-UP
APPROACH

WHAT IS

OOP

OOP stands for Object-Oriented Programming



AnkitPangasa

SWIPE

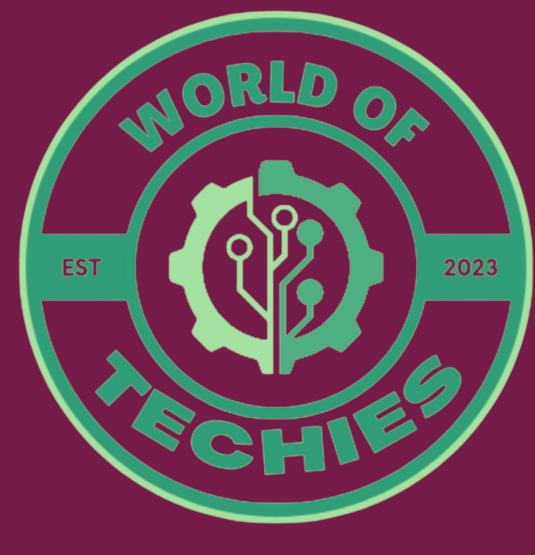
Features of OOPS

Encapsulation

Abstraction

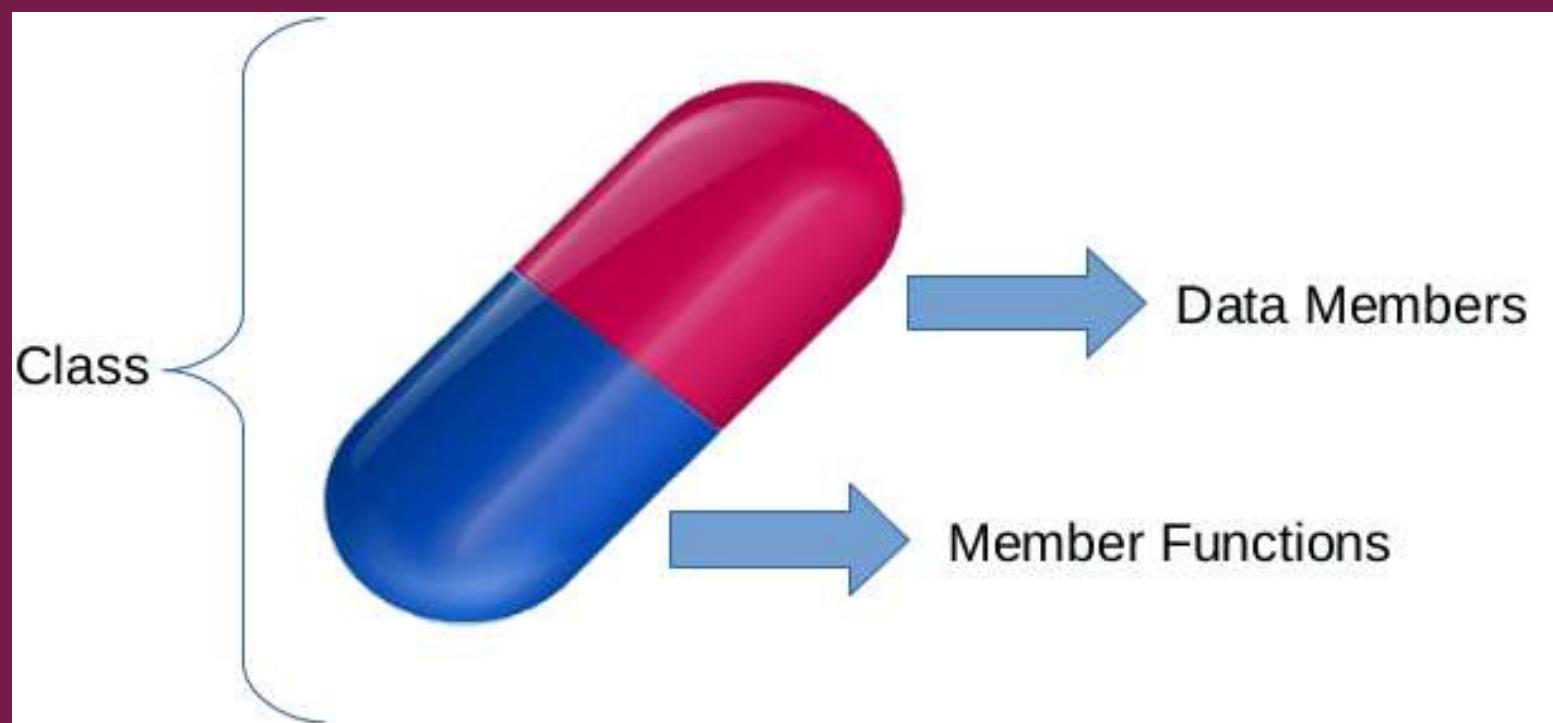
Inheritance

Polymorphism



1. ENCAPSULATION

It is a process of wrapping of data and methods in a single unit



AnkitPangasa

< SWIPE

CLASS and OBJECT

CLASS

Class is a container which collects variables and methods

It is mere a blueprint to which no memory is allocated at the time of declaration

One class definition should exist only once in the program

OBJECT

Object is an instance of a class

Sufficient memory space will be allocated for the variables of class at the time of declaration

For one class multiple objects can be created.



AnkitPangasa





ELEPHANT

SHEEP

TIGER

GIRAFFE

DOG

CLASS: ANIMAL



EST

2023

AnkitPangasa





PINE APPLE



LEMON



APPLE



BANNANA



GRAPES

CLASS: FRUIT



AnkitPangasa

SWIPE

SYNTAX:

CLASS

```
class class_name  
{  
access specifier:  
    data members;  
    member funtions()  
};
```

OBJECT

```
class_name object_reference;
```



```

<header files>
class addition
{ private:
int a, b, sum;
public:
void input()
{
cout<< "Enter two numbers";
cin>>a>>b;
}
void add()
{
sum=a+b;
}
void output()
{
cout<< "Sum of two number is= "<<sum;
}
};
void main()
{
addition ob;
ob. input();
ob. add();
ob.output();
getch();
}

```

Data members

Input() add() output() are the member functions

OUTPUT

Enter two numbers

10

20

Sum of two number is= 30

Object initialization

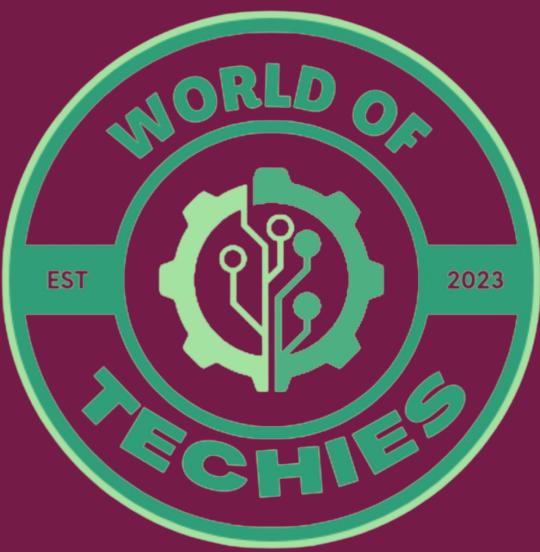
Dot operator is used for calling



```
<header files>
class addition
{
int a, b, sum;
public:
void add()
{
cout<< "Enter two numbers";
cin>>a>>b;
sum=a+b;
cout<< "Sum of two number is= "<<sum;
}
};
void main()
{
clrscr();
addition ob;
ob. add();
getch();
}
```

OUTPUT

```
Enter two numbers
10
20
Sum of two number is= 30
```



AnkitPangasa

SWIPE

2. Abstraction

Abstraction means displaying only essential information and hiding the details



AnkitPangasa



Using Access Specifier in Class

Public:

Accessible both inside and outside the class

Private:

Accessible inside the class but not outside the class

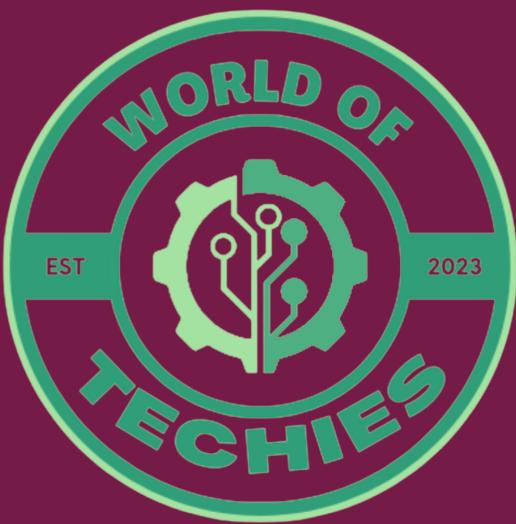
Protected:

accessible inside the class and only in inherited classes

//Error; a and b not accessible outside the class because they are declared private

```
<header files>
class addition
{
int a, b, sum;
public:
void add()
{
cout<< "Enter two numbers";
cin>>a>>b;
sum=a+b;
cout<< "Sum of two number is="
<<sum;
}
};

void main()
{
addition ob;
ob. a= 10;
ob. b=20;
ob. add();
getch();
}
```



3. INHERITANCE

It is a mechanism in which one class acquires the property of another class.

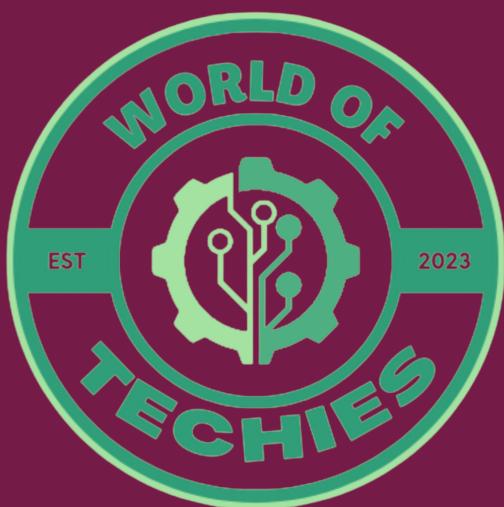


Derived Class (child)

- The class that inherits from another class

Base Class (Parent)

- The class being inherited from



AnkitPangasa



Maths teacher

Talk()
Walk()
TeachMaths()

Footballer

Talk()
Walk()
PlayFootball()

Businessman

Talk()
Walk()
RunBusiness()

Person

Talk()
Walk()
Eat()

TeachMaths()

Maths teacher

PlayFootball()

Footballer

RunBusiness()

Businessman

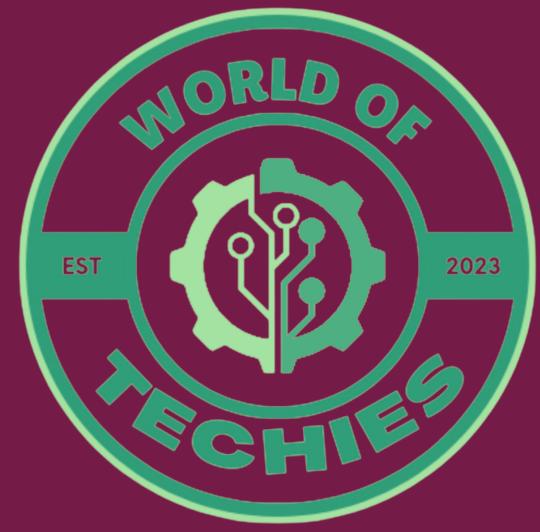
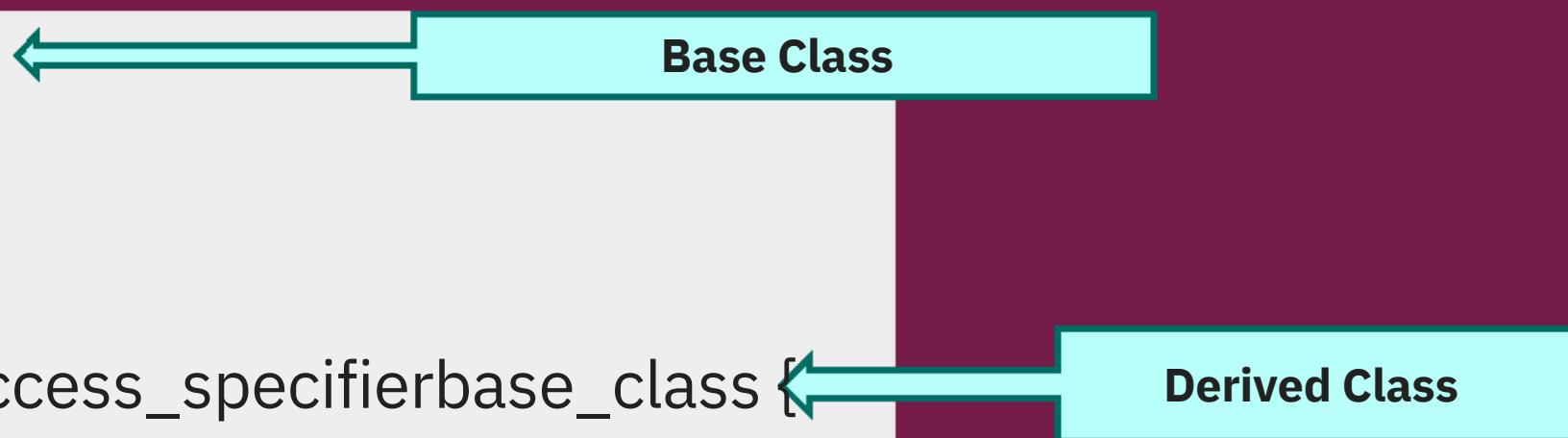


AnkitPangasa

SWIPE

Syntax:

```
class base_class  
{  
//Body of base class  
};  
class derived_class: accessSpecifierbase_class  
//Body of derived class  
};
```



4. Polymorphism

Polymorphism means "one interface, multiple forms"



AnkitPangasa

SWIPE

```

classAnimal {
public:
void animalSound() {
    cout<<"The animal makes a sound \n";
}
};

classPig :publicAnimal {
public:
void animalSound() {
cout<<"The pig says: wee wee\n";
}
};

classDog:publicAnimal {
public:
void animalSound() {
    cout<<"The dog says: bow wow \n";
}
};

void main() {
Animal myAnimal;
Pig myPig;
Dog myDog;
myAnimal.animalSound();
myPig.animalSound();
myDog.animalSound();
}

```

Base Class (Animal)

Derived Class (Pig)

Derived Class (Dog)

OUTPUT

The animal makes a sound
The pig says: wee wee
The dog says: bow wow



```

classAnimal {
public:
void animalSound() {
    cout<<"The animal makes a sound \n";
}
};

classPig :publicAnimal {
public:
void animalSound() {
cout<<"The pig says: wee wee\n";
}
};

classDog:publicAnimal {
public:
void animalSound() {
    cout<<"The dog says: bow wow \n";
}
};

void main() {
Animal myAnimal;
Pig myPig;
Dog myDog;
myAnimal.animalSound();
myPig.animalSound();
myDog.animalSound();
}

```

Base Class (Animal)

Derived Class (Pig)

Derived Class (Dog)

OUTPUT

The animal makes a sound
The pig says: wee wee
The dog says: bow wow

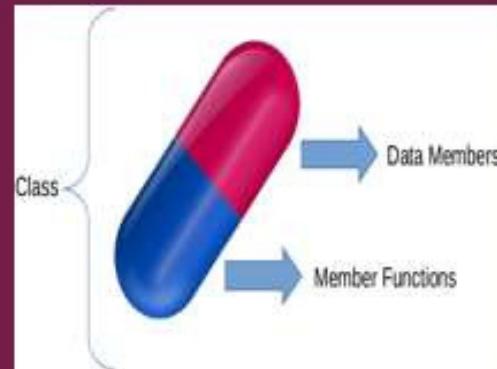


Conclusion



Programming Paradigms

Structured Programming



AnkitPangasa

SWIPE

LOVE THE
CONTENT



AnkitPangasa