OOPC-01-Introduction to OOP

- OOP refers to languages that use objects in programming.
- OOP aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.
- A common feature of objects is that methods are attached to them and can access and modify the object's data fields
- C++, Java, Dot Net, Python etc are the example of OOP.

POP vs OOP

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
Emphasis is on doing things not on data, means it is function driven	Emphasis is on data rather than procedure, means object driven
Main focus is on the function and procedures that operate on data	Main focus is on the data that is being operated
Top Down approach in program design	Bottom Up approach in program design
Large programs are divided into rams known as functions	Large programs are divided into classes and objects
Most of the functions share global data	Data is structured together with function in the data.
Data moves openly in the system from one function to another function	Data is hidden and cannot be accessed by external functions
Adding of data and function is difficult	Adding of data and function is easy
We cannot declare namespace directly	We can use namespace directly, Ex: using namespace std;
Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifiers are not available.	Concepts like inheritance, polymorphism, data encapsulation, abstraction, access specifiers are available and can be used easily
Examples: C, Fortran, Pascal, etc	Examples: C++, Java, C#, etc

POP vs OOP (short)

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
Structure oriented	Object oriented
Program is divided into functions.	Program is divided into objects.
Top-down approach	Bottom-up approach
Inheritance is not allowed.	Inheritance property is used.
It doesn’t use access specifier.	It uses access specifier.
No data hiding.	Encapsulation is used to hide the data.
No virtual function	Concept of virtual function

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
Parts of program are linked through parameter passing.	Object functions are linked through message passing
Expanding new data and functions is not easy	Adding new data and functions is easy
Not suitable for solving big problems.	Used for solving big problems.
C, Pascal	C++, Java

Basic Concept of OOP

- OOP is a type of programming which uses objects and classes its functioning.
- The OOP is based on real world entities like inheritance, polymorphism, data hiding, etc. It aims at binding together data and function work on these data sets into a single entity to restrict their usage.
- Some basic concepts of object oriented programming are:
  - Class
  - Object
  - Encapsulation
  - Polymorphism
  - Inheritance
  - Abstraction

C++ Classes and Objects

- Everything in C++ is associated with classes and objects, along with its attributes and methods.
- For example: in real life, a car is an object. The car has attributes, such as weight and colour, and methods, such as drive and brake.
- Attributes and methods are basically variables and functions that belongs to the class. These are often referred to as "class members".
- A class is a user-defined data type that we can use in our program, and it works as an object constructor, or a "blueprint" for creating objects.
- In C++, an object is created from a class. We have already created the class named MyClass, so now we can use this to create objects.
- To create an object of MyClass, specify the class name, followed by the object name.
- To access the class attributes (myNum and myString), use the dot syntax (.) on the object

```
class MyClass { // The class
  public: // Access specifier
    int myNum; // Attribute (int variable)
  string myString; // Attribute (string variable)
};

int main() {
  MyClass myObj; // Create an object of MyClass

  // Access attributes and set values
```

```
  myObj.myNum = 15;
  myObj.myString = "Some text";

  // Print attribute values
  cout << myObj.myNum << "\n";
  cout << myObj.myString;

  return 0;
}
```

Data Abstraction

- Just represent essential features without including the background details.
- They encapsulate all the essential properties of the objects that are to be created.
- The attributes are sometimes called ‘Data members’ because they hold information.
- The functions that operate on these data are sometimes called ‘methods’ or ‘member functions’.
- It is used to implement in class to provide data security.

Encapsulation

- Wrapping up of a data and functions into single unit is known as encapsulation.
- In C++ the data is not accessible to the outside world.
- Only those functions can access it which is wrapped together within single unit.

Inheritance

- Inheritance is the process, by which class can acquire the properties and methods of another class.
- The mechanism of deriving a new class from an old class is called inheritance.
- The new class is called derived class and old class is called base class.
- The derived class may have all the features of the base class.
- Programmer can add new features to the derived class.
- For example, Student is a base class and Result is derived class.

Polymorphism

- A Greek word Polymorphism means the ability to take more than one form.
- Polymorphism allows a single name to be used for more than one related purpose.
- The concept of polymorphism is characterized by the idea of ‘one interface, multiple methods’
- That means using a generic interface for a group of related activities.
- The advantage of polymorphism is that it helps to reduce complexity by allowing one interface to specify a general class of action’.
- It is the compiler’s job to select the specific action as it applies to each situation.
- It means ability of operators and functions to act differently in different situations.

```
int total(int, int);
int total(int, int, float);
```

Binding

Static Binding

- Static Binding defines the properties of the variables at compile time. Therefore they can't be changed.

Dynamic Binding

- Dynamic Binding means linking of procedure call to the code to be executed in response to the call.
- It is also known as late binding, because It will not bind the code until the time of call at run time.
- In other words properties of the variables are determined at runtimes.
- It is associated with polymorphism and inheritance.

Message Passing

- A program contains set of object that communicates with each other.
- Basic steps to communicate:
  - Creating classes that define objects and their behaviour.
  - Creating objects from class definition
  - Establishing communication among objects.
- Message passing involves the object name, function name and the information to be sent.
- Example: `employee.salary(name);`
- In above statement employee is an object. salary is message, and name is information to be sent.

Benefit of OOP

- We can eliminate redundant code though inheritance.
- Saving development time and cost by using existing module.
- Build secure program by data hiding.
- It is easy to partition the work in a project based on object.
- Data centered design approach captures more details of a programming model.
- It can be easily upgraded from small to large system.
- It is easy to map objects in the problem domain to those in the program.
- Through message passing interface between objects makes simpler description with external system.
- Software complexity can be easily managed.

Application of OOP

- Real-time systems

- Simulation and modeling
- Object oriented database
- Artificial intelligence and expert systems
- Neural networks and parallel programming
- Decision support and office automation
- CIM/CAM/CAD systems.
- Distributed/Parallel/Heterogeneous computing
- Data warehouse and Data mining/Web mining

Basic Structure of C++

A C++ program may have one or more sections but the sequence of sections is to be followed.

- Documentation section
- Linking section
- Definition section
- Global declaration section
- Class declaration Section
- Main function section

Documentation Section:

- It is used to document the use of logic or reasons in your program
- It can be used to write the program's objective, developer and logic details.
- Whatever is written between these two are called comments.

Linking Section:

- This section tells the compiler to link the certain occurrences of keywords or functions in your program to the header files specified in this section.
- e.g. `#include<iostream> using namespace std;`
- Directive causes the pre-processor to add the contents of the iostream file to the program. It contains declarations for `cout` and `cin`.

Definition Section:

- It is used to declare some constants and assign them some value.
- e.g. `#define MAX 25;`
- Here `#define` is a compiler directive which tells the compiler whenever MAX is found in the program replace it with 25.

Global Declaration Section:

- Here the variables and class definitions which are used throughout the program (including main and other functions) are declared so as to make them global (i.e. accessible to all parts of program).

Class Declaration Section

- A CLASS is a collection of data and functions that act or manipulate the data.

- The data components of a class are called data members and function components of a class are called member functions.

main() Function Section:

- It tells the compiler where to start the execution from main()
- Main function has two sections
  - Declaration section : In this the variables and their data types are declared.
  - Executable section or instruction section : This has the part of program which actually performs the task we need.

```
// Comments

#include <iostream>  // Preprocessor Directive
using namespace std;

// Global Declaration
int globalVariable = 10;

// Function Declaration/Prototype
void myFunction();

// Class Declaration
class MyClass {
public:
    void classFunction();
};

// main() Function: Entry point of the program
int main() {
    cout << "Hello, World!" << endl;  // Code statement

    myFunction();  // Function call

    MyClass myObject;  // Object creation
    myObject.classFunction();  // Member function call

    return 0;  // Return statement
}

// Function Definition
void myFunction() {
    cout << "This is a function." << endl;
}

// Member function definition
void MyClass::classFunction() {
    cout << "This is a member function." << endl;
}
```

Keywords

- They are explicitly reserved identifiers and cannot be used as names for the program variables or other user- defined program elements.
- Ex: int, class, void etc

Identifiers

- They refer to the names of variables, functions, arrays, classes etc., created by the programmer

Variables

- Variables in C++ is a name given to a memory location. It is the basic unit of storage in a program.
- The value stored in a variable can be changed during program execution
- A variable is only a name given to a memory location, all the operations done on the variable affects that memory location.

Reference Variable

- When a variable is declared as a reference, it becomes an alternative name for an existing variable.
- A variable can be declared as a reference by putting '&' in the declaration.
- We can define a reference variable as a type of variable that can act as a reference to another variable.
- '&' is used for signifying the address of a variable or any memory.
- Variables associated with reference variables can be accessed either by its name or by the reference variable associated with it.
- Syntax: `data_type &ref = variable;`

```
int main()
{
    int x = 10;

    // ref is a reference to x.
    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << '\n';

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}
```

Constants

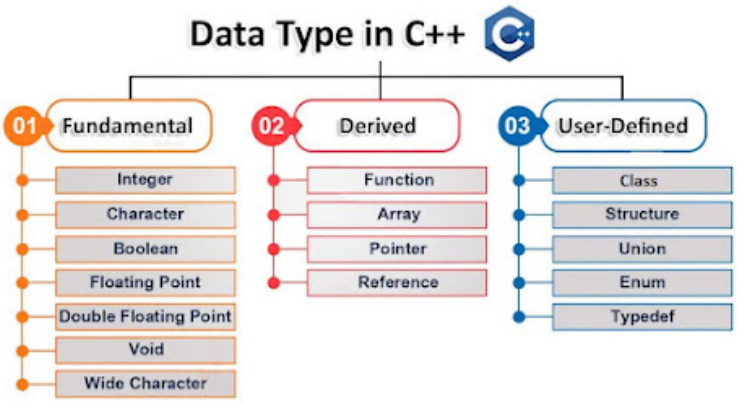
- Constants are data storage locations. But variables can vary, constants do not change.

- You must initialize a constant when you create it, and you can not assign new value later, after constant is initialized.
- The define pre-processor directive and the const keyword are the two methods to define a constant.

Tokens

- A C++ program is composed of tokens which are the smallest individual unit.
- Every word in a C++ source code can be considered a token.
- There are several types of tokens each of which serves a specific purpose in the syntax and semantics of C++.
- 6 types of Tokens:
  - **Keyword**
  - **Identifiers**
  - **Constants**
  - **Strings**
  - **Special symbols**
  - **Operators**

Data Types



Primitive (Fundamental) Data Types:

- These data types are built-in or predefined data types and can be used directly by the user to declare variables.

Derived Data Types:

- They are derived from the primitive or built-in datatypes referred to as Derived Data Types.

User-Defined (Abstract) Data Types:

- They are defined by the user itself

Types Casting

- Type casting refers to the conversion of one data type to another in a program.

- Typecasting can be done in two ways: automatically by the compiler and manually by the programmer or user.
- Type Casting is also known as Type Conversion.

Implicit Types Casting(Automatic)

- It automatically converted from one data type to another without any external intervention such as programmer or user. It means the compiler automatically converts one data type to another.
- Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data.
- All data type is automatically upgraded to the largest type without losing any information.
- It can only apply in a program if both variables are compatible with each other.
  `bool > char > short int > int > unsigned int > long > unsigned > long long > float > double > long double`

```
int x = 10; // integer x
char y = 'a'; // character c

// y implicitly converted to int. ASCII
// value of 'a' is 97

x = x + y;
```

Explicit (Manual) Type Casting:

- It is manually cast by the programmer or user to change from one data type to another type in a program.
- It means a user can easily cast one data to another according to the requirement in a program.
- It does not require checking the compatibility of the variables.
- In this casting, we can upgrade or downgrade the data type of one variable to another in a program.
- **Converting by Assignment:**
  - This is done by explicitly defining the required type in front of the expression in parenthesis.
  - This can be also considered as forceful casting.

```
double x = 1.2;
// Explicit conversion from double to int
int sum = (int)x + 1;
```

- **Converting using Cast Operator:**
  - A Cast operator is an unary operator which forces one data type to be converted into another data type.

Enumeration

- Enumeration (Enumerated type) is a user-defined data type that can be assigned some limited values.
- These values are defined by the programmer at the time of declaring the enumerated type.
- The C++ enum constants are static and final implicitly.
- C++ Enums can be thought of as classes that have fixed set of constants.
- After defining Enumerated type variables are created. Enumerators can be created in two types:
  - It can be declared during declaring enumerated types, just add the name of the variable before the semicolon
  - We can create enumerated type variables as the same as the normal variables

```
#include<iostream>
using namespace std;

enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };

int main()
{
    week day;
    day = Friday;
    cout << "Day: " << day+1<<endl;
    return 0;
}

// Output: Day: 5
```

Control Structure

- Control Structures are just a way to specify flow of control in programs.
- Any algorithm or program can be more clear and understood if they use self-contained modules called as logic or control structures.
- It basically analyses and chooses in which direction a program flows based on certain parameters or conditions.

1. **Sequential Logic (Sequential Flow) :**
   - Sequential logic as the name suggests follows a serial or sequential flow in which the flow depends on the series of instructions given to the computer.
   - Unless new instructions are given, the modules are executed in the obvious sequence.
   - The sequences may be given, by means of numbered steps explicitly.
   - Also, implicitly follows the order in which modules are written.
   - Most of the processing, even some complex problems, will generally follow this elementary flow pattern.

> **Note**

This is the most simple and basic form of a control structure. It is simply the plain logic we write; it only has simple linear instructions, no decision making, and no loop

2. **Selection Logic (Conditional Flow):**
   - Selection Logic simply involves a number of conditions or parameters which decides one out of several written modules.
   - The structures which use these type of logic are known as Conditional Structures

> **Note**
> Sometimes in our program, we will need to write certain condition checks that this part of code should only execute if a particular condition meets or another part of code should run.

3. **Iteration Logic (Repetitive Flow):**
   - The Iteration logic employs a loop which involves a repeat statement followed by a module known as the body of a loop.

> **Note**
> Whenever in our program we see that a certain piece of code is being repeated repeatedly, then we can enclose that in a loop structure and provide the condition that this code should execute these specific number of times.

Control Statement

- Control statements are used to evaluate one or more conditions and make the decision whether to execute a set of statements or not.
- These decision-making statements in programming languages decide the direction of the flow of program execution.

Conditional Statement

- Conditional Statements are used in C++ to run a certain piece of program only if a specific condition is met
- **if** Statement:
  - It is used to run a certain piece of code only if a certain condition is true

```
int x = 5;
if (x == 5){
    std::cout << "x is 5" << std::endl;
}
```

- **if-else** Statement:
  - The if-else statement is used when we want to execute some code only if some condition exists.
  - If the given condition is true then code will be executed otherwise else statement will be used to run other part of the code.

```cpp
int x = 5;
if (x == 5){
    std::cout << "x is 5" << std::endl;
}else{
    std::cout << "x is not 5" << std::endl;
}
```

- **switch** Statement
    - The switch statement is used to execute different blocks of code based on the value of a variable.

```cpp
int x = 2;
switch (x){
    case 1:
        std::cout << "x is 1" << std::endl;
        break;
    case 2:
        std::cout << "x is 2" << std::endl;
        break;
    default:
        std::cout << "x is not 1 or 2" << std::endl;
        break;
}
```

Loops

- A loop statement allows us to execute a statement or group of statements multiple times
- Two Types:
    - **Entry Controlled:** In this type of loop, the test condition is tested before entering the loop body. For Loop and While Loop is entry-controlled loops.
    - **Exit Controlled:** In this type of loop the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false.
- **while Loop:**
    - The while loop is used to execute when we want to run some code for until some specific condition matches.
    - Entry Controlled loop

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

- **do-while Loop:**
    - The do-while loop is the same as the while loop, but the condition is checked after the first iteration of the loop.

- Exit controlled loop

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

- **for Loop:**
  - The for loop allows a program to execute a piece of program a fixed number of times.
  - Entry controlled loop

```
for (int i = 1; i ⩽ 10; i++){
    cout << i << " ";
}
```

break Statement

- The break statement can be used to jump out of a loop.

```
for (int i = 0; i < 10; i++){
    if (i == 4){
        break;
    }
    cout << i << "\n";
}
```

continue Statement:

- The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.

```
for (int i = 0; i < 10; i++){
    if (i == 4){
        continue;
    }
    cout << i << "\n";
```

Types of Error

Syntax Error:

- These are also referred to as compile-time errors. These errors have occurred when the rule of C++ writing techniques or syntax has been broken.
- These types of errors are typically flagged by the compiler prior to compilation.

Runtime Error:

- This type of error occurs while the program is running.
- Because this is not a compilation error, the compilation will be completed successfully.
- These errors occur due to segmentation fault when a number is divided by division operator or modulo division operator.

Logical Errors:

- Even if the syntax and other factors are correct, we may not get the desired results due to logical issues.
- These are referred to as logical errors.
- We sometimes put a semicolon after a loop, which is syntactically correct but results in one blank loop.
- In that case, it will display the desired output.

Linker Errors:

- When the program is successfully compiled and attempting to link the different object files with the main object file, errors will occur. When this error occurs, the executable is not generated.
- This could be due to incorrect function prototyping, an incorrect header file, or other factors.
- If main() is written as Main(), a linked error will be generated.

Semantic Errors:

- When a sentence is syntactically correct but has no meaning, semantic errors occur.
- This is similar to grammatical errors.
- If an expression is entered on the left side of the assignment operator, a semantic error may occur.

Array

- An array is a data structure that is used to store multiple values of similar data types in a contiguous memory location

Properties of an Array

- Indexing of an array starts from 0. It means the first element is stored at the 0th index, the second at 1st, and so on.
- Elements of an array can be accessed using their indices.
- Once an array is declared its size remains constant throughout the program.
- An array can have multiple dimensions.
- The number of elements in an array can be determined using the sizeof operator.
- We can find the size of the type of elements stored in an array by subtracting adjacent addresses.

Declaration of an Array

- We can declare an array by simply specifying the data type first and then the name of an array with its size.

- Syntax: `data_type array_name[Size_of_array];`
- `string cars[4];`
- To insert values to it, we can use an array literal - place the values in a comma-separated list, inside curly braces
- `string cars[4] = {"Volvo", "BMW", "Ford", "Mazda"};`
- To access the values of an Array
- `cout << cars[0]; // Outputs Volvo`

```
int main(){
    // Initialize the array
    int table_of_two[10] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

    // Traverse the array using for loop
    for (int i = 0; i < 10; i++){
        // Print the array elements using indexing
        cout << table_of_two[i] << " ";
     }
    return 0;
}
```

Multi-Dimensional Array

- Arrays declared with more than one dimension are called multidimensional arrays.
- The most widely used multidimensional arrays are 2D arrays and 3D arrays.
- These arrays are generally represented in the form of rows and columns.
- Multidimensional Array Declaration : `Data_Type Array_Name[Size1][Size2]...[SizeN];`

2D Array

- A two-dimensional array is a grouping of elements arranged in rows and columns.
- Each element is accessed using two indices: one for the row and one for the column, which makes it easy to visualize as a table or grid
-

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

```
//Program to Declare, Initialise and print a 2D array

int main(){
    // Declaring 2D array
    int arr[4][4];

    // Initialize 2D array using loop
    for (int i = 0; i < 4; i++){
        for (int j = 0; j < 4; j++) {
            arr[i][j] = i + j;
        }
```

```
    }

    // Printing the element of 2D array
    for (int i = 0; i < 4; i++){
        for (int j = 0; j < 4; j++){
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Classes and Objects

- **Class** in C++ is the building block that leads to Object-Oriented programming.
- It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object
- Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behaviour of the objects in a Class
- *For Example: Consider the Class of Cars. There may be many cars with different names and brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc. So here, Car is the class, and wheels, speed limits, and mileage are their properties.*
- *In the above example of class Car, the data member will be speed limit, mileage, etc, and member functions can be applying brakes, increasing speed, etc.*
- An **Object** is an instance of a Class.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class

- A class is defined using the keyword class followed by the name of the class.
- The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class ClassNam{
    Access specifier:       //can be private,public or protected
    Data members;           // Variables to be used
    Member Functions() {}   //Methods to access data members
};                          // Class name ends with a semicolon
```

Declaring Object

- When a class is defined, only the specification for the object is defined; no memory or storage is allocated.
- To use the data and access functions defined in the class, you need to create objects
- Syntax: `ClassName ObjectName;`

Accessing Data Members and Member Functions

- The data members and member functions of the class can be accessed using the dot('.') operator with the object.
- For example, if the name of the object is `obj` and you want to access the member function with the name `printName()` then you will have to write `obj.printName()`.

Accessing Data Members

- The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object.
- Accessing a data member depends solely on the access control of that data member.
- This access control is given by access modifier in C++.
- There are **three access specifiers**:
  - **Public**: members are accessible from outside the class
  - **Private**: members cannot be accessed (or viewed) from outside the class
  - **Protected**: members cannot be accessed from outside the class, however, they can be accessed in inherited classes

Scope of Variables

- The scope of a variable is defined as the extent of the program code within which the variable can be accessed or declared or worked with

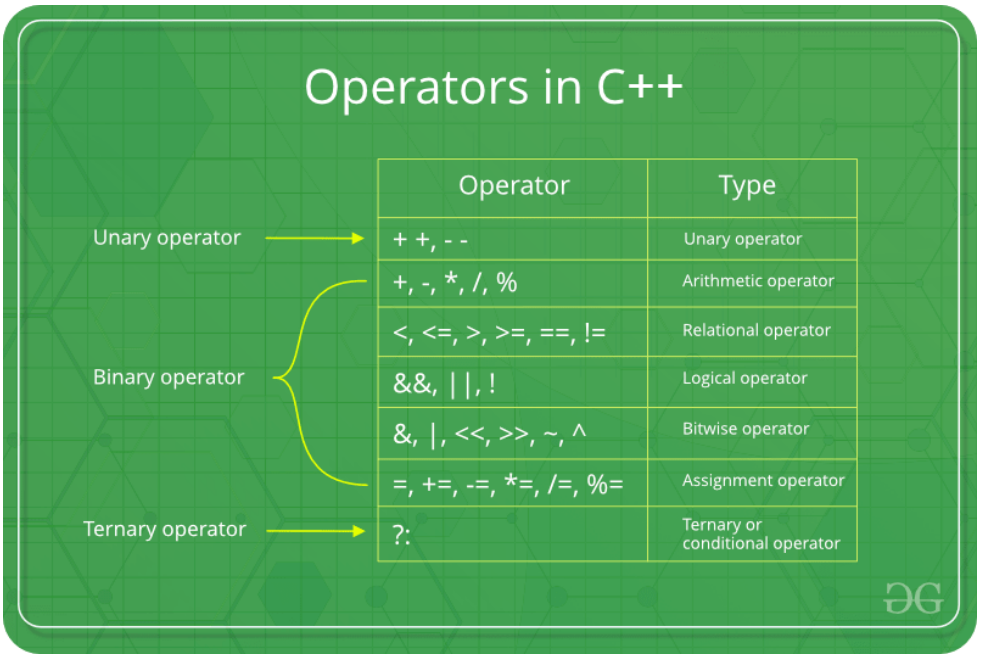
1. **Local Variable**
   - Variables defined within a function or block are said to be local to those functions.
   - Anything between '{' '}' is said to be inside a block.
   - Local variables do not exist outside the block in which they are declared
   - Declaring local variables: Local variables are declared inside a block.
2. **Global Variables:**

```
#include<iostream>
using namespace std;

//Global Variable
int global = 5;

int main(){
    //Local variable
    int local = 2;
    cout<<global<<endl;
    cout<<local<<endl;
}
```

Operators in C++



Unary Operator:

- These operators operate or work with a single operand.

Name	Symbol	Description	Example
Increment Operator	++	Increases the integer value of the variable by one	int a = 5; a++; //returns 6
Decrement Operator	--	Decreases the integer value of the variable by one	int a = 5; a--; //returns 4

> ⚠ **Caution**
>
> **++a** and **a++**, both are increment operators, however, both are slightly different.
> In **++a**, the value of the variable is incremented first and then It is used in the program.
> In **a++**, the value of the variable is assigned first and then It is incremented.
> Similarly happens for the decrement operator.

Binary Operators

1. **Arithmetic Operators**

- Are used to perform arithmetic or mathematical operations on the operands

Name	Symbol	Description	Example

Addition	+	Adds two operands	int a = 3, b = 6;
			
			int c = a+b; // c = 9

Subtraction	-	Subtracts second operand from the first	int a = 9, b = 6;
			
			int c = a-b; // c = 3

Multiplication	*	Multiplies two operands	int a = 3, b = 6;

Name	Symbol	Description	Example
			int c = a*b; // c = 18

Division	/	Divides first operand by the second operand	int a = 12, b = 6;
			
			int c = a/b; // c = 2

Modulo Operation	%	Returns the remainder an integer division	int a = 8, b = 6;
			
			int c = a%b; // c = 2

```cpp
// CPP Program to demonstrate the Binary Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 8, b = 3;

    // Addition operator
    cout << "a + b = " << (a + b) << endl;

    // Subtraction operator
    cout << "a - b = " << (a - b) << endl;

    // Multiplication operator
    cout << "a * b = " << (a * b) << endl;

    // Division operator
    cout << "a / b = " << (a / b) << endl;

    // Modulo operator
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

2. **Relational Operators:**

- These operators are used for the comparison of the values of two operands.
- The result returns a Boolean value, i.e., **true** or **false.**

Name	Symbol	Description	Example

Is Equal To	==	Checks if both operands are equal	int a = 3, b = 6;
			
			a==b;
			
			// returns false

Name	Symbol	Description	Example

Greater Than	>	Checks if first operand is greater than the second operand	int a = 3, b = 6;
			
			a>b;
			
			// returns false

Greater Than or Equal To	>=	Checks if first operand is greater than or equal to the second operand	int a = 3, b = 6;
			
			a>=b;
			
			// returns false

Less Than	<	Checks if first operand is lesser than the second operand	int a = 3, b = 6;
			
			a<b;
			
			// returns true

Less Than or Equal To	<=	Checks if first operand is lesser than or equal to the second operand	int a = 3, b = 6;
			
			a<=b;
			
			// returns true

Not Equal To	!=	Checks if both operands are not equal	int a = 3, b = 6;
			
			a!=b;
			
			// returns true

```cpp
// CPP Program to demonstrate the Relational Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;

    // Equal to operator
    cout << "a == b is " << (a == b) << endl;

    // Greater than operator
    cout << "a > b is " << (a > b) << endl;

    // Greater than or Equal to operator
```

```
    cout << "a ⩾ b is " << (a ⩾ b) << endl;

    // Lesser than operator
    cout << "a < b is " << (a < b) << endl;

    // Lesser than or Equal to operator
    cout << "a ⩽ b is " << (a ⩽ b) << endl;

    // true
    cout << "a ≠ b is " << (a ≠ b) << endl;

    return 0;
}
```

3. **Logical Operators:**

- These operators are used to combine two or more conditions or constraints or to complement the evaluation of the original condition in consideration.
- The result returns a Boolean value, i.e., **true** or **false**.

Name	Symbol	Description	Example

Logical AND	&&	Returns true only if all the operands are true or non-zero	int a = 3, b = 6;
			
			a&&b;
			
			// returns true

Logical OR	|	Returns true if either of the operands is true or non-zero	int a = 3, b = 6;
			
			a|b;
			
			// returns true

Logical NOT	!	Returns true if the operand is false or zero	int a = 3;
			
			!a;
			
			// returns false

```
// CPP Program to demonstrate the Logical Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;

    // Logical AND operator
    cout << "a && b is " << (a && b) << endl;

    // Logical OR operator
    cout << "a ! b is " << (a > b) << endl;
```

```
    // Logical NOT operator
    cout << "!b is " << (!b) << endl;

    return 0;
}
```

4. **Bitwise Operators:**

- These operators are used to perform bit-level operations on the operands.
- The operators are first converted to bit-level and then the calculation is performed on the operands.
- Mathematical operations such as addition, subtraction, multiplication, etc. can be performed at the bit level for faster processing.

Name	Symbol	Description	Example

Binary **AND**	&	Copies a bit to the evaluated result if it exists in both operands	int a = 2, b = 3;
			
			(a & b);
			//returns 2

Binary **OR**	|	Copies a bit to the evaluated result if it exists in any of the operand	int a = 2, b = 3;
			
			(a | b);
			//returns 3

Binary **XOR**	^	Copies the bit to the evaluated result if it is present in either of the operands but not both	int a = 2, b = 3;
			
			(a ^ b);
			//returns 1

Left Shift	<<	Shifts the value to left by the number of bits specified by the right operand.	int a = 2, b = 3;
			
			(a << 1);
			//returns 4

Right Shift	>>	Shifts the value to right by the number of bits specified by the right operand.	int a = 2, b = 3;
			
			(a >> 1);
			//returns 1

One's **Complement**	~	Changes binary digits 1 to 0 and 0 to 1	int b = 3;
			
			(~b);
			//returns -4

```
// CPP Program to demonstrate the Bitwise Operators
#include <iostream>
using namespace std;

int main()
{
```

```
    int a = 6, b = 4;

    // Binary AND operator
    cout << "a & b is " << (a & b) << endl;

    // Binary OR operator
    cout << "a | b is " << (a | b) << endl;

    // Binary XOR operator
    cout << "a ^ b is " << (a ^ b) << endl;

    // Left Shift operator
    cout << "a<<1 is " << (a << 1) << endl;

    // Right Shift operator
    cout << "a>>1 is " << (a >> 1) << endl;

    // One's Complement operator
    cout << "~(a) is " << ~(a) << endl;

    return 0;
}
```

> ⚠ **Caution**
>
> Only **char** and **int** data types can be used with Bitwise Operators.

5. **Assignment Operators:**

- These operators are used to assign value to a variable.
- The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value

Name	Symbol	Description	Example

Assignment Operator	=	Assigns the value on the right to the variable on the left	int a = 2;
			
			// a = 2

Add and Assignment Operator	+=	First adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left	int a = 2, b = 4;
			
			a+=b; // a = 6

Subtract and Assignment Operator	-=	First subtracts the value on the right from the current value of the variable on left and then assign the result to the variable on the left	int a = 2, b = 4;
			
			a-=b; // a = -2

Multiply and Assignment Operator	*=	First multiplies the current value of the variable on left to the value on the right and then assign the result to the variable on the left	int a = 2, b = 4;

Name	Symbol	Description	Example
			a*=b; // a = 8

Divide and Assignment Operator	/=	First divides the current value of the variable on left by the value on the right and then assign the result to the variable on the left	int a = 4, b = 2;
			
			a /=b; // a = 2

```cpp
// CPP Program to demonstrate the Assignment Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;

    // Assignment Operator
    cout << "a = " << a << endl;

    // Add and Assignment Operator
    cout << "a += b is " << (a += b) << endl;

    // Subtract and Assignment Operator
    cout << "a -= b is " << (a -= b) << endl;

    // Multiply and Assignment Operator
    cout << "a *= b is " << (a *= b) << endl;

    // Divide and Assignment Operator
    cout << "a /= b is " << (a /= b) << endl;

    return 0;
}
```

Ternary Operator

- This operator takes **three operands**, therefore it is known as a Ternary Operator.
- This operator returns the value based on the condition.
- Syntax; `Expression1? Expression2: Expression3`
- The ternary operator **?** determines the answer on the basis of the evaluation of **Expression1**. If it is **true**, then **Expression2** gets evaluated and is used as the answer for the expression. If **Expression1** is **false**, then **Expression3** gets evaluated and is used as the answer for the expression.

```cpp
// CPP Program to demonstrate the Conditional Operators
#include <iostream>
using namespace std;

int main()
```

```
{
    int a = 3, b = 4;

    // Conditional Operator
    int result = (a < b) ? b : a;
    cout << "The greatest number is " << result << endl;

    return 0;
}
```

Scope Resolution Operator

- The scope resolution operator is used to reference the global variable or member function that is out of scope.
- Therefore, we use the scope resolution operator to access the hidden variable or function of a program.
- The operator is represented as the **double colon (::)** symbol.

Memory Management Operator

- Unary operators such as new and delete are used to allocating and freeing the memory.
- A **new** operator is used to create the object while a **delete** operator is used to delete the object.
- When the object is created by using the new operator, then the object will exist until we explicitly use the delete operator to delete the object.
- Therefore, we can say that the lifetime of the object is not related to the block structure of the program.
- Syntax: `pointer_variable = new data-type`

Overloading Stream Insertion (<<) and Extraction (>>) Operators

- C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.
- The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.
- Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object

Manipulator Operators

- **endl**: It is defined in `iostream`. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
- **ws**: It is defined in `iostream` and is used to ignore the whitespaces in the string sequence.
- **ends**: It is also defined in `iostream` and it inserts a null character into the output stream. It typically works with `std::ostrstream`, when the associated output buffer needs to be null-terminated to be processed as a C string
- **flush**: It is also defined in `iostream` and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same,

but may not appear in real-time

Questions

1. Compare and contrast Procedural Vs. Object Oriented Programming. (MID-05M)
2. Explain various principles of OOP. (MID-04M)
3. Describe the basic structure of C++ program. (MID-04M)
4. What is an access specifier ? Explain with an example.
5. Describe scope of variables in C++.
6. Explain various types of operators supported by C++.
7. Explain Class and Object with example (MID-03M)
8. Explain advantages of OOP. (MID-03M)
9. Explain various types of access specifiers supported by C++. (MID-03M)
10. Differentiate between Polymorphism and Inheritance (MID-03M)
11. Explain input output statements used to read and write data in C++ (MID-04M)

Join Us on Telegram @SOU BCA