# C++

**Sessions 1: Getting Started**

**Installation and Setup of Development Environment:**

## Windows:

**1. Choose a Compiler:** C++ requires a compiler to translate your code into machine-readable format. Popular options include GCC (GNU Compiler Collection), Clang, and Microsoft Visual C++ Compiler. Install the compiler of your choice.

**2. Choose an Integrated Development Environment (IDE**): An IDE streamlines development by providing features like code editing, debugging, and project management. Common choices for C++ development include Visual Studio, Visual Studio Code, Code::Blocks, Eclipse CDT, and JetBrains CLion. Download and install your preferred IDE.

**3. Set Up the IDE:** Follow the instructions provided by the IDE to set it up. This typically involves configuring compiler settings and selecting a project directory.

**4. Test Your Setup:** Create a simple "Hello World" program (more on this later) and compile/run it to ensure everything is working correctly.

## Linux:

Linux distributions include GCC(GNU Compiler Collection) as their primary C++ compiler. Many Linux distributions also include CLang as an alternative C++ compiler. The C++ client libraries support both.

To install C++ in a Linux environment, install the appropriate packages for your distribution. For Debian and Ubuntu, this package is g++.

1. Install these packages using the following commands:

```
sudo apt update
sudo apt install g++
```

2. After the installations are complete, verify that you have g++ installed:

```
g++ --version
```

## MacOS:

1. You can get a C++ compiler by installing Xcode's command-line tools.
   ```
   xcode-select --install
   ```
2. After the installation is complete, verify that your compiler is available as c++:
   ```
   c++ --version
   ```

# The need of C++:

**a. Performance:** C++ allows for low-level manipulation and offers performance close to that of assembly language.

**b. Portability:** C++ code can be compiled on various platforms, making it suitable for developing cross-platform applications.

**c. Control**: C++ provides direct memory access and supports object-oriented programming, giving developers precise control over system resources.

# Features of C++:

- **Object-Oriented**: C++ supports the principles of encapsulation, inheritance, and polymorphism, facilitating modular and reusable code.
- **Template Metaprogramming**: C++ allows for generic programming through templates, enabling the creation of flexible and efficient algorithms.
- **Standard Template Library (STL)**: C++ provides a rich set of data structures and algorithms in its standard library, making common tasks easier and more efficient.
- **Machine Independent:** A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however, they are machine-independent.
- **Simple:** It is a simple language in the sense that programs can be broken down into logical units and parts, has rich library support and has a variety of data types.
- **High-Level Language:**
- **Case-sensitive:** It is clear that C++ is a case-sensitive programming language. For example, cin is used to take input from the input stream. But if the "Cin" won't work. Other languages like HTML and MySQL are not case-sensitive languages.
- **Compiler Based:** C++ is a compiler-based language, unlike Python. That is C++ programs used to be compiled and their executable file is used to run them. C++ is a relatively faster language than Java and Python.
- **Dynamic Memory Allocation:** When the program executes in C++ then the variables are allocated the dynamical heap space.
- **Memory Management:** C++ allows us to allocate the memory of a variable or an array in run time. This is known as Dynamic Memory Allocation.
- **Multi-threading:** Multithreading is a specialized form of multitasking and multitasking is a feature that allows your system to execute two or more programs concurrently.

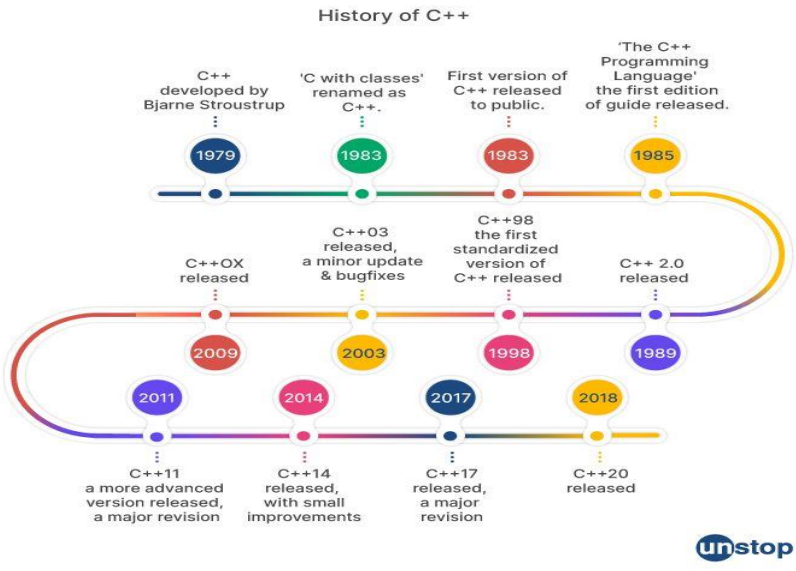
# C++ vs. C

While C and C++ share some similarities, they also have significant differences:

- **Paradigm:** C is a procedural programming language, whereas C++ supports both procedural and object-oriented programming.
- **Abstraction:** C++ provides stronger support for abstraction through features like classes, inheritance, and polymorphism, which are absent in C.
- **Memory Management**: C++ offers features like constructors, destructors, and smart pointers for automatic memory management, reducing the risk of memory leaks compared to C.
- **Standard Library:** C++ includes a more extensive standard library compared to C, with support for data structures, algorithms, input/output operations, and more.
- **Compatibility:** C++ is largely compatible with C code, allowing developers to mix C and C++ code within the same project.
- **Set:** C is a subset (for the most part) of C++ whereas C++ is a superset (for the most part) of C.
- **Friend/ Virtual Functions:** C supports neither virtual nor friend functions. C++ supports both friend functions as well as virtual functions.
- **Allocation of memory:** C uses calloc() and malloc() for dynamic memory allocation. C++ uses new() and free() for dynamic memory allocation.

# History of C ++ :

Let's understand the history of C ++ with a timeline.



C++ was developed by Bjarne Stroustrup at Bell Labs in the early 1980s as an extension of the C programming language. Its development was motivated by the need for a language that supported both low-level system programming and high-level application development. C++ has since evolved through several standardized versions, with the latest being C++20.

# Writing your first C++ program:

```
// C++ program to display "Hello World"

// Header file for input output functions

#include <iostream>

using namespace std;

// Main() function: where the execution of

// program begins

int main()

{

    // Prints hello world

    cout << "Hello World";

    return 0;

}
```

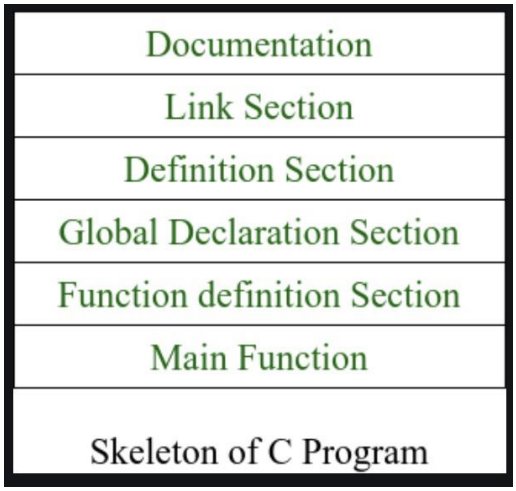
**Output:**

Hello World

# Session 2: Beginning with C++

## C++ Program structure:

The C++ program is written using a specific template structure. The structure of the program written in C++ language is as follows:




### Documentation Section:

- This section comes first and is used to document the logic of the program that the programmer going to code.
- It can be also used to write for purpose of the program.
- Whatever written in the documentation section is the comment and is not compiled by the compiler.
- Documentation Section is optional since the program can execute without them.

## Linking Section:

The linking section contains two parts:

**Header Files:**

- Generally, a program includes various programming elements like built-in functions, classes, keywords, constants, operators, etc. that are already defined in the standard C++ library.
- To use such pre-defined elements in a program, an appropriate header must be included in the program.
- Standard headers are specified in a program through the preprocessor directive #include. In Figure, the iostream header is used. When the compiler processes the instruction #include<iostream>, it includes the contents of the stream in the program. This enables the programmer to use standard input, output, and error facilities that are provided only through the standard streams defined in <iostream>. These standard streams process data as a stream of characters, that is, data is read and displayed in a continuous flow. The standard streams defined in <iostream> are listed here.

**#include<iostream>**

**Namespaces:**

- A namespace permits grouping of various entities like classes, objects, functions, and various C++ tokens, etc. under a single name.
- Any user can create separate namespaces of its own and can use them in any other program.
- In the below snippets, namespace std contains declarations for cout, cin, endl, etc. statements.

```
Using namespace std;
```

- Namespaces can be accessed in multiple ways:
  - using namespace std;
  - using std :: cout;

## Definition Section:

- It is used to declare some constants and assign them some value.
- In this section, anyone can define your own datatype using primitive data types.
- In #define is a compiler directive which tells the compiler whenever the message is found to replace it with "Factorial\n".
- typedef int INTEGER; this statement tells the compiler that whenever you will encounter INTEGER replace it by int and as you have declared INTEGER as datatype you cannot use it as an identifier.

## Global Declaration Section:

- Here, the variables and the class definitions which are going to be used in the program are declared to make them global.
- The scope of the variable declared in this section lasts until the entire program terminates.
- These variables are accessible within the user-defined functions also.

**Function Declaration Section:**

- It contains all the functions which our main functions need.
- Usually, this section contains the User-defined functions.
- This part of the program can be written after the main function but for this, write the function prototype in this section for the function which for you are going to write code after the main function.

```
//Function to implement the

//factorial of number num

INTEGER factorial(INTEGER num)

{

//Iterate over the loop from
```

//num to one

for(INTEGER i=1; i<=num; i++)

{

fact *=i;

}

//Return the factorial calculated

return fact;

}

## Main Function:

- The main function tells the compiler where to start the execution of the program. The execution of the program starts with the main function.
- All the statements that are to be executed are written in the main function.
- The compiler executes all the instructions which are written in the curly braces {} which encloses the body of the main function.
- Once all instructions from the main function are executed, control comes out of the main function and the program terminates and no further execution occur.

Below is the program to illustrate this:

C++

```
// Documentation Section
/*  This is a C++ program to find the
    factorial of a number
    The basic requirement for writing this
    program is to have knowledge of loops
    To find the factorial of a number
    iterate over the range from number to 1
*/

// Linking Section
#include <iostream>
using namespace std;

// Definition Section
#define msg "FACTORIAL\n"
typedef int INTEGER;

// Global Declaration Section
INTEGER num = 0, fact = 1, storeFactorial = 0;

// Function Section
INTEGER factorial(INTEGER num)
{
    // Iterate over the loop from
    // num to one
    for (INTEGER i = 1; i <= num; i++) {
        fact *= i;
    }

    // Return the factorial
    return fact;
}

// Main Function
INTEGER main()
{
    // Given number Num
    INTEGER Num = 5;

    // Function Call
    storeFactorial = factorial(Num);
    cout << msg;

    // Print the factorial
    cout << Num << "! = "
         << storeFactorial << endl;

    return 0;
}
```

## Introduction of advanced C++ concepts and feature of C++ 17:

C++17, released in December 2017, introduced several new features and improvements to the C++ programming language. Here are some of the key features of C++17:

- **Structured Bindings:** Structured bindings allow you to unpack the elements of a tuple or a struct into individual variables with a single declaration statement. This simplifies code and enhances readability.
- **if and switch with Initializer:** C++17 allows initialization statements (including variable declarations) directly inside if and switch statements, making code more concise and reducing the scope of variables.
- **Constexpr if**: constexpr if allows conditional compilation based on constant expressions, enabling more concise and expressive code in templates and generic programming.
- **std::optional:** The std::optional type provides a standardized way to represent optional values, allowing functions to return either a value or no value without resorting to null pointers or out-of-band signaling.
- **std::variant:** std::variant is a type-safe union that can hold values of different types. It provides a safer alternative to traditional unions by enforcing type safety and providing compile-time checks.
- **std::string_view:** std::string_view represents a view into a sequence of characters, allowing efficient manipulation of strings without requiring memory allocation or copying. It's particularly useful for passing substrings to functions without unnecessary copying.
- **std::any:** std::any provides a type-safe container for single values of any type, similar to boost::any. It allows storing values of different types in a single container without sacrificing type safety.
- **Parallel Algorithms:** C++17 introduces parallel versions of several standard algorithms (e.g., std::for_each, std::transform, std::reduce), allowing for easy parallelization of operations on containers using execution policies.
- **Filesystem Library**: The <filesystem> library provides a standardized way to manipulate files, directories, and paths in a platform-independent manner. It simplifies common filesystem operations and replaces platform-specific APIs like std::experimental::filesystem.
- **Other Language and Library Enhancements:** C++17 includes various other language and library enhancements, such as inline variables, constexpr lambdas, nested namespace definitions, and additional features in the standard library.

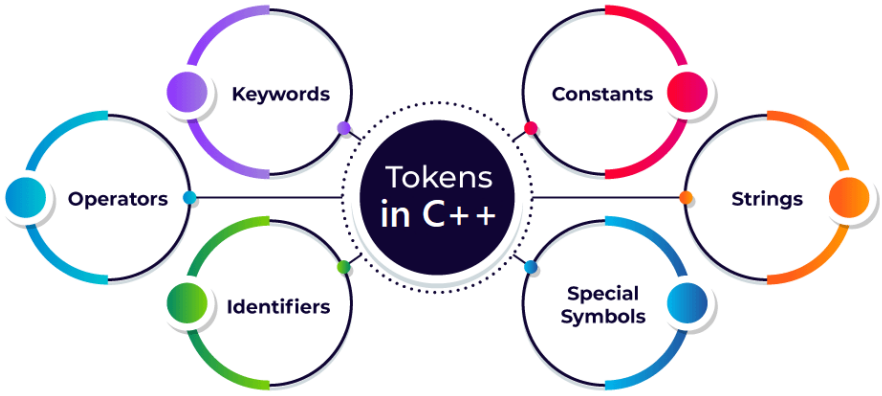
# C++ Tokens:

In C++, tokens can be defined as the smallest building block of C++ programs that the compiler understands. Every word in a C++ source code can be considered a token.

**Types of Tokens in C++:**

We have several types of tokens each of which serves a specific purpose in the syntax and semantics of C++. Below are the main types of tokens in C++:

1. Identifiers
2. Keywords
3. Constants
4. Strings
5. Special Symbols
6. Operators




## 1. Identifiers

In C++, entities like variables, functions, classes, or structs must be given unique names within the program so that they can be uniquely identified. The unique names given to these entities are known as identifiers.

It is recommended to choose valid and relevant names of identifiers to write readable and maintainable programs. Keywords cannot be used as an identifier because they are reserved words to do specific tasks. In the below example, "first_name' is an identifier.

string **first_name**= "Raju";

**We must follow a set of rules to define the name of identifiers as follows:**

1.An identifier can only begin with a **letter or an underscore (_).**

2.An identifier can consist of **letters** (A-Z or a-z), **digits** (0-9), and **underscores (_)**. White spaces and Special characters cannot be used as the name of an identifier.

3.Keywords cannot be used as an identifier because they are reserved words to do specific tasks. For example, **string**, **int, class**, **struct**, etc.

4. Identifier must be **unique** in its namespace.

5. As C++ is a case-sensitive language so identifiers such as 'first_name' and 'First_name' are different entities.

### Here are some examples of valid and invalid identifiers in C++:

| Valid Identifiers | Invalid Identifiers |
|---|---|
| _name | #name (Cannot start with special character except '_') |

| Valid Identifiers | Invalid Identifiers |
|---|---|
| Number89 | 2num (Cannot start with a digit) |
| first_name | first name (Cannot include space) |
| _last_name_ | string (Cannot be same as a keyword) |

**2. Keywords:**

Keywords in C++ are the tokens that are the reserved words in programming languages that have their specific meaning and functionalities within a program. Keywords cannot be used as an identifier to name any variables.

For example, a variable or function cannot be named as 'int' because it is reserved for declaring integer data type.

**3. Constants**

Constants are the tokens in C++ that are used to define variables at the time of initialization and the assigned value cannot be changed after that.

We can define the constants in C++ in two ways that are using the 'const' keyword and '#define' preprocessor directive.

## 4. Strings:

In C++, a **string** is not a built-in data type like 'int', 'char', or 'float'. It is a class available in the STL library which provides the functionality to work with a sequence of characters, that represents a string of text.

When we define any variable using the 'string' keyword we are actually defining an object that represents a sequence of characters. We can perform various methods on the string provided by the string class such as length(), push_backk(), and pop_back().

**Syntax of declaring a string:**

```
string variable_name;
```

Initialize the string object with string within the double inverted commas (**"**).

```
string variable_name = "This is string";
```

## 5. Special Symbols:

Special symbols are the token characters having specific meanings within the syntax of the programming language. These symbols are used in a variety of functions, including ending the statements, defining control statements, separating items, and more.

**Below are the most common special symbols used in C++ programming:**

- **Semicolon (;):** It is used to terminate the statement.
- **Square brackets []:** They are used to store array elements.
- **Curly Braces {}:** They are used to define blocks of code.
- **Scope resolution (::):** Scope resolution operator is used to access members of namespaces, classes, etc.
- **Dot (.):** Dot operator also called member access operator used to access class and struct members.
- **Assignment operator '=':** This operator is used to assign values to variables.
- **Double-quote ("):** It is used to enclose string literals.
- **Single-quote ('):** It is used to enclose character literals.

## 6. Operators:

C++ operators are special symbols that are used to perform operations on operands such as variables, constants, or expressions. A wide range of operators is available in C++ to perform a specific type of operations which includes arithmetic operations, comparison operations, logical operations, and more.

For example (A+B), in which 'A' and 'B' are operands, and '+' is an arithmetic operator which is used to add two operands.

### Types of Operators:

1. Unary Operators
2. Binary Operators
3. Ternary Operators

#### 1. Unary Operators

Unary operators are used with single operands only. They perform the operations on a single variable. For example, increment and decrement operators.

- **Increment operator ( ++ ):** It is used to increment the value of an operand by 1.
- **Decrement operator ( — ):** It is used to decrement the value of an operand by 1.

#### 2. Binary Operators

They are used with the two operands and they perform the operations between the two variables. For example (A<B), less than (<) operator compares A and B, returns true if A is less than B else returns false.

- **Arithmetic Operators:** These operators perform basic arithmetic operations on operands. They include **'+', '-', '*', '/',** and **'%'**
- **Comparison Operators:** These operators are used to compare two operands and they include '==', '!=', '<', '>', '<=', and '>='.
- **Logical Operators:** These operators perform logical operations on boolean values. They include **'**&&', '||', and '!**'.**

- **Assignment Operators:** These operators are used to assign values to variables and they include **‘variables**‘, ‘**-=**‘, ‘***=**‘, ‘**/=**‘, and ‘**%=’.**
- **Bitwise Operators:** These operators perform bitwise operations on integers. They include**‘&**’, ‘|’, ‘^’, ‘~’, ‘<<‘, and ‘>>**‘.**

### 3. Ternary Operator:

The ternary operator is the only operator that takes three operands. It is also known as a conditional operator that is used for conditional expressions.

**Syntax:**

Expression1 ? Expression2 : Expression3;

If ‘Expression1’ became **true** then ‘Expression2’ will be executed otherwise ‘Expression3’ will be executed.

# Initialization in C++:

Initialization in C++ refers to the process of assigning an initial value to a variable when it is created. C++ offers several ways to initialize variables, each with its own syntax and semantics. Here are some common initialization methods in C++:

### 1.Copy Initialization:

int x=10;

In copy initialization, the value on the right-hand side of the assignment is copied to initialize the variable on the left-hand side. This method can be used for fundamental types, user-defined types, and built-in arrays.

### 2.Direct Initialization:

int y(20);

Direct initialization uses parentheses instead of the assignment operator. It's generally more efficient than copy initialization because it avoids creating a temporary object.

### 3.Uniform Initialization (C++11 and later):

int z{30};

Uniform initialization, also known as brace initialization, uses braces {} to initialize variables. It provides consistent syntax for initialization and prevents narrowing conversions. It can be used with all types of variables.

### 4.Aggregate Initialization:

```
struct Point {
    int x;
    int y;
};

Point p = {5, 10};
```

Aggregate initialization allows initializing aggregates (arrays and structs) using a brace-enclosed list of values. Each value corresponds to a member of the aggregate in the order they are declared.

**5.List Initialization (C++11 and later):**

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
```

List initialization is a form of uniform initialization that uses braces {} to initialize containers, such as arrays, vectors, and initializer lists.

**6.Default Initialization:**

```
int a; // uninitialized, value is indeterminate
```

If a variable is declared without an explicit initializer, it is default-initialized. For fundamental types, this means the variable's value is indeterminate. For class types, default constructors are called.

**7.Initialization with Constructors:**

```
class MyClass {
public:
    MyClass(int value) : myValue(value) {}
private:
    int myValue;
};

MyClass obj(42);
```

Objects of user-defined types can be initialized using constructors. Constructor initialization is particularly useful for initializing members of classes.

# Static Member Function in C++

Static members in C++ are class members that are shared by all instances of the class. They belong to the class rather than to any individual object of the class. Here's an overview of static members:

1. **Static Data Members**: These are variables declared with the keyword **static** inside a class. They are shared by all instances of the class.

class MyClass

{

public: static int staticVariable;

```
};
int MyClass::staticVariable = 0; // Definition of static member outside the class
 int main()
{
MyClass::staticVariable = 10; // Accessing static member return 0;
 }
```

2. **Static Member Functions**: These are member functions declared with the keyword **static**. They can be called without creating an instance of the class and can only access static data members and other static member functions.

```
class MyClass {
 public: static void staticFunction()
 {
// Static member function
}
 };
 int main() {
MyClass::staticFunction(); // Calling static member function
return 0;
 }
```

3. **Static Constants**: In C++, **static** can also be used to declare class constants, which are shared by all instances of the class and cannot be modified.

```
class MyClass {
public:
static const int staticConstant = 100;
};
 int main() {
 int value = MyClass::staticConstant; // Accessing static constant
 return 0;
 }
```

Static members are commonly used for maintaining shared data or functionality across all instances of a class or for defining constants associated with a class. They are accessed using the scope resolution operator **::** followed by the class name.

# Constant member functions in C++:

The constant member functions are the functions which are declared as constant in the program. The object called by these functions cannot be modified. It is recommended to use const keyword so that accidental changes to object are avoided.

A constant member function can be called by any type of object. Non-const functions can be called by non-const objects only.

Here is the syntax of constant member function in C++ language,

```
datatype function_name const();
```

# Expressions Operators:

C++ Expression is a particular entity that contains constants, operators, and variables and arranges them according to the rules of the C++ language. If the question is what is an expression in C++ then it can be answered that the c++ expression contains function calls that return values. Every expression generates some values that are assigned to that particular variable with the help of the assignment operator.

# Arithmetic Operator:

Arithmetic Operators in C++ are used to perform arithmetic or mathematical operations on the operands. For example, '+' is used for addition, '–' is used for subtraction, '*' is used for multiplication, etc. In simple terms, arithmetic operators are used to perform arithmetic operations on variables and data; they follow the same relationship between an operator and an operand.

C++ Arithmetic operators are of 2 types:

1. Unary Arithmetic Operator
2. Binary Arithmetic Operator

## 1. Binary Arithmetic Operator:

These operators operate or work with two operands. C++ provides 5 *Binary Arithmetic Operators* for performing arithmetic functions:

| Operator | Name of the Operators | Operation | Implementation |
|---|---|---|---|
| + | Addition | Used in calculating the Addition of two operands | x+y |
| – | Subtraction | Used in calculating Subtraction of two operands | x-y |
| * | Multiplication | Used in calculating Multiplication of two operands | x*y |
| / | Division | Used in calculating Division of two operands | x/y |
| % | Modulus | Used in calculating Remainder after calculation of two operands | x%y |

## 2. Unary Operator:

These operators operate or work with a single operand.

| Operator | Symbol | Operation | Implementation |
|---|---|---|---|
| Decrement Operator | — | Decreases the integer value of the variable by one | –x or x — |
| Increment Operator | ++ | Increases the integer value of the variable by one | ++x or x++ |

# Relational Operator:

C++ Relational operators are used to compare two values or expressions, and based on this comparison, it returns a boolean value (either **true** or **false**) as the result. Generally **false** is represented as **0** and **true** is represented as any **non-zero value** (mostly **1**).

**How to use Relational Operator?**

All C++ relational operators are binary operators that are used with two operands as shown:

```
operand1   relational_operator  operand2
expression1   relational_operator  expression2
```

**Types of C++ Relational Operators:**
We have six relational operators in C++ which are explained below with examples.

| S. No. | Relational Operator | Meaning |
|---|---|---|
| **1.** | > | Greater than |
| **2.** | < | Less than |
| **3.** | >= | Greater than equal to |
| **4.** | <= | Less than equal to |
| **5.** | == | Equal to |
| **6.** | != | Not equal to |

# Logical Operator:

In C++ programming languages, logical operators are symbols that allow you to combine or modify conditions to make logical evaluations. They are used to perform logical operations on boolean values (**true** or **false**).

In C++, there are three logical operators:

1. **Logical AND ( && ) Operator**
2. **Logical OR ( || ) Operator**
3. **Logical NOT ( ! ) Operator**

## 1. Logical AND Operator ( && ):

The C++ **logical AND operator (&&)** is a binary operator that returns true if both of its operands are true. Otherwise, it returns false.

**Syntax of Logical AND**

*expression1* **&&** *expression2*

**2. Logical OR Operator ( || ):**
The C++ logical OR operator **( || )** is a binary operator that returns true if at least one of its operands is true. It returns false only when both operands are false.

**Syntax of Logical OR**

*expression1* || *expression2*

**3. Logical NOT Operator ( ! ):**
The C++ logical NOT operator ( ! ) is a unary operator that is used to negate the value of a condition. It returns true if the condition is false, and false if the condition is true.

**Syntax of Logical NOT**

**!** *expression*

# Unary Operator:

Unary operators in C++ are those operators that work on a single value (operand). They perform operations like changing a value's sign, incrementing or decrementing it by one, or obtaining its address. Examples include ++ for increment, — for decrement, + for positive values, – for negative values,! for logical negation, and & for retrieving memory addresses.

## Types of Unary Operators in C++

C++ has a total of 8 unary operators which are as follows:

1. Increment Operator (++)
2. Decrement Operator (–)
3. Unary Plus Operator (+)
4. Unary Minus Operator (-)
5. Logical NOT Operator (!)
6. Bitwise NOT Operator (~)
7. Addressof Operator (&)
8. Indirection Operator (*)
9. sizeof Operator

**1. Increment Operator (++):**

The increment operator is used to increase the value of its operand by 1. It works on the numeric operands and be use used as both prefix and postfix to the operand

**Syntax:**

++ *operand* or ++operand

**2. Decrement Operator (–):**

The decrement operator is used to decrement the value by 1. Just like the increment operator, it can be used as pre-decrement and post-decrement. It works on numeric values.

**Syntax:**

operand – or -- operand

## 3. Unary Plus Operator (+):

The symbol (+) represents the unary plus operator in C++, which explicitly specifies the positive value of its operand. It doesn't change the sign of a if it's already positive, but it can be used for clarity in expressions.

**Syntax**: "**+** *operand"*

### 4. Unary Minus (-) Operator:

The symbol (+) represents the unary plus operator in C++, which changes the sign of its operand to negative. If the operand is negative, this operator will make it positive and vice versa.

**Syntax**: "**-** *operand"*

### 5. Logical NOT Operator (!)

The logical NOT operator (!) is used for negating the value of a boolean expression. It returns true if the operand is false, and false if the operand is true.

**Syntax: "! *Operand"***

**6. Bitwise NOT Operator (~)**

The bitwise NOT operator ~ in C++ performs a bitwise negation operation on integral data types, such as integers. It inverts each bit of the operand, changing every 0 bit to 1 and every 1 bit to 0′.

**Syntax**: "**~** *operand"*

**7. Addressof Operator (&)**

The addressof operator (&) retrieves the memory address of a variable. It returns the memory location where the variable is stored in the computer's memory.

**Syntax**: "**&** *operand"*

**8. Dereference Operator (*)**

The dereference operator (*) is used to access the value at a specific memory address stored in a pointer. It "points to" the value stored at that Sddress.

**Syntax**: "***** *operand"*

**9. sizeof Operator:**The sizeof operator is a special unary operator in C++ that returns the size of its operand it bytes. Its operand can be any data type or a variable.

**Syntax**: "**sizeof** (*operand*)"

# Ternary Operator:

In C++, the ternary or conditional operator ( ? : ) is the shortest form of writing conditional statements. It can be used as an inline conditional statement in place of if-else to execute some conditional code.

Syntax of Ternary Operator ( ? : )

The syntax of the ternary (or conditional) operator is:

expression ? *statement_1* : *statement_2;*

As the name suggests, the ternary operator works on three operands where

- **expression:** Condition to be evaluated.
- **statement_1**: Statement that will be executed if the expression evaluates to true.
- **statement_2:** Code to be executed if the expression evaluates to false.

# Assignment Operator:

In C++, the assignment operator (=) is used to assign a value to a variable. It takes the value on its right and assigns it to the variable on its left.

**Syntax:** **"variable = value;"**

# Session 3: Conditional and Looping Statements

## If, else if, switch:

In C++, conditional statements like if, else if, and switch are used for decision-making in a program.

1. **if statement:** It is used to execute a block of code if a specified condition is true.

```
int x = 10; if (x > 5)
 {
// This block will execute because x is greater than 5
cout << "x is greater than 5";
}
```

2. **else if statement:** It is used to specify a new condition to test, if the first condition is false.

```
int x = 3; if (x > 5) {
cout << "x is greater than 5"; }
else if (x > 0) {
// This block will execute because x is greater than 0 but not greater than 5
cout << "x is positive but not greater than 5"; }
```

3. **switch statement:** It is used to select one of many code blocks to be executed.

```
int day = 4; switch (day) {
case 1: cout << "Monday";
break;
 case 2: cout << "Tuesday";
 break;
case 3: cout << "Wednesday";
break;
 case 4:
// This block will execute because day is 4
 cout << "Thursday";
break;
 default: cout << "Unknown day";
 }
```

In the switch statement, the break statement is used to exit the switch block. If break is not used, execution will continue with the next case without any checks. The default case is executed when none of the other cases match the value of the variable being switched on.

## for loop:

In C++, the for loop is a control flow statement that allows you to execute a block of code repeatedly for a fixed number of times. It has the following

**syntax:** for (initialization; condition; update) { // code to be executed }

- **Initialization:** Typically initializes a loop control variable. It's executed once before the loop starts.
- **Condition:** A boolean expression that is evaluated before each iteration. If it evaluates to true, the loop continues. If it evaluates to false, the loop terminates.
- **Update:** Executed at the end of each iteration. It's usually used to update the loop control variable.

Here's an example of a simple for loop:

```
for (int i = 0; i < 5; i++) {

cout << "Iteration " << i + 1 << endl;

 }
```

In this example:

- int i = 0; initializes the loop control variable i to 0.
- i < 5; is the condition. The loop will continue as long as i is less than 5.
- i++ is the update statement. It increments i by 1 after each iteration.

The loop will print:

```
Iteration 1

 Iteration 2

 Iteration 3

Iteration 4

 Iteration 5
```

You can also use the for loop to iterate over elements of arrays, containers, or other sequences.

For example:

```
vector<int> numbers = {1, 2, 3, 4, 5};

for (int i = 0; i < numbers.size(); i++)

 {

 cout << numbers[i] << " ";

 }
```

**Output:** 1 2 3 4 5

## while loop:

In C++, the while loop is a control flow statement that allows you to execute a block of code repeatedly as long as a specified condition is true.

**Syntax:** while (condition) { // code to be executed }

- **Condition**: A boolean expression that is evaluated before each iteration. If it evaluates to true, the loop continues. If it evaluates to false, the loop terminates.

Here's an example of a simple while loop:

```
int i = 0; while (i < 5)

{

 cout << "Iteration " << i + 1 << endl; i++;

 // Increment i to avoid an infinite loop

 }
```

In this example:

- int i = 0; initializes a loop control variable i to 0 before the loop starts.
- i < 5; is the condition. The loop will continue as long as i is less than 5.
- i++; increments i by 1 after each iteration.

The loop will print:

Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5

You can also use the while loop to iterate over elements of arrays, containers, or other sequences.

For example:

```
vector<int> numbers = {1, 2, 3, 4, 5};
```

```
int i = 0;

while (i < numbers.size())

{

cout << numbers[i] << " "; i++;

// Increment i to avoid an infinite loop

}
```

**Output:** 1 2 3 4 5

## do while loop:

In C++, the `do-while` loop is a control flow statement that is similar to the `while` loop. The difference is that the `do-while` loop executes its block of code at least once, and then it checks the condition.

**Syntax:**

```
do {

   // code to be executed

} while (condition);
```

**Code to be executed:** The block of code that you want to execute repeatedly.

**Condition**: A boolean expression that is evaluated after each iteration. If it evaluates to `true`, the loop continues. If it evaluates to `false`, the loop terminates.

**Example `do-while` loop:**

```
int i = 0;

do {

   cout << "Iteration " << i + 1 << endl;

   i++; // Increment i to avoid an infinite loop

} while (i < 5);
```

In this example:

- **int i = 0;** initializes a loop control variable `i` to 0 before the loop starts.
- **cout << "Iteration " << i + 1 << endl;** is the code to be executed. It prints the iteration number.
- **i++;** increments `i` by 1 after each iteration.
- **i < 5**; is the condition. The loop will continue as long as `i` is less than 5.

The loop will print:

Iteration 1

Iteration 2

Iteration 3

Iteration 4

Iteration 5

## Jump statement (break, continue& return keyword):

In C++, jump statements are used to alter the flow of control within loops, switch statements, and functions.

The three main jump statements are `break`, `continue`, and `return`.

1. **break:** The `break` statement is used to exit the loop or switch statement prematurely. When encountered within a loop or switch statement, it causes the immediate termination of the loop or switch, and control is transferred to the statement immediately following the loop or switch.

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break; // Exit the loop when i reaches 3
    }
    cout << i << " ";
}
```

**Output:** 0 1 2

2. **Continue:** The `continue` statement is used to skip the current iteration of a loop and continue with the next iteration. When encountered within a loop, it skips the remaining code in the loop body and proceeds with the next iteration.

```
for (int i = 0; i < 5; i++) {
    if (i == 2) {
```

```
        continue; // Skip iteration when i is 2
    }
    cout << i << " ";
}
```

**Output:** 0 1 3 4

3. **return:** The `return` statement is used to exit a function and return a value to the calling function. It can also be used to return early from a function, even if it's declared to return void.

```
int add(int a, int b) {
    return a + b; // Return the sum of a and b
}
void printMessage() {
    cout << "This message is printed before return." << endl;
    return; // Return early, the function exits here
    cout << "This message is not printed because it's after the return statement." << endl;
}
```

These jump statements provide control over the flow of execution in C++ programs, allowing you to exit loops early, skip iterations, or return values from functions.

## Arrays:

In C++, an array is a collection of elements of the same data type stored in contiguous memory locations. Each element in the array is accessed by its index, starting from 0. Arrays provide a convenient way to store and manipulate a fixed-size sequence of elements.

Here's how you declare an array in C++:

```
dataType arrayName[arraySize];
```

- **dataType**: Specifies the data type of the elements in the array.
- **arrayName:** Name of the array.
- **arraySize:** Specifies the number of elements in the array.

Example, to declare an array of integers with 5 elements:

```
int numbers[5];
```

You can also initialize the array at the time of declaration:

**int numbers[5] = {1, 2, 3, 4, 5};**

Or you can partially initialize it:

**int numbers[5] = {1, 2}; // Initializes the first two elements, rest are set to 0**

Accessing elements of an array is done using square brackets `[]` with the index of the element you want to access:

**int value = numbers[2]; // Accesses the third element (index 2) of the array**

Arrays are zero-indexed, meaning the first element is at index 0, the second element is at index 1, and so on.

You can also iterate over the elements of an array using loops like `for` or `while`:

```
for (int i = 0; i < 5; i++) {
    cout << numbers[i] << " ";
}
```

Arrays are very useful in many programming scenarios, but they have some limitations, such as fixed size and lack of bounds checking. In modern C++, you might prefer to use standard library containers like `std::vector` for more flexibility and safety.

## Declaration and initialization of an array:

Declare and initialize an array in C++:

```
// Declaration of an array of integers with size 5
    int numbers[5];
// Initialization of the array
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

This code declares an array named `numbers` of type `int` with a size of 5. Then it initializes each element of the array individually. Alternatively, you can declare and initialize the array in one step:

// Declaration and initialization of an array of integers with size 5

```
int numbers[5] = {10, 20, 30, 40, 50};
```

In this case, the values `{10, 20, 30, 40, 50}` are assigned to the elements of the array `numbers` in sequence.

If you're using C++11 or later, you can also use the uniform initialization syntax with curly braces:

// Uniform initialization of an array of integers with size 5

```
int numbers[5] {10, 20, 30, 40, 50};
```

This syntax is equivalent to the previous initialization methods but is more consistent with initialization of other types of variables in modern C++.

## 1-D and 2-D arrays:

**1-D Arrays:** A 1-D array is a linear collection of elements of the same data type. It is declared and initialized as follows:

```
// Declaration and initialization of a 1-D array

dataType arrayName[arraySize] = {value1, value2, ..., valueN};
```

Example of a 1-D array of integers:

```
int numbers[5] = {1, 2, 3, 4, 5};
```

**2-D Arrays:** A 2-D array is a collection of elements organized in rows and columns. It is declared and initialized as follows:

```
// Declaration and initialization of a 2-D array

dataType arrayName[rowSize][colSize] = {{value11, value12, ...}, {value21, value22, ...}, ..., {valueM1, valueM2, ...}};
```

Example of a 2-D array of integers:

```
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

In this example, `matrix` is a 3x3 array.

You can access elements of a 2-D array using two indices: one for the row and one for the column. For example:

```
int element = matrix[1][2]; // Accesses the element at row 1, column 2 (which is 6 in this case)
```

You can also use loops to iterate over the elements of a 2-D array:

```
for (int i = 0; i < 3; i++) {
```

```
    for (int j = 0; j < 3; j++) {
        cout << matrix[i][j] << " ";
    }
    cout << endl;
}
```

This code will print the entire matrix.

# Session 4: Functions in C++:

## Different forms of functions:

In C++, functions can take different forms based on their return type, parameters, and whether they belong to a class. Here are the main forms of functions in C++:

**1.Standard (Non-member) Functions:** These are standalone functions that are not part of any class. They are declared and defined outside of any class.

```
// Declaration
returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...);

// Definition
returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...) {
    // Function body
}
```

**2. Member Functions:** These functions belong to a class and can access the member variables and other member functions of that class. They are defined within the class declaration.

```
class MyClass {
public:
    // Declaration
    returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...);

    // Definition
    returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...) {
        // Function body
    }
};
```

**3.Static Member Functions:** These are member functions that belong to the class rather than to any specific object instance. They can be called using the class name, without creating an object of the class.

```
class MyClass {
public:
    // Declaration
    static returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...);

    // Definition
    static returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...) {
        // Function body
```

```
    }
};
```

**4.Function Overloading:** In C++, you can define multiple functions with the same name but with different parameter lists. This is called function overloading.

```
// Function with one parameter
returnType functionName(parameterType parameter) {
    // Function body
}

// Overloaded function with two parameters
returnType functionName(parameterType1 parameter1, parameterType2 parameter2) {
    // Function body
}
```

**5. Recursive Functions:** These are functions that call themselves either directly or indirectly. They are often used to solve problems that can be broken down into smaller, similar sub-problems.

```
returnType functionName(parameterType parameter) {
    if (base case) {
        // Base case: return some value
    } else {
        // Recursive case: call the function recursively with modified parameters
        functionName(modifiedParameter);
    }
}
```

Each form of function has its own use cases and advantages. Understanding these different forms allows you to write more versatile and efficient C++ code.

## Function prototyping:

Function prototyping in C++ allows you to declare the signature of a function before its actual implementation. This allows you to use the function in your code before defining it. This is especially useful when you have functions that call each other or when you want to separate the interface of your functions from their implementation.

**Syntax :** returnType functionName(parameterType1, parameterType2, ...);

**For example:**

```
// Function prototype

int add(int a, int b);

int main() {

    int result = add(3, 5);

    cout << "Result: " << result << endl;

    return 0; }
```

**// Function definition**

```
int add(int a, int b) {
    return a + b;
}
```

In this example, `add` function is prototyped at the beginning of the program before its actual definition. This allows `main` function to call `add` before its implementation.

It's common to place function prototypes in header files (.h files) and function definitions in source files (.cpp files). This way, you can include the header file in multiple source files to make the functions available to all of them.

Function prototyping helps in:

1. **Order of Definition:** You can define functions in any order, regardless of the order in which they are called.

2. **Readability**: It provides a clear separation between function declarations and their implementations, making the code easier to read and understand.

3. **Modularity**: It allows you to divide your program into smaller, more manageable parts, each with its own interface.

## Call by Reference:

In C++, passing arguments by reference allows a function to modify the original value of a variable passed to it. When you pass an argument by reference, you pass the memory address of the variable rather than its value. This means that any changes made to the parameter within the function will affect the original variable outside the function.

Here's how you can pass arguments by reference:

```
void myFunction(int& x) {
    x = x * 2; // Modify the value of x
}

int main() {
    int number = 5;
    cout << "Before: " << number << endl; // Output: Before: 5
    myFunction(number); // Pass by reference
    cout << "After: " << number << endl; // Output: After: 10
    return 0;}
```

In this example, `number` is passed to the function `myFunction` by reference. Inside `myFunction`, any modifications made to `x` will directly affect `number`. As a result, `number` is doubled inside the function, and its updated value is reflected outside the function.

Key points about passing by reference in C++:

**1. Syntax:** To pass by reference, you use the `&` symbol after the parameter type in the function declaration.

**2. Memory Address:** Passing by reference allows you to avoid making a copy of the argument, which can be more efficient for large objects.

**3. Modification:** Any changes made to the parameter inside the function will affect the original variable outside the function.

**4. Readability and Clarity:** Passing by reference can make the code clearer, especially when a function modifies its arguments.

**5.Risk of Unexpected Modification:** Be cautious when passing arguments by reference, as it can lead to unexpected modifications of variables, especially in larger codebases.

Passing by reference is particularly useful when you need to modify variables within functions or when you want to avoid unnecessary copies of large objects.

## Inline Functions:

In C++, an inline function is a function that is expanded in place wherever it is called, rather than being called like a normal function. The compiler replaces the function call with the actual code of the function body, which can lead to faster execution time, especially for small and frequently called functions.

To declare an inline function, you use the `inline` keyword before the function declaration:

```
inline returnType functionName(parameterType1 parameter1, parameterType2 parameter2, ...) {
    // Function body
}
```

**For example:**

```
inline int add(int a, int b) {
    return a + b;
}
```

Here are some key points about inline functions:

**1. Performance**: Inline functions can improve performance by avoiding the overhead of function calls. This is especially beneficial for small functions that are called frequently.

**2. Size:** Inline functions may increase the size of the executable because the function code is duplicated at each call site. However, this increase is usually minimal for small functions.

**3. Recommendation:** The decision to use inline functions should be made judiciously. They are most effective for small functions (e.g., accessors, mutators, or simple operations) that are called frequently.

**4. Single-Statement Functions**: Inline functions are often used for single-statement functions, such as getters and setters, or for simple arithmetic operations.

**5. Header-Only Libraries:** Inline functions are commonly used in header-only libraries, where the implementation is provided directly in the header file.

**Example:**

```
#include <iostream>

inline int add(int a, int b) {

    return a + b;

}

int main() {

    int result = add(3, 5); // Compiler replaces this with: int result = 3 + 5;

    std::cout << "Result: " << result << std::endl;

    return 0;

}
```

In this example, the `add` function is declared as inline, and the compiler replaces the call to `add(3, 5)` with `3 + 5` directly, resulting in efficient code execution.

## Math library functions:

In C++, the `<cmath>` header provides a variety of mathematical functions for performing common mathematical operations. These functions are part of the standard library and can be used by including the `<cmath>` header in your program. Here are some commonly used mathematical functions available in the `<cmath>` header:

**1. Trigonometric Functions:**

- **sin(x):** Returns the sine of `x` (where `x` is in radians).
- **cos(x):** Returns the cosine of `x` (where `x` is in radians).
- **tan(x):** Returns the tangent of `x` (where `x` is in radians).
- **asin(x):** Returns the arc sine of `x`, in the range of [-π/2, π/2] radians.
- **acos(x):** Returns the arc cosine of `x`, in the range of [0, π] radians.
- **atan(x):** Returns the arc tangent of `x`, in the range of [-π/2, π/2] radians.

**2. Exponential and Logarithmic Functions:**

- **exp(x):** Returns the exponential value of `x`.
- **log(x):** Returns the natural logarithm (base `e`) of `x`.
- **log10(x):** Returns the base-10 logarithm of `x`.
- **pow(x, y):** Returns `x` raised to the power of `y`.
- **sqrt(x):** Returns the square root of `x`.

**3. Rounding and Absolute Functions:**

- **ceil(x):** Returns the smallest integer greater than or equal to `x`.
- **floor(x):** Returns the largest integer less than or equal to `x`.
- **fabs(x):** Returns the absolute value of `x`.
- **round(x):** Returns the nearest integer value to `x`, rounding halfway cases away from zero.

**4. Constants:**

**M_PI:** Represents the mathematical constant π (pi).

These are just a few examples of the functions available in the `<cmath>` header. There are many more mathematical functions provided, covering a wide range of operations. Remember to include the `<cmath>` header at the beginning of your program to use these functions.

Memory Management and Pointers:

**Introduction:**

- Memory management in C++ involves allocating and deallocating memory dynamically during program execution.
- It ensures efficient utilization of memory resources and avoids memory leaks and dangling pointers.

**Pointers in C++:**

- Pointers are variables that store memory addresses.
- They allow for dynamic memory allocation, accessing memory directly, and creating data structures like linked lists and trees.

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

int main() {
    int x = 10;
    int* ptr = &x; // Pointer declaration and initialization

    cout << "Value of x: " << x << endl;
    cout << "Address of x: " << &x << endl;
    cout << "Value stored in ptr: " << ptr << endl;
    cout << "Value at address pointed by ptr: " << *ptr << endl;

    return 0;
}
```

**Arrays Using Pointers**:

- Arrays in C++ can be accessed using pointers.
- Array elements are stored in contiguous memory locations.

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int* ptr = arr; // Pointer to the first element of the array

    cout << "Array elements using pointer: ";
    for (int i = 0; i < 5; ++i) {
        cout << *(ptr + i) << " ";
    }
    cout << endl;

    return 0;
}
```

**Enumeration**:

- Enumeration (enum) is a user-defined data type used to assign names to integral constants.
- Enums improve code readability and maintainability.

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

enum Color {RED, GREEN, BLUE};

int main() {
    Color c = RED;
    cout << "Value of RED: " << c << endl; // Output: 0

    return 0;
}
```

**Typedef**:

- Typedef is used to create a new name for an existing data type.
- It enhances code readability and portability.

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

typedef int Age;

int main() {
    Age myAge = 25;
    cout << "My age is: " << myAge << endl;

    return 0;
}
```

**Using New Operator**:

- The **new** operator is used to dynamically allocate memory for objects on the heap.
- It returns a pointer to the allocated memory.

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

int main() {
    int* ptr = new int(10);
    cout << "Value stored in dynamically allocated memory: " << *ptr << endl;

    delete ptr; // Freeing dynamically allocated memory

    return 0;
}
```

**Class Pointer and this Pointer**:

- Pointers can be used to access class members.
- The **this** pointer refers to the current object instance within a member function.

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

class MyClass {
public:
    int x;

    void setX(int val) {
        this->x = val; // Using this pointer to access class member
    }
};

int main() {
    MyClass obj;
    obj.setX(5);
    cout << "Value of x: " << obj.x << endl;

    return 0;
}
```

**Comparison of New over Malloc, Calloc, and Realloc:**

- **new** is preferred in C++ as it handles object construction and initialization.
- **malloc**, **calloc**, and **realloc** are used in C and require explicit memory management.

**Memory Freeing Using Delete Operator**:

- The **delete** operator is used to free dynamically allocated memory.

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

int main() {
    int* ptr = new int(10);
    cout << "Value stored in dynamically allocated memory: " << *ptr << endl;

    delete ptr; // Freeing dynamically allocated memory

    return 0;
}
```

# OOPS – CONCEPTS.

## Discussion on object oriented concepts. –

Object-oriented programming (OOP) is a programming building block like a complete construction of programming logic based on the concept of "objects", which can contain data in the form of fields and code in the form of procedures (often known as methods). Here's a brief discussion on some key concepts of object-oriented programming:

1. **Classes and Objects** :
   - A class is a blueprint for creating objects. It defines the attributes and behaviors (data and functions) that all objects of the class will have.
   - An object is an instance of a class. It represents a specific entity with its own unique data and behavior.
2. **Encapsulation** :
   - Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, called a class.
   - Encapsulation helps in hiding the internal state of an object and only exposing the necessary interfaces for interacting with it. This is often achieved by making the data members private and providing public methods to access and modify them.
   - # an approach which is used to protect method and function in a single entity.
3. **Inheritance** :
   - Inheritance is the mechanism by which one class can inherit properties and behavior from another class.
   - The class that is inherited from is called the base class or superclass, and the class that inherits is called the derived class or subclass.
   - Inheritance promotes code reusability and allows for the creation of a hierarchy of classes.
4. **Polymorphism** :
   - Polymorphism allows objects of different classes to be treated as objects of a common superclass.
   - It enables a single interface to be used for different types of objects, thus providing flexibility and extensibility in the design.
   - Polymorphism can be achieved through method overriding (runtime polymorphism) and method overloading (compile-time polymorphism).
5. **Abstraction** :
   - Abstraction refers to the process of hiding the complex implementation details and showing only the essential features of an object.
   - It allows programmers to focus on the relevant aspects of an object while ignoring the irrelevant details.
   - Abstraction is often achieved through abstract classes and interfaces, which define a set of methods without providing the implementation.

6. **Association, Aggregation, and Composition** :

- Association represents a relationship between two or more classes, where objects of one class are related to objects of another class.
- Aggregation and composition are forms of association that represent "has-rel "relationships bbetween classes:
  - Aggregation implies a weaker relationship, where one class contains objects of another class, but the contained objects can exist independently.
  - Composition implies a stronger relationship, where one class contains objects of another class, and the contained objects are dependent on the container class.

***Class:***

A class is a blueprint or template for creating objects. It defines the attributes (data members) and behaviors (member functions) that all objects of that class will have.

Classes provide a way to model real-world entities in software by encapsulating

**Example:**

```
// Define a class named 'Person' to represent people
class Person { public:
   // Data members (attributes)
string    name;
   int age;


   // Member functions (behaviors)

   void displayInfo() {
 cout << "Name: " << name << ", Age: " << age << endl;
   } };
```

***Object:***

An object is an instance of a class. It represents a specific entity or instance of the class with its own unique set of data and behaviors.

Objects are created based on the blueprint defined by the class. Each object has its own copy of the class's data members and can invoke its member functions.

Example:

```
int main() {
   // Create objects of the 'Person'
class
   Person person1, person2;

   // Set data for person1
 person1.name = "Alice";
   person1.age = 30;

   // Set data for person2
person2.name = "Bob";
```

```
 person2.age = 25;

   // Call member function to display information about person1
person1.displayInfo();

   // Call member function to display information about person2
person2.displayInfo();

   return 0;
}
```

== OUTPUT: ==

Name: Alice, Age: 30
Name: Bob, Age: 25

***ACCESS SPECIFIERS :***

Access specifiers in C++ are keywords used to control the accessibility of class members (data members and member functions) from outside the class. There are three access specifiers in C++:

- ✓ **Public:**

Members declared as public are accessible from outside the class.
Public members can be accessed by objects of the class and by functions that are not members of the class.

- Public:
- Int a,b,c ;

- ✓ **Private:**

Members declared as private are not accessible from outside the class.
Private members can only be accessed by member functions of the same class.
By default, all members of a class are private if no access specifier is specified.
Protected:
Members declared as protected are similar to private members, but they are accessible in derived classes.
Protected members can only be accessed by member functions of the same class or by derived classes.

- Private:
- Int a,b ;
- Can not be directly accessible.

***Overloading:***

Function overloading is a feature in C++ that allows you to define multiple functions with the same name but with different parameter lists. The compiler determines which function to call based on the number and types of arguments passed to it.

CODE---

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

// Function to add two integers

int add(int a, int b)
{
 return a + b;
}

// Function to add three integers

int add(int a, int b, int c) {
return a + b + c;
}

// Function to add two doubles double
add(double a, double b) {
   return a + b;

}

int main() {
int sum1 = add(3, 5); // Calls the first add() function
cout << "Sum of 3 and 5: " << sum1 << endl;


 int sum2 = add(3, 5, 7); // Calls the second add() function
cout << "Sum of 3, 5, and 7: " << sum2 << endl;


 double sum3 = add(3.5, 2.5); // Calls the third add() function
cout << "Sum of 3.5 and 2.5: " << sum3 << endl;
  return 0;
}
```

Output:

Sum of 3 and 5  : 8

Sum of 3, 5, and 7  : 15

Sum of 3.5 and 2.5  : 6

***INHERITANCE:***

Inheritance is a key concept in object-oriented programming that allows a new class (derived class) to inherit properties and behavior from an existing class (base class). In C++, inheritance is implemented using the class keyword along with the public, protected, or private access specifiers. Here's an overview of inheritance in C++:

- ✓ Base Class (Parent Class):

A base class is the existing class from which properties and behavior are inherited.

It serves as the foundation for derived classes.

It contains common attributes and methods that can be shared among multiple derived classes.

- ✓ Derived Class (Child Class):

A derived class is a new class that inherits properties and behavior from a base class.

It extends or specializes the functionality of the base class by adding new features or overriding existing ones.

It can also have its own additional attributes and methods.

- *Types of Inheritance:*
  - Single Inheritance: A derived class inherits from only one base class.
  - Multiple Inheritance: A derived class inherits from multiple base classes.
  - Multilevel Inheritance: A derived class serves as a base class for another derived class.
  - Hierarchical Inheritance: Multiple derived classes inherit from a single base class.
  - Hybrid Inheritance: Combination of multiple types of inheritance.

Syntax:

In C++, inheritance is specified using the class keyword followed by a colon (:), followed by the access specifier (public, protected, or private) and the name of the base class.

The access specifier determines the visibility of the base class members in the derived class.

```
// Base class
class Shape {
public:
 double width;
 double height;
 void setWidth(double w) {
 width = w;
 }
 void setHeight(double h) {
 height = h;
 }
};
```

```
// Derived class
class Rectangle : public Shape {
public:
 double getArea() {
 return width * height;
 }
};
int main() {
 Rectangle rect;
 rect.setWidth(5.0);
 rect.setHeight(3.0);
 cout << "Area of the rectangle: " << rect.getArea() << endl;
 return 0;
}
```

***TYPES OF INHERITENCE:***

**Single Inheritance:**

Single inheritance is the simplest form of inheritance, where a derived class inherits properties from a single base class.

In single inheritance, the derived class has only one immediate base class.

Example:

```
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
 void sound() {
 cout << "Animal makes a sound" << endl;
 }
};
// Derived class inheriting from Animal
class Dog : public Animal {
public:
 void bark() {
 cout << "Dog barks" << endl;
 }
};
int main() {
 // Creating an object of the derived class
 Dog myDog;
 // Accessing methods from the base class
 myDog.sound();
 // Accessing methods from the derived class
 myDog.bark();
 return 0;
}
```

In this example:

Animal is the base class, which has a method sound().

Dog is the derived class, which inherits from Animal.

The Dog class adds a method bark().

When an object of Dog is created, it can access both the sound() method inherited from Animal and the bark() method defined in Dog.

Single inheritance is useful for creating a hierarchical relationship where derived classes share common behavior from a single base class. It promotes code reuse and simplifies the structure of the program.

**Multiple Inheritance:**

Multiple inheritance allows a derived class to inherit properties from multiple base classes.

In multiple inheritance, the derived class has more than one immediate base class. . Multiple inheritance forms a hierarchy where a class inherits from more than one class, forming a diamond-shaped inheritance structure.

Example:

```
#include <iostream>
using namespace std;
// Base class 1
class Animal {
public:
 void sound() {
 cout << "Animal makes a sound" << endl;  } };
// Base class 2
class Vehicle {
public:
 void drive() {
 cout << "Vehicle is being driven" << endl;  } };
// Derived class inheriting from both Animal and Vehicle
class DogCar : public Animal, public Vehicle {
public:
 void bark() {
 cout << "Dog barks" << endl; }  };
int main() {
 DogCar myDogCar;// Creating an object of the derived class
 myDogCar.sound(); // Accessing method from Animal
 myDogCar.drive(); // Accessing method from Vehicle
 // Accessing method from the derived class
 myDogCar.bark();
 return 0;
```

In this example:

Animal and Vehicle are two base classes.

DogCar is the derived class, which inherits from both Animal and Vehicle.

When an object of DogCar is created, it can access methods from both Animal and Vehicle, as well as its own methods.

Multiple inheritance can lead to ambiguity if both base classes have methods with the same name. In such cases, you may need to use scope resolution (::) to specify which method to call, or use virtual inheritance to resolve the ambiguity.

Multiple inheritance allows for greater flexibility in class design but can also make the code more complex and harder to maintain. It should be used judiciously based on the specific requirements of the program.

**Multilevel Inheritance:**

Multilevel inheritance involves a chain of inheritance where a derived class inherits properties from a base class, and another class inherits from this derived class.

syntax: class Derived1 : public Base { ... }; class Derived2 : public Derived1
{ ... };

Example:

```cpp
#include <iostream>
using namespace std;
class Animal {
public:
 void sound() {
 cout << "Animal makes a sound" << endl; } };
class Dog : public Animal {// Derived class inheriting from Animal
public:
 void bark() {
 cout << "Dog barks" << endl; } };
class Labrador : public Dog {// Derived class inheriting from Dog
public:
 void color() {
 cout << "Labrador is black" << endl; } };
int main() {
 Labrador myLab;// Creating an object of the derived class
 myLab.sound(); // Accessing method from Animal
 myLab.bark(); // Accessing method from Dog
 // Accessing method from the derived class
 myLab.color();
 return 0;
}
```

In this example:

Animal is the base class.

Dog is the derived class inheriting from Animal.

Labrador is another derived class inheriting from Dog.

When an object of Labrador is created, it can access methods from both Animal and Dog, as well as its own methods.

Multilevel inheritance creates a hierarchy where each derived class adds its own specific properties while sharing the common properties of the base classes.

Multilevel inheritance is useful for creating a hierarchical relationship where derived classes can inherit properties from multiple levels of base classes

**Hierarchical Inheritance:**

Hierarchical inheritance involves multiple derived classes inheriting from a single base class.

Each derived class may add its own specific properties while sharing the common properties of the base class.

syntax: class Base { ... }; class Derived1 : public Base { ... }; class Derived2 : public Base { ... };

example:

```
#include <iostream>
using namespace std;
class Animal {
public:
 void sound() {
 cout << "Animal makes a sound" << endl;  }  };
class Dog : public Animal {// Derived class 1 inheriting from Animal
public:
 void bark() {
 cout << "Dog barks" << endl;}  };
class Cat : public Animal {// Derived class 2 inheriting from Animal
public:
 void meow() {
 cout << "Cat meows" << endl;  }  };
int main() {
 // Creating objects of the derived classes
 Dog myDog;
 Cat myCat;
 myDog.sound(); // Accessing method from Animal
 myCat.sound(); // Accessing method from Animal
 myDog.bark();
 myCat.meow();
 return 0;
}
```

In this example:

Animal is the base class, which has a method sound().

Dog and Cat are two derived classes inheriting from Animal.

Both Dog and Cat share the sound() method from the Animal class.

Each derived class adds its own specific methods (bark() for Dog and meow() for Cat).

Hierarchical inheritance allows for code reuse and promotes a clear hierarchical relationship between classes.

Hierarchical inheritance is useful when multiple derived classes share common properties with a single base class but have their own distinct behavior. It simplifies the structure of the program and promotes code reusability. However, it's important to ensure that the base class contains properties that are relevant to all derived classes in the hierarchy.

## Hybrid Inheritance:

Hybrid inheritance is a combination of multiple types of inheritance.

It can include multiple, multilevel, and hierarchical inheritance in the same class hierarchy.

Example: Combination of single, multiple, multilevel, or hierarchical inheritance in a single class hierarchy.

```
#include <iostream>
using namespace std;
class Animal {
public:
 void sound() {
 cout << "Animal makes a sound" << endl;  }  };
class Vehicle {
public:
 void drive() {
 cout << "Vehicle is being driven" << endl; } };
class DogCar : public Animal, public Vehicle {
public:
 void bark() {
 cout << "Dog barks" << endl; }  };
// Derived class inheriting from DogCar
class LabradorCar : public DogCar {
public:
 void color() {
 cout << "Labrador is black" << endl; } };
int main() {
 // Creating an object of the derived class
 LabradorCar myLabCar;
 // Accessing methods from the base classes
 myLabCar.sound(); // Accessing method from Animal
 myLabCar.drive(); // Accessing method from Vehicle
 // Accessing methods from the derived classes
 myLabCar.bark(); // Accessing method from DogCar
 myLabCar.color(); // Accessing method from LabradorCar
 return 0;
}
```

In this example:

Animal and Vehicle are two base classes.

DogCar is a derived class inheriting from both Animal and Vehicle, demonstrating multiple inheritance.

LabradorCar is another derived class inheriting from DogCar, demonstrating multilevel inheritance.

The LabradorCar class exhibits hybrid inheritance, combining multiple and multilevel inheritance.

Objects of LabradorCar can access methods from Animal, Vehicle, DogCar, and LabradorCar classes.

Hybrid inheritance allows for complex class hierarchies that combine different inheritance patterns to achieve specific design goals. However, it can lead to code complexity and potential ambiguity, so it should be used judiciously based on the requirements of the program.

## Virtual Inheritance:

Virtual inheritance is used to resolve ambiguity that arises when a derived class inherits from multiple base classes that have a common base class.

It ensures that only one instance of the common base class is inherited by the derived class.

Example: class Derived : public virtual Base1, public virtual Base2 { ... };

Each type of inheritance has its own use cases and implications, and the choice of inheritance depends on the design requirements and the relationship between classes in the program. .

***:VIRTUAL BASE CLASS:***

In C++, a virtual base class is a base class that resolves ambiguity in multiple inheritance hierarchies. When a class is declared as a virtual base class, only one instance of its member variables is inherited by the derived classes, even if the class appears multiple times in the inheritance hierarchy.

The virtual base class mechanism is used to avoid the "diamond problem" or "diamond inheritance problem," which occurs when a derived class inherits from two or more classes that have a common base class. Without virtual inheritance, this can lead to ambiguity in accessing members of the base class through the derived class.

```cpp
#include <iostream>
using namespace std;
// Base class
class Animal {
public:
 void sound() {
 cout << "Animal makes a sound" << endl;  }  };
class Dog : public Animal {// Intermediate class 1
public:
 void bark() {
 cout << "Dog barks" << endl; } };
class Cat : public Animal {// Intermediate class 2
public:
 void meow() {
 cout << "Cat meows" << endl;  } };
class Hybrid : public Dog, public Cat {// Derived class inheriting from both Dog and Cat
public:
 void hybridAction() {
 cout << "Hybrid performs action" << endl;  }  };
int main() {
 Hybrid myHybrid;// Creating an object of the derived class
 // Accessing methods from the base classes
 myHybrid.sound(); // Error: ambiguous call
 return 0;
}
```

In this example, the Hybrid class inherits from both Dog and Cat, which in turn inherit from Animal. When trying to access the sound() method through an object of Hybrid, the compiler generates an error due to ambiguity in the inheritance hierarchy.

To resolve this ambiguity, we can use virtual inheritance for the common base class Animal:

```
// Intermediate class 1 with virtual inheritance
class Dog : virtual public Animal {
public:
 void bark() {
 cout << "Dog barks" << endl;
 }
};
// Intermediate class 2 with virtual inheritance
class Cat : virtual public Animal {
public:
 void meow() {
 cout << "Cat meows" << endl;
 }
};
```

By declaring Animal as a virtual base class in Dog and Cat, only one instance of Animal's member variables will be inherited by Hybrid, resolving the ambiguity in the inheritance hierarchy.

Virtual base classes are essential for managing complex inheritance hierarchies and preventing ambiguity, especially in cases of multiple inheritance. However, they should be used judiciously to avoid unintended consequences and maintain code clarity.

***CONSTRUCTOR IN DERIVED CLASS:***

In object-oriented programming, constructors are special methods used for initializing objects. When you derive a class from a base class (inheritance), you often want to customize the initialization process by providing your own constructor in the derived class. Here's how constructors work in derived classes:

- ✓ Implicit Inheritance: By default, when you create a derived class, it inherits the constructors from its base class. This means that you can create objects of the derived class using the constructors of the base class.
- ✓ Custom Constructors: You can also define your own constructors in the derived class. These constructors can initialize the data members of the derived class and optionally call the constructor of the base class to initialize inherited members.
- ✓ Constructor Overloading: Just like in the base class, you can overload constructors in the derived class. This allows you to provide multiple ways of initializing objects.

Here's a simple example in C++ to illustrate constructors in derived classes:

```cpp
#include <iostream>
using namespace std;
// Base class
class Base {
public:
 Base() {
 cout << "Base class constructor" << endl;
 }
 Base(int x) {
 cout << "Base class constructor with parameter " << x << endl;
 }
};
// Derived class
class Derived : public Base {
public:
 Derived() {
 cout << "Derived class constructor" << endl;
 }
 Derived(int x) : Base(x) {
 cout << "Derived class constructor with parameter " << x << endl;
 }
};
int main() {
 // Creating objects
 Derived obj1; // Calls default constructors of Base and Derived
 Derived obj2(10); // Calls parameterized constructors of Base and Derived
 return 0;
}
```

In this example:

Derived is derived from Base.

Derived has its own constructors, and it also calls the constructor of Base explicitly with Derived(int x) : Base(x).

When Derived objects are created, constructors of both Derived and Base classes are called appropriately.

**********

***POLYMORPHISM:***

Polymorphism is a fundamental concept in object-oriented programming that allows objects to be treated as instances of their parent class, even when they are actually instances of a derived class. This enables flexibility in programming and supports code reuse and extensibility. Polymorphism is typically achieved through two mechanisms:function overloading and function overriding.

Types:

- ✓ Compile-time Polymorphism (Function Overloading):

Compile-time polymorphism, also known as static polymorphism, is achieved through function overloading.

Function overloading allows multiple functions with the same name but different parameter lists to exist within the same scope.

The appropriate function to be called is determined by the number and types of arguments passed to it at compile time.

This allows the same function name to be used for different behaviors, improving code readability and reducing the need for multiple function names.

```
#include <iostream>
using namespace std;
// Function overloading
int add(int a, int b) {
 return a + b;
}
double add(double a, double b) {
 return a + b;
}
int main() {
 cout << "Sum of integers: " << add(3, 5) << endl;
 cout << "Sum of doubles: " << add(3.5, 2.5) << endl;
 return 0;
}
```

- ✓ Run-time Polymorphism (Function Overriding):

Run-time polymorphism, also known as dynamic polymorphism, is achieved through function overriding.

Function overriding occurs when a derived class provides a specific implementation of a method that is already defined in its base class.

The appropriate function to be called is determined by the object's runtime type, allowing for dynamic binding of method calls.

This enables code to be written in terms of abstract base classes and then specialized behavior to be implemented in derived classes.

Example:

```
#include <iostream>
using namespace std;
// Base class
class Shape {
public:
 virtual void draw() {
 cout << "Drawing a shape" << endl;
 }
};
// Derived class
class Circle : public Shape {
public:
 void draw() override {
 cout << "Drawing a circle" << endl;
 }
}
int main() {
 Shape* shape = new Circle(); // Polymorphic behavior
 shape->draw(); // Calls Circle's draw() method
 delete shape;
 return 0;
```

Key points:

Compile-Time (Static) Polymorphism also known as early binding or method overloading.

- Resolved during compile time based on the number and types of arguments in the method call.
- Examples include function overloading and operator overloading.
- It is efficient as the binding of function calls is done during compile time.

Runtime (Dynamic) Polymorphism also known as late binding or method overriding.

- Resolved during runtime based on the actual object type.
- Achieved through inheritance and virtual functions (in C++) or interfaces (in languages like Java and C#).
- Allows a derived class to provide a specific implementation of a method that is already defined in its base class.
- It offers more flexibility as the decision of which function to call is made at runtime.
- In addition to these two main types, there's also a concept called Ad hoc polymorphism, which is not a distinct type of polymorphism but rather a characteristic exhibited by both compile

## FUNCTION OVERLOADING

Function overloading is a form of compile-time (static) polymorphism that allows multiple functions in the same scope to have the same name but with different parameters. The compiler determines which function to call based on the number and types of arguments passed to it. Here's how it works:

- ✓ Same Function Name: In function overloading, you define multiple functions with the same name.
- ✓ Different Parameters: Each of these functions must have a unique parameter list. The parameters can differ in number, type, or both.
- ✓ Compile-Time Resolution: When a function call is made, the compiler determines the appropriate function to call based on the arguments provided and their types.

Here's a simple example in C++ to illustrate function overloading:

```
#include <iostream>
using namespace std;
int square(int x) {// Function to calculate the square of an integer
 return x * x;  }
double square(double x) {// Function to calculate the square of a double
 return x * x; }
int main() {
 int intResult = square(5); // Calls int version of square
 double doubleResult = square(2.5); // Calls double version of square
 cout << "Square of 5 is: " << intResult << endl;
 cout << "Square of 2.5 is: " << doubleResult << endl;
 return 0; }
```

In this example:

There are two functions named square, one taking an integer parameter and the other taking a double parameter.

When calling square(5), the integer version of square is called, and when calling square(2.5), the double version of square is called.

The decision of which function to call is made at compile-time based on the argument types.

Function overloading provides a way to define multiple functions with the same name but different behaviors, improving code readability and flexibility.

## OVERLOADING OPERATOR:

Operator overloading is a feature in many programming languages that allows you to redefine the behavior of operators for user-defined types. By overloading operators, you can specify how operators should behave when applied to objects of your custom classes. This enables you to use operators in a natural and intuitive way with your own data types.

Here's how it works:

- ✓ Operator Functions: You define special member functions called operator functions that define the behavior of operators for your class.
- ✓ Syntax: Operator functions have a specific syntax, using the operator keyword followed by the operator symbol you want to overload.
- ✓ Custom Behavior: Inside the operator function, you specify the custom behavior for the operator when applied to objects of your class.

Here's a simple example in C++ to illustrate operator overloading:

```cpp
#include <iostream>
using namespace std;
// Class representing a complex number
class Complex {
private:
 double real;
 double imag;
public:
 Complex(double r = 0.0, double i = 0.0) : real(r), imag(i) {}// Constructor
 Complex operator+(const Complex& other) const {// Overloading addition operator '+'
 return Complex(real + other.real, imag + other.imag); }
 Complex operator-(const Complex& other) const {  // Overloading subtraction operator '-'
 return Complex(real - other.real, imag - other.imag); }
 friend ostream& operator<<(ostream& os, const Complex& c) { // Overloading output stream operator
 os << "(" << c.real << " + " << c.imag << "i)";
 return os; } };
int main() {
 Complex c1(2.0, 3.0);// Creating complex numbers
 Complex c2(1.0, 2.0);
 Complex sum = c1 + c2; // Using overloaded operators
 Complex diff = c1 - c2;
 cout << "Sum: " << sum << endl;// Printing results
 cout << "Difference: " << diff << endl;
 return 0;
}
```

In this example:

We define a Complex class to represent complex numbers.

We overload the addition operator + and subtraction operator - for adding and subtracting complex numbers.

We also overload the output stream operator << to enable printing of complex numbers using cout.

Operator overloading allows you to use operators in a natural way with your custom types, enhancing code readability and expressiveness. However, it's important to use operator overloading judiciously to maintain code clarity and avoid ambiguity.

**FRIEND FUNCTION:**

Friend functions in C++ provide access to the private and protected members of a class from outside the class scope. They are declared using the friend keyword within the class declaration. Here's how friend functions work:

- ✓ Accessing Private Members: Normally, only member functions of a class have access to its private and protected members. However, by declaring a function as a friend, you grant it access to these members.
- ✓ Declared Inside Class: Friend functions are declared inside the class, but they are not members of the class. They are standalone functions that have access to the private and protected members of the class.
- ✓ Not Inherited or Overloaded: Friend functions are not inherited by derived classes, nor can they be overridden or overloaded.
- ✓ No Access to 'this' Pointer: Friend functions do not have a 'this' pointer, as they are not members of any object. They act more like ordinary functions, but with special privileges to access private and protected members of the class they are declared as friends of.

Here's an example to illustrate friend functions:

```
#include <iostream>
using namespace std;
class B;// Forward declaration of class B
class A {// Class A with a friend function
private:
 int dataA;
public:
 A(int x) : dataA(x) {}
 friend void showData(A a, B b); };// Declaring friend function
class B {// Class B
private:
 int dataB;
public:
 B(int y) : dataB(y) {}
 friend void showData(A a, B b);};// Friend function defined outside class scope
void showData(A a, B b) {// Definition of friend function
 cout << "Data of A: " << a.dataA << endl;
 cout << "Data of B: " << b.dataB << endl;  }
int main() {
 A objA(5);
 B objB(10);
 showData(objA, objB); // Accessing private members of A and B using the
friend function
 return 0;  }
```

In this example:

Class A and B have private data members dataA and dataB respectively.

The function showData is declared as a friend of both classes A and B.

Inside showData, we can directly access the private members of both A and B objects passed to it.

Friend functions are useful in scenarios where you need to allow certain functions or classes special access to the private members of a class without making those members public. However, they should be used with caution to avoid violating encapsulation principles.

**CONSTANT FUNCTION:**

Constant functions, also known as const member functions in C++, are member functions of a class that promise not to modify the state of the object on which they are called. These functions are declared with the const keyword after the function prototype in the class declaration. Here's how constant functions work:

- ✓ Object State Preservation: Constant functions guarantee that they will not modify the state of the object. This means they can only read data members and call other constant member functions.
- ✓ Declared with 'const': Constant functions are declared with the const keyword at the end of the function prototype in the class declaration. This informs the compiler that the function does not modify the object's state.
- ✓ Compiler-Enforced: If a constant member function attempts to modify a nonstatic data member of the class or call a non-constant member function, the compiler will generate an error.

- ✓ Enabling Operations on Constant Objects: Constant functions allow you to invoke member functions on constant objects, ensuring that the object's state remains unchanged.

Here's an example to illustrate constant functions in C++:

```
#include <iostream>
using namespace std;
class MyClass {
private:
 int value;
public:
 MyClass(int x) : value(x) {}
 int getValue() const { // Constant member function
member function// value = 10; // Error: Cannot modify non-static member within const
 return value; }
 void setValue(int x) {// Non-constant member function
 value = x; } };
int main() {
 const MyClass obj(5); // Constant object
 cout << "Value: " << obj.getValue() << endl; // Invoking constant member
function
constant object // obj.setValue(10); // Error: Cannot call non-constant member function on
 return 0;
}
```

In this example:

The member function getValue() is declared as const, indicating that it will not modify the object's state.

Attempting to modify the value member inside the getValue() function would result in a compiler error.

In the main() function, we create a constant object obj, and we can only call constant member functions on it. Attempting to call setValue() on obj would result in a compiler error.

Constant member functions are essential for ensuring data integrity and enabling operations on constant objects in C++. They play a crucial role in enforcing the concept of const-correctness, which is vital for writing robust and maintainable code.

## CONCEPT OF RUNTIME POLYMORPHISM:

Runtime polymorphism, also known as dynamic polymorphism, is a core concept in object-oriented programming (OOP) that allows a method or function to behave differently based on the actual object type at runtime. It enables you to invoke a method on an object without knowing its specific type at compile time. This is achieved through inheritance and virtual functions (or abstract methods in languages like Java and C#).

Here's how runtime polymorphism works:

- ✓ Inheritance: Runtime polymorphism relies on the inheritance hierarchy. You have a base class (also called a superclass or parent class) and one or more derived classes (also called subclasses or child classes) that inherit from the base class.
- ✓ Virtual Functions: In the base class, you can declare one or more member functions as virtual using the virtual keyword. These virtual functions can be overridden in derived classes.
- ✓ Late Binding: When you call a virtual function on a base class pointer or reference that points to a derived class object, the actual implementation of the function is determined at runtime. This process is known as late binding or dynamic binding.
- ✓ Override: Derived classes provide their own implementation (override) for the virtual functions declared in the base class. This allows you to customize the behavior of the function for each derived class.

Here's a simple example in C++ to illustrate runtime polymorphism using virtual functions: #include <iostream> using namespace std;

```
#include <iostream>
using namespace std;
class Animal {
public:
 virtual void makeSound() { // Virtual function
 cout << "Animal makes a sound" << endl;  }  };
class Dog : public Animal { // Derived class
public:
 void makeSound() override {  // Override the virtual function
 cout << "Dog barks" << endl; } };
class Cat : public Animal {// Derived class
public:
 void makeSound() override { // Override the virtual function
 cout << "Cat meows" << endl; } };
int main() {
 Animal* ptr1 = new Dog(); // Base class pointer pointing to derived class objects
 Animal* ptr2 = new Cat();
 ptr1->makeSound(); // Output: Dog barks/ /Call virtual function
 ptr2->makeSound(); // Output: Cat meows
 delete ptr1;
 delete ptr2;
 return 0;
}
```

In this example:

Animal is the base class with a virtual function makeSound(). Dog and Cat are derived classes that override the makeSound() function. We create base class pointers (ptr1 and ptr2) pointing to objects of the derived classes.

When we call makeSound() through these pointers, the appropriate overridden version of the function is called based on the actual object type (Dog or Cat), demonstrating runtime polymorphism.

Runtime polymorphism is a powerful mechanism that enables flexible and extensible code by allowing different implementations of methods to be invoked based on the runtime type of objects.

**VIRTUALFUNCTION AND PUREVIRTUAL FUNCTION:**

Virtual functions and pure virtual functions are key features of object-oriented programming languages like C++ and Java, facilitating polymorphism and abstract classes. Here's an overview of each:

*Virtual Functions:*

- ✓ Definition: A virtual function is a member function declared within a base class that is expected to be redefined in derived classes. It allows the function to be overridden in derived classes, enabling polymorphic behavior.
- ✓ Syntax: In C++, you declare a virtual function by using the virtual keyword in the base class. Derived classes can then override this function using the same signature.
- ✓ Late Binding: Virtual functions are resolved at runtime (late binding or dynamic binding). This means that the appropriate function to call is determined based on the actual type of the object at runtime.

```
class Base {
public:
 virtual void someFunction() {
 cout << "Base class function" << endl;
 }
};
class Derived : public Base {
public:
 void someFunction() override {
 cout << "Derived class function" << endl;
 }
};
int main() {
 Base* ptr = new Derived();
 ptr->someFunction(); // Output: "Derived class function"
 delete ptr;
 return 0;
```

*Pure Virtual Functions:*

- ✓ Definition: A pure virtual function (also known as an abstract method) is a virtual function that is declared in a base class but has no implementation. It serves as a placeholder for functions that must be implemented by derived classes.
- ✓ Syntax: In C++, you declare a pure virtual function by appending = 0 to its declaration in the base class. This indicates that the function has no implementation and must be overridden by derived classes.
- ✓ Abstract Classes: A class containing at least one pure virtual function is called an abstract class. You cannot create objects of an abstract class, but you can have pointers and references of the abstract class type.

Virtual functions and pure virtual functions are essential for achieving polymorphism and abstraction in object-oriented programming. They allow for flexible and extensible code by enabling derived classes to provide custom implementations for base class functions.

```
class AbstractBase {
public:
 virtual void pureVirtualFunction() = 0; // Pure virtual function
};
class Derived : public AbstractBase {
public:
 void pureVirtualFunction() override {
 cout << "Implemented pure virtual function" << endl;
 }
};
int main() {
 AbstractBase* ptr = new Derived();
 ptr->pureVirtualFunction(); // Output: "Implemented pure virtual function"
 delete ptr;
 return 0;
}
```

***Dynamic_cast, static_cast,const_cast, reinterpret_cast:***

dynamic_cast, static_cast, const_cast, and reinterpret_cast are four types of casting operators available in C++. Each serves a different purpose and has different behavior. Here's an overview of each:

1. Dynamic_cast:

- ✓ Purpose: Used for safe casting of pointers or references to polymorphic types (i.e., classes with virtual functions) during runtime.
- ✓ Behavior: Performs a runtime check to ensure that the conversion is valid. It only works with pointers and references and can be used to convert pointers or references of a base class to pointers or references of a derived class, and vice versa.
- ✓ Use Cases: Commonly used for downcasting (from base class to derived class) in inheritance hierarchies.

Syntax:

Derived*derivedPtr=dynamic_cast<Derived*>(basePtr);
if (derivedPtr) { // Conversion successful }
else { // Conversion failed }

2. Static_cast:

- ✓ Purpose: Used for general type conversions that do not require runtime type checking, such as converting between related types or performing implicit conversions.
- ✓ Behavior: Performs a compile-time check for type safety. It is less safe than dynamic_cast as it does not perform any runtime checks.
- ✓ Use Cases: Converting between built-in types, casting to related types (e.g., derived to base), and performing static type conversions.

Syntax:

int intValue = 10;

double doubleValue = static_cast<double>(intValue);

3. Const_cast:

- ✓ Purpose: Used to add or remove const-ness or volatile-ness from a variable.
- ✓ Behavior: Performs a simple cast that adjusts the const or volatile qualifiers of a pointer or reference without changing the underlying object.
- ✓ Use Cases: Modifying the const-ness or volatile-ness of a variable when necessary.

Syntax:

```
const int* ptr = &value;
int* nonConstPtr = const_cast<int*>(ptr);
```

4. Reinterpret_cast:

- ✓ Purpose: Used for low-level conversions between pointer types or between pointer and integral types.
- ✓ Behavior: Performs a simple bit-wise reinterpretation of the pointer or value. It is highly implementation-dependent and should be used with caution.
- ✓ Use Cases: Converting between unrelated pointer types, performing type punning, or dealing with low-level memory manipulation.

Syntax:

```
int intValue = 10;

double* doublePtr = reinterpret_cast<double*>(&intValue);
```

It's important to use the appropriate casting operator for the specific task at hand, considering safety, readability, and maintainability. Misusing casting operators can lead to undefined behavior and runtime errors.

**********

## Constructor and destructor:

In C++, constructors and destructors are special member functions of a class that are automatically called when objects of the class are created and destroyed, respectively. Here's an explanation of constructors and destructors:

- ✓ **Constructor:**

A constructor is a member function of a class that is automatically called when an object of the class is created.

It initializes the object's data members and performs any necessary setup operations.

Constructors have the same name as the class and no return type (not even void).

Constructors can be overloaded, allowing multiple constructors with different parameter lists to exist.

If a class does not explicitly define a constructor, the compiler provides a default constructor, which initializes data members to default values.

CONSTRUCTOR CODE ex:

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
 // Default constructor
 MyClass() {
 cout << "Constructor called" << endl;
 }
 // Parameterized constructor
 MyClass(int value) {
 cout << "Parameterized constructor called with value: " << value << endl;
 }
};
int main() {
 MyClass obj1; // Calls default constructor
 MyClass obj2(10); // Calls parameterized constructor
 return 0;
}
```

***PARAMETRIZED CONSTRUCTOR:***

A parameterized constructor in C++ is a constructor that accepts parameters, allowing you to initialize the object's data members with custom values at the time of object creation. This provides flexibility in object initialization and allows you to create objects with different initial states based on the values passed to the constructor.

Here are some key points about parameterized constructors:

- ✓ Accept Parameters: Parameterized constructors accept parameters just like regular functions, allowing you to pass values to initialize the object's data members.

- ✓ Initialization: Inside the parameterized constructor, you can use the passed parameters to initialize the object's data members based on the provided values.
- ✓ Usage: Parameterized constructors are useful when you want to create objects with specific initial states or when you need to initialize object data members with values provided by the user or calculated at runtime.
- ✓ Default Arguments: You can also provide default values for parameters in parameterized constructors, allowing you to create objects without explicitly providing values for all parameters.

Here's an example of a class with a parameterized constructor:

```cpp
#include <iostream>
using namespace std;
class Rectangle {
private:
 int length;
 int width;
public:
 // Parameterized constructor
 Rectangle(int l, int w) {
 length = l;
 width = w;
 }
 // Method to calculate area
 int area() {
 return length * width;
 }
};
int main() {
 // Creating an object of the class with a parameterized constructor
 Rectangle rect(5, 3);
 // Calling the area method to calculate the area of the rectangle
 cout << "Area of the rectangle: " << rect.area() << endl;
 return 0;
}
```

***MULTIPLE CONSTRUCTOR IN CLASS-:***

Certainly! In C++, a class can have multiple constructors, each with a different parameter list. This allows you to create objects using different initialization methods or with different sets of parameters. This feature is known as constructor overloading.

Here's an example demonstrating a class with multiple constructors:

```cpp
#include <iostream>
using namespace std;
class MyClass {
private:
 int value;
public:
 // Default constructor
 MyClass() {
 value = 0; // Default initialization
 }
 // Parameterized constructor with one parameter
 MyClass(int val) {
 value = val;
 }
 MyClass(int val1, int val2)// Parameterized constructor with two parameters
 {value = val1 + val2; // Initialization based on sum of parameters
 }
 void display() {
 cout << "Value: " << value << endl;}
};
int main() {
 // Creating objects using different constructors
 MyClass obj1; // Default constructor
 MyClass obj2(10); // Constructor with one parameter
 MyClass obj3(5, 7); // Constructor with two parameters
 // Displaying values of objects
 obj1.display(); // Value: 0
 obj2.display(); // Value: 10
 obj3.display(); // Value: 12
 return 0;
}
```

In this example:

The MyClass class has three constructors: a default constructor, a parameterized constructor with one parameter, and a parameterized constructor with two parameters.

Each constructor initializes the value data member of the object using different initialization methods.

In the main() function, objects obj1, obj2, and obj3 are created using different constructors, and their values are displayed using the display() method.

The output demonstrates that each object is initialized based on the constructor used to create it.

***DYNAMIC INITIALIZATION OF OBJECT:***

Dynamic initialization of objects in C++ involves creating objects dynamically at runtime using pointers and the new operator. This allows you to allocate memory for objects on the heap and initialize them as needed.

Here's how you can dynamically initialize objects in C++:

- Allocate Memory: Use the new operator to allocate memory for the object on the heap. This returns a pointer to the newly allocated memory.
- Initialize Object: Use the pointer returned by new to initialize the object using constructor(s) as required.
- Use Object: Once the object is initialized, you can use it just like any other object created statically.

- Deallocate Memory: After you are done using the dynamically allocated object, use the delete operator to free the memory allocated for the object. Here's an example demonstrating dynamic initialization of objects:

```
#include <iostream>
using namespace std;
class MyClass {
private:
 int value;
public:
 // Constructor
 MyClass(int val) {
 value = val;
 }
 // Method to display value
 void display() {
 cout << "Value: " << value << endl;
 }
};
int main() {
 // Dynamic initialization of object
 MyClass *ptr = new MyClass(10);
 // Use the object
 cout << "Dynamic object created." << endl;
 ptr->display();
 // Deallocate memory
 delete ptr;
 cout << "Dynamic object deleted." << endl;
 return 0;
}
```

In this example:

We dynamically initialize an object of the MyClass class using new, passing the value 10 to its constructor.

We use the -> operator to access the display() method of the dynamically allocated object through the pointer ptr..After using the object, we deallocate the memory allocated for the object using the delete operator to avoid memory leaks.

***COPY CONSTRUCTOR:***

A copy constructor in C++ is a special constructor that initializes a new object as a copy of an existing object of the same class. It is invoked automatically when:

- An object is passed by value as an argument to a function.
- An object is returned by value from a function.
- An object is initialized with another object of the same class.
- An object is assigned the value of another object of the same class.
- The syntax for a copy constructor is similar to that of a regular constructor, but it takes a reference to an object of the same class as its parameter.

Here's a basic example of a copy constructor:

```
#include <iostream>
using namespace std;
class MyClass {
private:
 int value;
public:
 // Default constructor
 MyClass() {
 value = 0;}
 // Parameterized constructor
 MyClass(int val) {
 value = val;}
 // Copy constructor
 MyClass(const MyClass &other) {
 value = other.value; // Copy the value from 'other' object
 }
 // Method to display value
 void display() {
 cout << "Value: " << value << endl;  }  };
int main() {
 // Creating an object using the default constructor
 MyClass obj1;
 obj1.display(); // Value: 0
 // Creating another object using the parameterized constructor
 MyClass obj2(10);
 obj2.display(); // Value: 10
 // Creating a third object using the copy constructor
 MyClass obj3 = obj2;
 obj3.display(); // Value: 10
 return 0;  }
```

In this example:

The copy constructor MyClass(const MyClass &other) takes a reference to another object of the same class as its parameter.

When the statement MyClass obj3 = obj2; is executed, the copy constructor is automatically invoked to initialize obj3 as a copy of obj2.

After execution, obj3 will have the same value as obj2, demonstrating the behavior of the copy constructor.

- ✓ **Destructor:**

A destructor is a member function of a class that is automatically called when an object of the class is destroyed. It performs cleanup operations, such as releasing dynamically allocated memory or closing files.

Here are some key points about destructors:

- Name: Destructors have the same name as the class preceded by a tilde (~) and no parameters or return type.
- Automatic Invocation: Destructors are automatically invoked when an object of the class goes out of scope, or when delete is called explicitly on a dynamically allocated object.

- Order of Execution: Destructors are called in the reverse order of the constructors' execution. This means that the destructor of a base class is called before the destructor of a derived class.
- Resource Cleanup: Destructors are typically used to release resources acquired by the object during its lifetime. This includes freeing memory allocated on the heap, closing files, releasing locks, and performing other cleanup tasks.
- Default Destructor: If a class does not provide any destructor, the compiler generates a default destructor automatically. This default destructor does not perform any cleanup operations.

Basic example:

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
 // Constructor
 MyClass() {
 cout << "Constructor called" << endl;
 }
 // Destructor
 ~MyClass() {
 cout << "Destructor called" << endl;
 }
};
int main() {
 {
 MyClass obj; // Calls constructor
 } // Calls destructor when obj goes out of scope
 return 0;
}
```

**********

## NAMESPACES:

In C++, namespaces are used to organize code into logical groups and prevent naming conflicts between different parts of a program. A namespace defines a scope within which names for variables, functions, classes, and other identifiers are unique.

Here's an explanation of namespaces in C++:

- ✓ **Defining a Namespace:**

A namespace is defined using the namespace keyword followed by the namespace name and a block of code enclosed in curly braces {}.

All code within the namespace block belongs to that namespace.

Example:

```
// Defining a namespace named 'MyNamespace'
namespace MyNamespace {
    // Code belonging to the 'MyNamespace' namespace
    int myFunction(int x) {
        return x * x;
    }
}
```

- ✓ **Accessing Namespaces:**

To access names within a namespace, you can use the scope resolution operator ::.

The scope resolution operator allows you to specify the namespace in which a name is defined.

Example:

```
// Accessing the 'myFunction' function defined in the 'MyNamespace' namespace
 int result = MyNamespace::myFunction(5);
```

- ✓ **Using Directive and Using Declaration:**

The using directive and using declaration are used to bring names from a namespace into the current scope, making them accessible without using the scope resolution operator.

```
// Using directive
using namespace MyNamespace;
 // Accessing 'myFunction' directly without using the scope resolution operator
int result = myFunction(5);
```

- ✓ **Nested and Anonymous Namespace**:

Namespaces can be nested within other namespaces to create a hierarchical organization of code.

Anonymous namespaces are used to define names that are accessible only within the translation unit (source file), providing a form of internal linkage.

Example:

```
// Nested namespaces
namespace OuterNamespace {
 namespace InnerNamespace {
 int innerFunction(int y) {
 return y * y;
 }
 }
}
// Anonymous namespace
namespace {
 int internalVariable = 10;
}
```

- ✓ **Namespace Aliases:**

Namespace aliases allow you to define an alternative name for a namespace, making it easier to refer to.

Example:

```
// Namespace alias
namespace MN = MyNamespace;


// Accessing 'myFunction' using the namespace alias
int result = MN::myFunction(5);
```

**********

## INTERFACES:

In C++, interfaces are simulated using abstract base classes with pure virtual functions. While C++ does not have a native interface construct like Java or C#, you can achieve similar functionality by defining an abstract class with pure virtual functions. Here's how interfaces can be implemented in C++:

- ✓ Abstract Base Class: Define an abstract base class that contains only pure virtual functions. A pure virtual function is a virtual function that is declared with = 0 at the end of its declaration, indicating that it has no implementation in the base class.
- ✓ Implementation by Derived Classes: Any class that wants to implement the interface must derive from the abstract base class and provide concrete implementations for all the pure virtual functions declared in the interface.

- ✓ Polymorphism: You can then use polymorphism to treat objects of derived classes as instances of the interface type, enabling code reuse and flexibility.

Here's an example demonstrating how to create an interface in C++:

```
#include <iostream>
using namespace std;
class Animal {// Interface definition
public:
 virtual void eat() const = 0;// Pure virtual functions
 virtual void sleep() const = 0;  };
class Dog : public Animal {  // Class implementing the interface
public:
 void eat() const override { // Concrete implementations for pure virtual functions
 cout << "Dog eats" << endl;  }
 void sleep() const override {
 cout << "Dog sleeps" << endl;  }  };
int main() {
 Animal* myPet = new Dog();//POLYMORPHISM
 myPet->eat();
 myPet->sleep();
 delete myPet;
 return 0;
}
```

In this example:

Animal is an abstract base class representing the interface, containing pure virtual functions eat() and sleep().

Dog is a class that implements the Animal interface by providing concrete implementations for the pure virtual functions.

In the main() function, polymorphism is demonstrated by creating an Animal pointer pointing to a Dog object. This allows us to treat the Dog object as an Animal object.

While C++ does not have a dedicated keyword for interfaces, you can achieve interface-like behavior using abstract base classes and pure virtual functions, allowing you to design flexible and extensible code.

## ABSTRACT CLASS:

An abstract class in C++ is a class that cannot be instantiated on its own and serves as a blueprint for other classes. It may contain both regular member functions with implementations and pure virtual functions. Abstract classes are used to define interfaces and provide a common base for derived classes to inherit from.

Here are the key characteristics and uses of abstract classes:

- ✓ Cannot be Instantiated: Abstract classes cannot be instantiated, meaning you cannot create objects of an abstract class directly. Attempting to do so will result in a compilation error.

- ✓ Contain Pure Virtual Functions: Abstract classes typically contain at least one pure virtual function, indicated by appending = 0 to the function declaration. Pure virtual functions have no implementation in the abstract class and must be overridden by derived classes.
- ✓ May Contain Regular Member Functions: Abstract classes may also contain regular member functions with implementations. These functions provide common functionality shared by all derived classes.
- ✓ Derived Class Implementation: Any class that inherits from an abstract class must provide concrete implementations for all pure virtual functions declared in the abstract class. Otherwise, the derived class will also be considered abstract.
- ✓ Provides Interface: Abstract classes often serve as interfaces, defining a contract that derived classes must adhere to. They specify a set of methods that derived classes must implement, ensuring consistency across different implementations.

Here's an example demonstrating how to create an abstract class in C++:

```cpp
#include <iostream>
using namespace std;
// Abstract class definition
class Shape {
public:
 // Pure virtual function
 virtual void draw() const = 0;
 // Regular member function with implementation
 void displayInfo() const {
 cout << "This is a shape." << endl;
 }
};
// Derived class implementing the abstract class
class Circle : public Shape {
public:
 // Concrete implementation of pure virtual function
 void draw() const override {
 cout << "Drawing a circle." << endl;
 }
};
int main() {
 // Shape shape; // Error: Cannot instantiate abstract class
 Circle circle;
 circle.draw(); // Output: "Drawing a circle."
 circle.displayInfo(); // Output: "This is a shape."
 return 0;
}
```

In this example:

Shape is an abstract class containing a pure virtual function draw() and a regular member function displayInfo().
Circle is a class that inherits from Shape and provides a concrete implementation for the pure virtual function draw().

You cannot create objects of the Shape class directly due to its abstract nature, but you can create objects of derived classes like Circle.

# Exception Handling

**Introduction :**

- Exception are some unpredictable circumstances, when our program terminates suddenly with some errors or issues.
- It provide us the facilities to handed the exception so that our program keeps running.
- errors can broadly be categorized into three main types:

- **Compile-time errors**: These errors occur during the compilation of the program.
- **Run-time errors**: These errors occur while the program is running.
- **Logic errors**: These errors occur when the code compiles and runs without throwing any errors or exceptions, but it does not produce the expected result.

**Keywords in exception handling:**

- **Throwing an exception**: When a critical error or exceptional condition is encountered within your code, you can use the **throw** keyword to raise an exception.
- **Catching an exception**: When an exception is thrown within a **try** block, after that the program looks for a matching catch block, when will **catch** block is found, the code within that block is executed. You can have multiple **catch** blocks to handle different types of exceptions.

**Here is a simple code using try and catch block:**

```
 9  #include <iostream>
10  using namespace std;
11  int main(){
12      int a,b,c;
13      cout<<"\n enter two number";
14      cin>>a>>b;
15      try{
16          if(b!=0){
17              c=a/b;
18              cout<<"\n result:"<<c;
19          }
20          else{
21              throw(b);
22          }
23      }
24      catch(int b){
25          cout<<"divided by "<<b<<"can't be acceptable";
26      }
27      return 0;
28  }
```

- **Re-throwing an exception:** This is useful when you want to handle certain exceptions locally but still want higher-level code to be aware of the error. You can re-throw an exception using the **throw;** statement without any arguments.

```
try {
    // Code that may throw an exception
} catch (const SomeException& e) {
    // Handle the exception locally
    // Optionally, re-throw the exception
    throw; // Re-throw the same exception
}
```

**Exception Specifications:**

Advanced exception handling techniques such as nested exception handling, handling exceptions across thread boundaries, and exception propagation in asynchronous programming

- **Nested exception handling:** process to placing one or more try-catch blocks within another try block's catch block.
  can be particularly useful when dealing with functions or code sections where multiple operations

```
#include <iostream>
using namespace std;
void innerFunction() {
    throw "Inner function error";
}
void outerFunction() {
    try {
        innerFunction();
    } catch (const char* innerException) {
        cout<< "Caught inner exception: " << innerException <<endl;
    }
}
int main() {
    try {
        outerFunction();
    } catch (const char* outerException) {
        cout << "Caught outer exception: " << outerException << endl;
    }
    return 0;
}
```

In this example:

- The **innerFunction()** throws an exception of type **const char***.
- Inside the **outerFunction()**, the call to **innerFunction()** is wrapped within a try-catch block.

- If an exception is thrown within **innerFunction()**, it is caught in the catch block of **outerFunction()**.
- Inside the catch block of **outerFunction()**, you can perform additional error handling or re-throw the caught exception.
- The exception is then caught again in the catch block of **main()**.

## Managing Console I/O Operations:

1. **Introduction to Managing Console I/O Operations:**
    - Console Input/Output (I/O) operations are fundamental for interacting with users through a command-line interface.
    - In C++, console I/O operations are facilitated through streams.
    - Understanding how to manage console I/O is crucial for building interactive and user-friendly applications.
2. **C++ Streams:**
    - In C++, a stream is an abstraction that represents a sequence of bytes, facilitating input and output operations.
    - Streams can be either input streams (for reading data) or output streams (for writing data).
    - The primary standard streams in C++ are **cin** (standard input) and **cout** (standard output).
3. **C++ Stream Classes:**
    - C++ provides a set of stream classes to work with different types of I/O operations.
    - Common stream classes include **iostream**, **ifstream**, and **ofstream**.
    - **iostream** handles both input and output operations, while **ifstream** and **ofstream** are specifically for file input and output operations.

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

int main() {
    // Example of ifstream and ofstream
    ifstream inputFile("input.txt");
    ofstream outputFile("output.txt");

    int num;
    inputFile >> num;
    outputFile << "The number is: " << num << endl;

    inputFile.close();
    outputFile.close();
    return 0;
}
```

4. **Unformatted I/O Operations**:
    - Unformatted I/O operations involve reading or writing data without any specific formatting applied.
    - Examples include using **cin** for input and **cout** for output, which directly read and write data without any formatting.
5. **Formatted I/O Operations**:
    - Formatted I/O operations involve reading or writing data with specific formatting rules applied.
    - This can include specifying the width, precision, and formatting flags for data output using manipulators.
6. **Managing Output with Manipulators**:
    - Manipulators are functions or objects used to modify the behavior of output streams.
    - They allow for precise control over how data is formatted and presented in the output.
    - Examples of manipulators include **setw**, **setprecision**, **left**, **right**, etc., which control the width, precision, alignment, and other formatting aspects of output.

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

int main() {
    // Example of using manipulators
    double num = 123.456789;
    cout << "Default formatting: " << num << endl;
    cout << "Fixed-point notation: " << fixed << num << endl;
    cout << "Scientific notation: " << scientific << num << endl;
    return 0;
}
```

# File Handling:

1. **Definition of File:**
    - A file is a collection of related data stored on a secondary storage device, such as a hard drive or SSD.
    - Files provide a persistent means of storing data that can be accessed by programs.
    - In C++, file handling involves operations such as creating, opening, reading from, writing to, and closing files.
2. **File Handling in C++:**
    - C++ provides a rich set of features for file handling through the standard input/output library (**<fstream>**).
    - File handling involves the use of stream classes such as **ifstream** for reading from files and **ofstream** for writing to files.

File handling in C++ allows you to read from and write to files. Here are some key points:

1. ***File Streams*:** C++ provides classes like ifstream for reading from files and ofstream for writing to files. These classes are derived from istream and ostream respectively.

2. ***Opening and Closing Files*:** Before performing any operations on a file, you need to open it using the open() method. After finishing the operations, close the file using the close() method.

3. ***Reading from Files*:** Use input file streams (ifstream) along with extraction operator (>>) to read data from files. You can read data of different types like integers, strings, etc.

4**. *Writing to Files*:** Use output file streams (ofstream) along with insertion operator (<<) to write data to files. You can write data of different types like integers, strings, etc.

5. ***Error Handling*:** Always check if a file is successfully opened before performing any operations. Also, handle errors that might occur during file operations.

6**. *File Modes*:** You can specify different modes while opening a file, such as ios::in for input (reading), ios::out for output (writing), ios::app for append mode, ios::binary for binary files, etc.

Here's a basic example:

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

int main() {
    // Writing to a file
    ofstream outfile("example.txt");
    if (outfile.is_open()) {
        outfile << "This is a test." << endl;
        outfile.close();
    } else {
        cout << "Unable to open file for writing." << endl;
    }

    // Reading from a file
    ifstream infile("example.txt");
    if (infile.is_open()) {
        string line;
        while (getline(infile, line)) {
            cout << line << endl;
        }
        infile.close();
    } else {
        cout << "Unable to open file for reading." << endl;
    }

    return 0;
}
```

//This code writes "This is a test." to a file named example.txt, and then reads it back and prints its content to the console.

## Templets:

1. **Introduction to Templates**:
    - Templates are a powerful feature in C++ that allows for generic programming.
    - They enable the creation of functions and classes that work with any data type.
    - Templates provide a way to write reusable code by allowing the definition of functions and classes that work with generic types.
2. **Function Templates**:
    - Function templates allow you to write a single function definition that can work with different data types.
    - The syntax for declaring a function template involves using the **template** keyword followed by the template parameter list enclosed in angle brackets <>.

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

// Function template to find the maximum of two values
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max of 5 and 10: " << max(5, 10) << endl; // Invoking with integers
    cout << "Max of 3.5 and 2.8: " << max(3.5, 2.8) << endl; // Invoking with doubles
    return 0;
}
```

3. **Class Templates**:
    - Class templates enable the creation of generic classes that can work with any data type.
    - Similar to function templates, class templates are declared using the **template** keyword followed by the template parameter list.

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

// Class template for a generic Pair
template <typename T>
class Pair {
private:
    T first;
    T second;
public:
    Pair(T a, T b) : first(a), second(b) {}
    T getFirst() { return first; }
    T getSecond() { return second; }
};

int main() {
    Pair<int> intPair(5, 10); // Instantiating with int type
    Pair<double> doublePair(3.5, 2.8); // Instantiating with double type

    cout << "First value of intPair: " << intPair.getFirst() << endl;
    cout << "Second value of doublePair: " << doublePair.getSecond() << endl;

    return 0;
}
```

## STL(Standard Template Library)

1. **Introduction to C++ Standard Library (STL)**:
    - The C++ Standard Library provides a rich set of libraries that offer various functionalities for tasks such as input/output operations, string manipulation, data structures, algorithms, and more.
    - One of the key components of the C++ Standard Library is the Standard Template Library (STL), which provides generic classes and functions that implement several popular data structures and algorithms.
    - The STL consists of containers, iterators, algorithms, and function objects, offering a high level of abstraction and reusability in C++ programming.

2. **Introduction to RTTI (Run-Time Type Information) in C++**:
    - RTTI (Run-Time Type Information) is a mechanism in C++ that allows you to query the type of an object at runtime.

- It enables dynamic casting and typeid operator, which provides information about the type of an object.
- RTTI is particularly useful in scenarios where the exact type of an object is unknown at compile time but needs to be determined at runtime.

Now, let's see some coding examples:

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

// Base class
class Base {
public:
    virtual void display() {
        cout << "Base class" << endl;
    }
};

// Derived class
class Derived : public Base {
public:
    void display() override {
        cout << "Derived class" << endl;
    }
};

int main() {
    // RTTI example
    Base* basePtr = new Derived();

    // Dynamic casting to Derived class
    Derived* derivedPtr = dynamic_cast<Derived*>(basePtr);
    if (derivedPtr) {
        derivedPtr->display(); // Output: Derived class
    } else {
        cout << "Dynamic cast failed" << endl;
    }

    // Using typeid to get type information
    cout << "Type of derivedPtr: " << typeid(*derivedPtr).name() << endl;

    delete basePtr;
    return 0;
}
```