**PHP Full Stack Assignment**

**Name :- Makvana Chandrakant Maganbhai**

**Module 3**

* **Introduction to C++**

**Q1. What are the key differences between Procedural Programming and Object-Oriented Programming (OOP)?**

**1. What is Procedural Programming?**

**Procedural Programming** is a programming paradigm based on the concept of **procedures or functions**. It focuses on **step-by-step instructions** to perform a task.

* C is a classic example of a procedural language.

**Characteristics:**

* Code is organized into **functions**.
* Data and functions are **separate**.
* Emphasis on **sequence**, **selection**, and **iteration**.
* Execution flows from top to bottom.

**2. What is Object-Oriented Programming (OOP)?**

**Object-Oriented Programming (OOP)** organizes code around **objects**, which are instances of **classes**. It combines **data** and **functions** that operate on that data into a single unit.

* C++, Java, and Python are examples of OOP languages.

**Characteristics:**

* Code is organized into **classes and objects**.
* Emphasis on **data encapsulation**, **inheritance**, and **polymorphism**.
* Encourages **modularity**, **reuse**, and **scalability**.

**Key Differences Between Procedural and Object-Oriented Programming**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| **Approach** | Function-based | Object-based |
| **Data Handling** | Data is **exposed** and passed to functions | Data is **encapsulated** inside objects |
| **Modularity** | Code is modularized using **functions** | Code is modularized using **classes/objects** |
| **Reusability** | Limited (code reuse via functions only) | High (through **inheritance and polymorphism**) |
| **Security** | Less secure (global data can be modified freely) | More secure (private/protected access) |
| **Ease of Maintenance** | Can be harder for large programs | Easier due to modular structure |
| **Examples of Languages** | C, Pascal, FORTRAN | C++, Java, Python, C# |
| **Real-world Mapping** | No real-world modeling | Models real-world entities as objects |

**Q2. List and explain the main advantages of OOP over POP.**

## Advantages of OOP Over POP

|  |  |  |
| --- | --- | --- |
| **OOP Feature** | **Description** | **Advantage Over POP** |
| **1. Encapsulation** | Bundles data and methods into a single unit (class), and restricts direct access to data. | Improves **data security** and code clarity. |
| **2. Inheritance** | Allows one class to inherit properties and methods from another. | Promotes **code reuse** and reduces redundancy. |
| **3. Polymorphism** | Allows objects to behave differently based on context (e.g., method overloading or overriding). | Supports **flexible** and **scalable** design. |
| **4. Abstraction** | Hides unnecessary details and exposes only the relevant features. | Helps manage **complexity** more easily. |
| **5. Modularity** | Code is divided into self-contained classes and objects. | Makes code **easier to manage, test, and debug**. |
| **6. Maintainability** | Encapsulated, modular design allows developers to update code with minimal impact on others. | Enhances long-term **project maintainability**. |
| **7. Reusability** | Classes can be reused across programs or projects. | Saves time and effort in **development**. |
| **8. Real-world modeling** | Objects represent real-world entities (e.g., Car, Student), improving conceptual design. | More intuitive and **easier to understand**. |
| **9. Collaboration** | Encourages teamwork through well-defined interfaces and class structures. | Facilitates **team development**. |

**Q3. Explain the steps involved in setting up a C++ development environment.**

**Step 1: Install a C++ Compiler**

**Recommended Compiler: GCC (GNU Compiler Collection)**

Other options include Microsoft’s MSVC (Windows) and Clang (macOS/Linux).

**For Windows:**

**Option A: Install Code::Blocks (comes with GCC)**

1. Go to <https://www.codeblocks.org/downloads>
2. Download the version with “**mingw-setup**”
3. Install and launch Code::Blocks
4. Start writing C++ programs immediately

**Option B: Install MinGW manually**

1. Download MinGW from <https://osdn.net/projects/mingw/>
2. During installation, select:
   * mingw32-gcc-g++ (C++ compiler)
3. Add C:\MinGW\bin to your **system PATH**
4. Open Command Prompt:
5. g++ --version

If version info appears, you're good to go!

**For macOS:**

1. Install **Xcode Command Line Tools**:
2. xcode-select --install
3. Check installation:
4. g++ --version

**For Linux (Ubuntu/Debian):**

sudo apt update

sudo apt install g++

g++ --version

**Step 2: Choose and Set Up an IDE or Text Editor**

**Popular IDEs for C++:**

|  |  |  |
| --- | --- | --- |
| **IDE** | **Features** | **Platforms** |
| **Code::Blocks** | Lightweight, built-in debugger | Windows, Linux |
| **Dev-C++** | Simple, beginner-friendly | Windows |
| **Visual Studio Code (VS Code)** | Modern, extensible via extensions | Windows, macOS, Linux |
| **CLion** | Smart IDE by JetBrains | Cross-platform |

**Setting Up VS Code for C++**

1. Install [VS Code](https://code.visualstudio.com/)
2. Install the **C/C++ extension** by Microsoft
3. Install compiler (GCC or MinGW as explained above)
4. Configure a tasks.json and launch.json for build/run (VS Code will guide you)

**Step 3: Write Your First Program**

Create a file named hello.cpp:

#include <iostream>

using namespace std;

int main() {

cout << "Hello, World!" << endl;

return 0;

}

**Step 4: Compile and Run**

**In Terminal or Command Prompt:**

g++ hello.cpp -o hello

./hello

On Windows:

g++ hello.cpp -o hello.exe

hello

**Tools**

|  |  |
| --- | --- |
| **Tool** | **Purpose** |
| **Compiler** | Translates C++ code to machine code |
| **IDE/Editor** | Provides code editing, debugging, UI |
| **Debugger** | Helps troubleshoot errors |

**Q4. What are the main input/output operations in C++? Provide examples.**

**Main Input Operation: cin**

**Main Output Operation: cout**

These are part of the **iostream** library, and they use **stream operators**:

|  |  |  |
| --- | --- | --- |
| **Operation** | **Operator** | **Description** |
| Output | << | Insertion operator |
| Input | >> | Extraction operator |

**Include the I/O Stream Library**

#include <iostream>

using namespace std;

**Example: Basic Input and Output**

#include <iostream>

using namespace std;

int main() {

int age;

string name;

cout << "Enter your name: ";

cin >> name;

cout << "Enter your age: ";

cin >> age;

cout << "Hello, " << name << "! You are " << age << " years old." << endl;

return 0;

}

**Notes:**

* cin reads input from the **keyboard**.
* cout sends output to the **console**.
* endl is used to insert a newline (same as \n but flushes the output buffer).
* For full-line input (with spaces), use getline():

getline(cin, name); // Reads a full line, including spaces

**Common I/O Operations in C++**

|  |  |  |
| --- | --- | --- |
| **Task** | **Function or Operator** | **Example** |
| Print to console | cout + << | cout << "Hello"; |
| Read from console | cin + >> | cin >> number; |
| Read full line | getline(cin, variable) | getline(cin, name); |
| Format output | iomanip library | setprecision(2) |
| File input/output | ifstream, ofstream | ofstream fout("data.txt"); |

**Example: Reading Full Line with Spaces**

#include <iostream>

#include <string>

using namespace std;

int main() {

string fullName;

cout << "Enter your full name: ";

getline(cin, fullName);

cout << "Hello, " << fullName << "!" << endl;

return 0;

}

* **2. Variables, Data Types, and Operators**

**Q5. What are the different data types available in C++? Explain with examples.**

**1. Basic (or Fundamental) Data Types**

These are built into the language and include:

|  |  |  |
| --- | --- | --- |
| **Data Type** | **Description** | **Example** |
| int | Integer numbers (no decimals) | int age = 25; |
| float | Floating-point numbers (single precision) | float temperature = 36.6; |
| double | Double-precision floating-point numbers | double pi = 3.14159; |
| char | Single character | char grade = 'A'; |
| bool | Boolean type (true or false) | bool isPassed = true; |

**2. Derived Data Types**

These are built from fundamental types.

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| Array | Collection of elements of the same type | int numbers[5] = {1, 2, 3, 4, 5}; |
| Pointer | Stores memory address of another variable | int\* ptr = &age; |
| Reference | Alias for another variable | int& ref = age; |
| Function | Returns a type and can take arguments | int add(int a, int b) { return a + b; } |

**3. User-Defined Data Types**

You can create your own data types using:

|  |  |  |
| --- | --- | --- |
| **Type** | **Description** | **Example** |
| struct | Groups variables under one name | struct Person { string name; int age; }; |
| class | Similar to struct but supports OOP concepts | class Car { public: string model; }; |
| union | Shares memory among members | union Data { int i; float f; }; |
| enum | Enumerated constants | enum Color { Red, Green, Blue }; |
| typedef or using | Creates a new name for an existing type | typedef int Marks; or using Marks = int; |

**4. Void Type**

* Used for functions that do not return a value.
* void greet() {
* cout << "Hello!" << endl;
* }

**5. Wide Character Type**

* Used to store larger character sets like Unicode.
* wchar\_t w = L'Ω'; // L denotes a wide character

## Q6.Explain the difference between implicit and explicit type conversion in C++.

**1. Implicit Type Conversion (Type Promotion)**

Also called **automatic type conversion**, this occurs **automatically by the compiler** when:

* Operands of different types are used in an expression.
* A value is assigned to a variable of a different type (if safe).

**Key Points:**

* Happens **without** programmer intervention.
* Usually converts to a **larger** or more **precise** data type (e.g., int to float).

**Example:**

int i = 10;

float f = i; // int is implicitly converted to float

float result = i + 5.5; // i is promoted to float before addition

**2. Explicit Type Conversion (Type Casting)**

Also known as **type casting**, this occurs when the **programmer manually converts** one data type to another using casting operators.

**Key Points:**

* Requires **manual** specification.
* Can lead to **data loss** (e.g., float to int).
* Syntax:
  + **C-style cast:** (new\_type)expression
  + **C++ style cast:** static\_cast<new\_type>(expression)

**Example:**

float f = 3.14;

int i = (int)f; // C-style cast (i = 3)

int j = static\_cast<int>(f); // C++ style cast (j = 3)

**Comparison Table**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Implicit Conversion** | **Explicit Conversion** |
| Who performs it? | Compiler | Programmer |
| Syntax | Automatic | Manual: (type) or static\_cast<type> |
| Risk of data loss | Low (usually safe promotions) | High (may truncate or lose precision) |
| Control | Less | More |

**Example Showing Both:**

#include <iostream>

using namespace std;

int main() {

int a = 5;

float b = 2.5;

float sum = a + b; // implicit: a is converted to float

int intSum = (int)(a + b); // explicit: result is cast to int

cout << "Implicit sum: " << sum << endl; // 7.5

cout << "Explicit sum: " << intSum << endl; // 7

return 0;

}

**Q7.What are the different types of operators in C++? Provide examples of each.**

.**1. Arithmetic Operators**

Used to perform mathematical operations.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | a + b |
| - | Subtraction | a - b |
| \* | Multiplication | a \* b |
| / | Division | a / b |
| % | Modulus (remainder) | a % b |

### Example:

int a = 10, b = 3;

cout << a + b; // 13

## ****2. Relational (Comparison) Operators****

Used to compare two values.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal to | a == b |
| != | Not equal to | a != b |
| > | Greater than | a > b |
| < | Less than | a < b |
| >= | Greater or equal | a >= b |
| <= | Less or equal | a <= b |

## ****3. Logical Operators****

Used to combine multiple conditions.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | (a > 0 && b > 0) |
| ` |  | ` |
| ! | Logical NOT | !(a > 0) |

## ****4. Assignment Operators****

Used to assign values to variables.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Assign | a = 5 |
| += | Add and assign | a += 3 (a = a + 3) |
| -= | Subtract and assign | a -= 2 |
| \*= | Multiply and assign | a \*= 2 |
| /= | Divide and assign | a /= 4 |
| %= | Modulus and assign | a %= 3 |

## ****5. Increment and Decrement Operators****

Used to increase or decrease a value by one.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ++ | Increment | ++a or a++ |
| -- | Decrement | --a or a-- |

## ****6. Bitwise Operators****

Operate at the binary level.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | AND | a & b |
| ` | ` | OR |
| ^ | XOR | a ^ b |
| ~ | NOT (1’s complement) | ~a |
| << | Left shift | a << 1 |
| >> | Right shift | a >> 1 |

## ****7. Conditional (Ternary) Operator****

A shortcut for if-else.

| **Syntax** | **Description** |
| --- | --- |
| condition ? x : y; | If condition is true, returns x; otherwise y |

### Example:

int a = 10, b = 20;

int max = (a > b) ? a : b;

## ****8. Special Operators****

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| sizeof | Returns size of data type/variable | sizeof(int) |
| typeid | Returns type information | typeid(a).name() |
| & | Address of | &a |
| \* | Pointer dereference | \*ptr |
| -> | Access members of a structure/class through pointer | ptr->member |
| . | Access member of a structure/class | obj.member |

## ****9. Scope Resolution Operator**** ::

Used to access global variables or class members when there is ambiguity.

### Example:

int x = 10;

void show() {

int x = 20;

cout << ::x; // refers to global x

}

**Q8. Explain the purpose and use of constants and literals in C++.**

## ****1. Constants in C++****

### ****Purpose:****

Constants are used to define values that **should not be changed** after their initial assignment. They improve:

* Code **readability**
* **Maintainability**
* **Safety** (prevents accidental changes)

### ****Types of Constants:****

#### a. ****Literal Constants****

* Directly used values (like 5, 3.14, 'A')

#### b. const ****Keyword****

* Used to declare a constant variable.

const int MAX = 100;

#### c. #define ****Preprocessor Directive****

* Creates a symbolic name for a constant.

#define PI 3.14159

#### d. constexpr ****(C++11 and above)****

* Compile-time constant, more powerful than const.

constexpr int size = 10;

## ****2. Literals in C++****

### ****Purpose:****

Literals are the **actual fixed values** you use directly in code. C++ supports several types:

### ****Types of Literals:****

|  |  |  |
| --- | --- | --- |
| **Type** | **Example** | **Description** |
| Integer literal | 100, 0xFF, 075 | Decimal, hexadecimal, octal |
| Floating-point | 3.14, 2.5e-3 | Standard and exponential formats |
| Character literal | 'A', '9' | Enclosed in single quotes |
| String literal | "Hello" | Enclosed in double quotes |
| Boolean literal | true, false | Boolean values |
| Null pointer | nullptr | Special constant for null pointer |

## ****Example Combining Constants and Literals****

#include <iostream>

using namespace std;

const double PI = 3.14159; // constant

constexpr int RADIUS = 5; // compile-time constant

int main() {

double area = PI \* RADIUS \* RADIUS; // using literals and constants

cout << "Area of circle: " << area << endl;

return 0;

}

## ****Why Use Constants and Literals?****

|  |  |
| --- | --- |
| **Benefit** | **Description** |
| Avoid Magic Numbers | Replace hard-coded values with named constants |
| Easy Maintenance | Change value in one place, not throughout code |
| Prevent Modification | Protect critical values from accidental changes |
| Readability & Clarity | Named constants make code more understandable |

* **3. Control Flow Statements**

**Q9. What are the different data types available in C++? Explain with examples.**

C++ offers a variety of data types that specify the kind and size of data a variable can hold. These data types are broadly categorized into fundamental (primitive), derived, and user-defined types. Below is an explanation of the main data types available in C++ with examples:

## Fundamental Data Types

1. **Integer (int)**
   * Stores whole numbers without decimals.
   * Size: Typically 2 or 4 bytes.
   * Example:

cpp

**int** myNum = 5; *// integer variable holding the value 5*

* + Can be signed or unsigned.
  + Range: Usually from -2,147,483,648 to 2,147,483,647 for 4 bytes.

1. **Character (char)**
   * Stores a single character or ASCII value.
   * Size: 1 byte.
   * Example:

cpp

**char** myLetter = 'D'; *// character variable holding 'D'*

1. **Boolean (bool)**
   * Stores true or false.
   * Size: 1 byte.
   * Example:

cpp

**bool** isTrue = true; *// boolean variable holding true*

1. **Floating Point (float)**
   * Stores fractional numbers (decimals).
   * Size: 4 bytes.
   * Precision: About 6-7 decimal digits.
   * Example:

cpp

**float** myFloatNum = 5.99f; *// float variable holding 5.99*

1. **Double Floating Point (double)**
   * Stores fractional numbers with double precision.
   * Size: 8 bytes.
   * Precision: About 15 decimal digits.
   * Example:

cpp

**double** myDoubleNum = 9.98; *// double variable holding 9.98*

1. **Void (void)**
   * Represents absence of type.
   * Used for functions that do not return a value or for generic pointers.
   * Size: 0 bytes.
   * Example:

cpp

**void** myFunction() { */\* no return value \*/* }

**void**\* genericPointer; *// pointer to any type*

1. **Wide Character (wchar\_t)**
   * Used to store wide characters (Unicode).
   * Size: 2 or 4 bytes depending on the system.
   * Example:

cpp

**wchar\_t** myWideChar = L'A'; *// wide character 'A'*

## Derived Data Types

* **Array**
  + Collection of elements of the same data type stored contiguously.
  + Example:

cpp

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

* **Function**
  + Functions have return types and parameters, acting as user-defined data types.
  + Example:

cpp

**int** add(**int** a, **int** b) {

**return** a + b;

}

* **Pointer, Reference** (not detailed here but are important derived types).

## User-Defined Data Types

* Created by programmers using struct, class, enum, etc.
* Allow complex data structures tailored to specific needs.

## Summary Table of Fundamental Data Types

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Size (bytes)** | **Description** | **Example** |
| int | 2 or 4 | Whole numbers | int age = 18; |
| char | 1 | Single character | char letter = 'A'; |
| bool | 1 | Boolean true/false | bool flag = true; |
| float | 4 | Floating-point decimal numbers | float pi = 3.14f; |
| double | 8 | Double precision floating-point | double e = 2.71828; |
| void | 0 | No type (used for functions) | void func(); |
| wchar\_t | 2 or 4 | Wide character (Unicode) | wchar\_t w = L'Ω'; |

## Example Code Demonstrating Various Data Types

cpp

#**include** <iostream>

**using** **namespace** std;

**int** main() {

**int** myInt = 42;

**char** myChar = 'A';

**bool** myBool = true;

**float** myFloat = 3.14159f;

**double** myDouble = 2.718281828;

**wchar\_t** myWideChar = L'Ω';

cout << "Integer: " << myInt << endl;

cout << "Character: " << myChar << endl;

cout << "Boolean: " << boolalpha << myBool << endl;

cout << "Float: " << myFloat << endl;

cout << "Double: " << myDouble << endl;

wcout << L"Wide Character: " << myWideChar << endl;

**return** 0;

}

Q10. Explain the difference between implicit and explicit type conversion in C++.

## Implicit Type Conversion (Automatic Conversion)

* **Definition:** The compiler automatically converts one data type to another without any explicit instruction from the programmer.
* **When it happens:** Usually when different data types are mixed in expressions, assignments, or function calls.
* **Purpose:** To make operations between compatible types seamless and avoid data loss or errors.
* **Example:**

cpp

**int** i = 9;

**float** f = i; *// int to float conversion done automatically by compiler*

* **Characteristics:**
  + No special syntax needed.
  + Happens silently during compilation.
  + Generally safe but can cause subtle bugs like precision loss (e.g., float to int truncation).
  + Example risk: assigning a float value 9.99 to an int results in 9 losing the decimal part.
* **Common uses:** Numeric promotions (e.g., int to double), pointer upcasting (derived class pointer to base class pointer)[3](https://www.scaler.com/topics/cpp/type-conversion-in-cpp/)[4](https://www.w3schools.com/c/c_type_conversion.php)[5](https://intellipaat.com/blog/type-conversion-in-cpp/)[6](https://www.softwaretestinghelp.com/type-conversions-in-cpp/).

## Explicit Type Conversion (Type Casting)

* **Definition:** Conversion manually specified by the programmer to convert one data type to another.
* **How it is done:** Using cast operators such as C-style cast (type)variable, function-style cast type(variable), or C++ cast operators like static\_cast<>.
* **Purpose:** To override implicit conversion rules, handle incompatible types, or enforce specific conversions.
* **Example:**

cpp

**double** d = 9.99;

**int** i = (**int**)d; *// explicit cast from double to int, truncates decimal part*

* **Characteristics:**
  + Requires explicit syntax.
  + Gives programmer full control over conversion.
  + Can introduce risks like data loss or undefined behavior if used improperly.
  + Used to prevent unintended implicit conversions by marking constructors as explicit.
* **Example with explicit constructor:**

cpp

**class** Money {

**public**:

**explicit** Money(**double** amount) : amount(amount) {}

**double** amount;

};

Money m1 = 79.99; *// Error: implicit conversion disallowed*

Money m2(79.99); *// OK: explicit conversion via constructor*

Money m3 = (Money)9.99; *// OK: explicit cast*

* **Common uses:** Downcasting, overriding implicit conversions, converting incompatible types safely[1](https://learn.microsoft.com/en-us/cpp/cpp/user-defined-type-conversions-cpp?view=msvc-170)[2](https://stackoverflow.com/questions/7099957/implicit-vs-explicit-conversion)[3](https://www.scaler.com/topics/cpp/type-conversion-in-cpp/)[5](https://intellipaat.com/blog/type-conversion-in-cpp/)[6](https://www.softwaretestinghelp.com/type-conversions-in-cpp/).

## Summary Table

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Implicit Conversion** | **Explicit Conversion** |
| Initiated by | Compiler (automatic) | Programmer (manual) |
| Syntax | None (automatic) | Cast operators (type), static\_cast<> |
| Control | Limited by compiler rules | Full control by programmer |
| Safety | Generally safe but can lose data | Riskier, can cause data loss or errors |
| Use cases | Numeric promotions, compatible types | Incompatible types, forced conversions |
| Example | float f = 9; | int i = (int)9.99; |

**Q11. What are the different types of operators in C++? Provide examples of each.**

C++ provides a rich set of operators categorized by the types of operations they perform on variables and values. Here are the main types of operators in C++ with examples for each:

## 1. Arithmetic Operators

Used to perform basic mathematical operations.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | x + y |
| - | Subtraction | x - y |
| \* | Multiplication | x \* y |
| / | Division | x / y |
| % | Modulus (remainder) | x % y |
| ++ | Increment | ++x or x++ |
| -- | Decrement | --x or x-- |

**Example:**

cpp

**int** a = 10, b = 3;

**int** sum = a + b; *// 13*

**int** mod = a % b; *// 1*

a++; *// a becomes 11*

## 2. Assignment Operators

Used to assign values to variables, including compound assignments.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment | x = 5 |
| += | Add and assign | x += 3 |
| -= | Subtract and assign | x -= 2 |
| \*= | Multiply and assign | x \*= 4 |
| /= | Divide and assign | x /= 2 |
| %= | Modulus and assign | x %= 3 |

**Example:**

cpp

**int** x = 10;

x += 5; *// x is now 15*

## 3. Relational (Comparison) Operators

Used to compare two values, returning true or false.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal to | x == y |
| != | Not equal to | x != y |
| < | Less than | x < y |
| > | Greater than | x > y |
| <= | Less than or equal to | x <= y |
| >= | Greater than or equal to | x >= y |

**Example:**

cpp

**if** (a != b) {

*// Executes if a is not equal to b*

}

## 4. Logical Operators

Used to combine or invert boolean expressions.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | (x > 0 && y > 0) |
| ` |  | ` |
| ! | Logical NOT | !(x == y) |

**Example:**

cpp

**bool** result = (a > 0) && (b < 5);

## 5. Bitwise Operators

Operate on individual bits of integer types.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Bitwise AND | x & y |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR | x ^ y |
| ~ | Bitwise NOT | ~x |
| << | Left shift | x << 2 |
| >> | Right shift | x >> 2 |

**Example:**

cpp

**int** x = 5; *// binary 0101*

**int** y = x << 1; *// y is 10 (binary 1010)*

## 6. Ternary (Conditional) Operator

A shorthand for an if-else statement with three operands.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ?: | Conditional expression | condition ? expr1 : expr2 |

**Example:**

cpp

**int** max = (a > b) ? a : b; *// max is assigned the greater of a and b*

## 7. Other Operators (Miscellaneous)

* **Sizeof Operator:** Returns the size of a data type or variable.

cpp

**sizeof**(**int**); *// typically 4 bytes*

* **Comma Operator:** Allows multiple expressions in one statement, evaluates left to right.

cpp

**int** a = (1, 2, 3); *// a is assigned 3*

* **Address-of Operator (&):** Returns the memory address of a variable.
* **Pointer Dereference Operator (\*):** Accesses the value at a pointer.
* **Member Access Operators:** . (direct), -> (through pointer).

## Summary Table of Main Operator Types

|  |  |  |
| --- | --- | --- |
| **Operator Type** | **Purpose** | **Example** |
| Arithmetic | Math operations | +, -, \*, / |
| Assignment | Assign values | =, +=, -= |
| Relational | Compare values | ==, !=, < |
| Logical | Logical operations on booleans | &&, ` |
| Bitwise | Bit-level operations | &, ` |
| Ternary (Conditional) | Conditional expressions | ?: |
| Miscellaneous | Size, address, member access | sizeof, &, . |

**Q12. Explain the purpose and use of constants and literals in C++.**

Constants and literals in C++ both represent fixed values, but they differ in their definitions and usage:

## Constants in C++

* **Definition:** Constants are variables whose values cannot be changed once initialized. They are defined using the const keyword.
* **Purpose:** To provide meaningful names to fixed values, improving code readability and preventing accidental modification.
* **Syntax:**

cpp

**const** data\_type constant\_name = value;

* **Example:**

cpp

**const** **int** DAYS\_IN\_WEEK = 7;

* **Behavior:** Attempting to modify a constant after its definition results in a compile-time error.
* **Use case:** When you want to use a fixed value repeatedly with a descriptive name, such as PI, MAX\_USERS, or TIMEOUT.

## Literals in C++

* **Definition:** Literals are the actual fixed values written directly in the source code. They represent raw data of various types.
* **Types of literals:**
  + **Integer Literals:** e.g., 10, 0, -25, 0x4B (hexadecimal), 0213 (octal)
  + **Floating-Point Literals:** e.g., 3.14, -0.001, 2e10
  + **Character Literals:** e.g., 'a', '1', '@'
  + **String Literals:** e.g., "Hello, World!", "C++"
  + **Boolean Literals:** true, false
* **Example:**

cpp

**int** x = 5; *// 5 is an integer literal*

**char** c = 'A'; *// 'A' is a character literal*

**bool** flag = true; *// true is a boolean literal*

* **Characteristics:** Literals are fixed values embedded directly in code and cannot be changed.

## Relationship and Differences

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Constants** | **Literals** |
| What they represent | Named fixed values (variables) | Fixed values written directly |
| Mutability | Cannot be changed after initialization | Always fixed and unchangeable |
| Defined by | const keyword | Directly written in code |
| Usage | To give meaningful names to values | To represent raw data in code |
| Example | const int DAYS = 7; | 7 (integer literal) |

## Summary

* **Constants** are variables declared with const whose values cannot be altered after initialization.
* **Literals** are the actual fixed values used in the code.
* Constants often use literals to initialize their values.
* Using constants improves code clarity and safety by preventing modification of important fixed values.

## Example demonstrating constants and literals:

cpp

#**include** <iostream>

**using** **namespace** std;

**int** main() {

**const** **double** PI = 3.14159; *// PI is a constant initialized with a floating-point literal*

**int** radius = 5; *// radius is a variable*

**double** area = PI \* radius \* radius; *// using constant and literals in expression*

cout << "Area of circle: " << area << endl;

**return** 0;

}

* **Functions and Scope**

Q13. What is a function in C++? Explain the concept of function declaration, definition, and calling.

A **function in C++** is a group of statements that perform a specific task. Functions help in modularizing a program by dividing code into reusable blocks. Every C++ program has at least one function, main(), which is the entry point of the program[2](https://www.tutorialspoint.com/cplusplus/cpp_functions.htm)[5](https://simplesnippets.tech/functions-in-cpp-declaring-defining-and-calling/).

## Concept of Function Declaration, Definition, and Calling

**1. Function Declaration:**  
This tells the compiler about the function's name, return type, and parameters (if any). It acts as a prototype so that the compiler knows what to expect when the function is called. The declaration does not contain the body of the function.  
Example:

cpp

**void** myFunction();

This informs the compiler that there is a function named myFunction that returns void and takes no parameters[1](https://www.w3schools.com/cpp/cpp_functions.asp)[2](https://www.tutorialspoint.com/cplusplus/cpp_functions.htm)[5](https://simplesnippets.tech/functions-in-cpp-declaring-defining-and-calling/)[7](https://learncplusplus.org/learn-about-function-declaration-and-definition-in-c/).

**2. Function Definition:**  
This provides the actual body of the function — the code that runs when the function is called. The definition includes the function header (which repeats the return type, name, and parameters) and the function body enclosed in braces {}.  
Example:

cpp

**void** myFunction() {

std::cout << "I just got executed!";

}

This is where the task of the function is implemented[1](https://www.w3schools.com/cpp/cpp_functions.asp)[2](https://www.tutorialspoint.com/cplusplus/cpp_functions.htm)[5](https://simplesnippets.tech/functions-in-cpp-declaring-defining-and-calling/)[7](https://learncplusplus.org/learn-about-function-declaration-and-definition-in-c/).

**3. Function Calling:**  
To execute the function, you call it by its name followed by parentheses and a semicolon. If the function takes parameters, you pass the required arguments inside the parentheses. When a function is called, program control transfers to the function, executes its body, and then returns control back to the caller.  
Example:

cpp

myFunction();

This will invoke the function and execute its code[1](https://www.w3schools.com/cpp/cpp_functions.asp)[2](https://www.tutorialspoint.com/cplusplus/cpp_functions.htm)[5](https://simplesnippets.tech/functions-in-cpp-declaring-defining-and-calling/)[7](https://learncplusplus.org/learn-about-function-declaration-and-definition-in-c/).

## Summary Example

cpp

#**include** <iostream>

*// Function declaration*

**void** myFunction();

**int** main() {

*// Function call*

myFunction();

**return** 0;

}

*// Function definition*

**void** myFunction() {

std::cout << "I just got executed!";

}

In this example, the function myFunction is declared before main(), defined after main(), and called inside main(). This separation helps organize code and avoid errors related to calling functions before the compiler knows about them[1](https://www.w3schools.com/cpp/cpp_functions.asp)[7](https://learncplusplus.org/learn-about-function-declaration-and-definition-in-c/).

**In brief:**

* **Declaration**: Tells the compiler about the function's signature.
* **Definition**: Contains the actual code to be executed.
* **Calling**: Executes the function by transferring control to it.

**Q14. What is the scope of variables in C++? Differentiate between local and global scope.**

## Scope of Variables in C++

The **scope** of a variable in C++ refers to the region or portion of the program where that variable is visible and accessible. It determines where you can use the variable in your code and also affects the variable's lifetime during program execution[1](https://codefinity.com/courses/v2/b7943689-5d01-4a22-a024-0d29968dd8ac/2c4bad66-393e-48cd-8e28-bf69f97feb30/938339d5-8406-486c-84db-a4abbcc28de5)[2](https://www.w3schools.com/cpp/cpp_scope.asp)[3](https://learn.microsoft.com/en-us/cpp/cpp/scope-visual-cpp?view=msvc-170).

## Local Scope vs Global Scope

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Local Scope** | **Global Scope** |
| **Definition** | Variables declared inside a function or block ({}) | Variables declared outside all functions, usually at the top of the file |
| **Visibility** | Accessible only within the function or block where declared | Accessible throughout the entire program after declaration |
| **Lifetime** | Created when the block/function is entered, destroyed when exited | Exists for the entire duration of the program execution |
| **Usage** | Used for temporary data needed only within a specific function or block | Used for data that needs to be shared across multiple functions |
| **Example** | int x = 5; inside a function | int g = 10; declared outside any function |
| **Access outside scope** | Not accessible; attempting to do so causes a compile error | Accessible from any function or block in the program |

## Explanation with Examples

**Local Variable Example:**

cpp

**void** myFunction() {

**int** localVar = 10; *// local scope*

std::cout << localVar; *// valid here*

}

**int** main() {

myFunction();

std::cout << localVar; *// Error: 'localVar' not declared in this scope*

}

Here, localVar is only accessible inside myFunction() and cannot be used in main() or elsewhere[1](https://codefinity.com/courses/v2/b7943689-5d01-4a22-a024-0d29968dd8ac/2c4bad66-393e-48cd-8e28-bf69f97feb30/938339d5-8406-486c-84db-a4abbcc28de5)[2](https://www.w3schools.com/cpp/cpp_scope.asp)[7](https://www.codespeedy.com/scope-of-variables-in-cpp/).

**Global Variable Example:**

cpp

#**include** <iostream>

**int** globalVar = 20; *// global scope*

**void** myFunction() {

std::cout << globalVar; *// accessible here*

}

**int** main() {

std::cout << globalVar; *// accessible here too*

myFunction();

**return** 0;

}

globalVar is declared outside any function and can be accessed anywhere in the program[4](https://www.tutorialspoint.com/cplusplus/cpp_variable_scope.htm)[5](https://open-catalog.codee.com/Glossary/Variable-scope/)[6](https://www.codecademy.com/learn/learn-intermediate-c-plus-plus/modules/variable-scope-and-storage-classes/cheatsheet).

**Q15. Explain recursion in C++ with an example.**

Recursion in C++ is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, simpler sub-problems until a base condition is met that stops the recursive calls[1](https://www.w3schools.com/cpp/cpp_functions_recursion.asp)[3](https://www.programiz.com/cpp-programming/recursion)[5](https://www.tutorialspoint.com/cplusplus/cpp_recursion.htm).

## Key Concepts of Recursion

* **Recursive Case:** The part where the function calls itself with modified arguments to approach the base case.
* **Base Case:** The condition that stops further recursive calls and allows the function to return a value.

## Example: Sum of First N Natural Numbers Using Recursion

cpp

#**include** <iostream>

**using** **namespace** std;

**int** sum(**int** k) {

**if** (k > 0) {

**return** k + sum(k - 1); *// Recursive call*

} **else** {

**return** 0; *// Base case*

}

}

**int** main() {

**int** result = sum(10);

cout << "Sum of first 10 natural numbers is " << result;

**return** 0;

}

## Explanation:

* The function sum calls itself with k-1 until k becomes 0.
* When k is 0, it returns 0 (base case), stopping recursion.
* Each recursive call adds the current k to the sum returned by the next call.
* The final result is the sum of numbers from 10 down to 1.

## How It Works:

The calls unfold like this:

text

sum(10) = 10 + sum(9)

sum(9) = 9 + sum(8)

...

sum(1) = 1 + sum(0)

sum(0) = 0 (base case)

Then the results return back up the call stack adding all values together[1](https://www.w3schools.com/cpp/cpp_functions_recursion.asp)[3](https://www.programiz.com/cpp-programming/recursion)[5](https://www.tutorialspoint.com/cplusplus/cpp_recursion.htm).

## Advantages of Recursion

* Simplifies code for problems naturally defined by repetition or self-similarity (e.g., factorial, Fibonacci, tree traversals).
* Provides elegant and clear solutions for complex problems.

## Disadvantages of Recursion

* Uses more stack memory and processor time compared to iterative solutions.
* Can be harder to debug.
* Risk of infinite recursion if base case is not properly defined.

**Q16. What are function prototypes in C++? Why are they used?**

A **function prototype in C++** is a declaration of a function that specifies its name, return type, and the types of its parameters, but does not include the function body (implementation)[2](https://en.wikipedia.org/wiki/Function_prototype)[5](https://cppscripts.com/cpp-function-prototype). It informs the compiler about the function's interface before its actual definition appears in the code.

## Why Are Function Prototypes Used?

* **Enable Early Recognition:** They allow the compiler to recognize and validate function calls that appear before the function is defined in the source code[1](https://www.reddit.com/r/learnprogramming/comments/1gmvkqq/what_is_a_function_prototype_in_c_and_what_is_it/)[3](https://www.tutorialspoint.com/what-is-the-purpose-of-a-function-prototype-in-c-cplusplus)[5](https://cppscripts.com/cpp-function-prototype).
* **Type Checking:** Prototypes provide information about the number and types of parameters and the return type, enabling the compiler to perform type checking during function calls, which helps catch mismatches and errors at compile time[3](https://www.tutorialspoint.com/what-is-the-purpose-of-a-function-prototype-in-c-cplusplus)[6](https://www.careerride.com/C++-what-are-function-prototypes.aspx).
* **Code Organization:** They allow function definitions to be placed after the main() function or in separate files, improving code readability and modularity[1](https://www.reddit.com/r/learnprogramming/comments/1gmvkqq/what_is_a_function_prototype_in_c_and_what_is_it/)[5](https://cppscripts.com/cpp-function-prototype).
* **Separate Compilation:** In larger projects, prototypes are typically placed in header files (.h), allowing multiple source files to share function declarations and enabling separate compilation and linking[1](https://www.reddit.com/r/learnprogramming/comments/1gmvkqq/what_is_a_function_prototype_in_c_and_what_is_it/)[2](https://en.wikipedia.org/wiki/Function_prototype).

## Example of a Function Prototype

cpp

**int** add(**int** a, **int** b); *// Function prototype*

This tells the compiler that a function named add exists, which takes two integers as parameters and returns an integer. The actual function definition can appear later in the code.

## Summary

|  |  |
| --- | --- |
| **Aspect** | **Description** |
| What it is | Declaration specifying function name, return type, and parameter types without the body |
| Purpose | Inform compiler about function before use, enable type checking, and support modular code organization |
| Where used | Typically above main() or in header files |
| Benefit | Prevents errors, allows flexible code structure |

* **5. Arrays and Strings**

**Q17. What are arrays in C++? Explain the difference between single-dimensional and multi- dimensional arrays.**

## Arrays in C++

An **array** in C++ is a collection of elements of the same data type stored in contiguous memory locations. It allows storing multiple values under a single variable name, accessible via indices starting from 0. Arrays are useful for managing large sets of data efficiently without declaring multiple separate variables[1](https://www.w3schools.com/cpp/cpp_arrays.asp)[2](https://cplusplus.com/doc/tutorial/arrays/)[3](https://unstop.com/blog/array-in-cpp)[5](https://www.luisllamas.es/en/cpp-what-are-arrays/)[7](https://www.softwaretestinghelp.com/arrays-in-cpp/).

## Declaration and Initialization

You declare an array by specifying the data type, the array name, and the number of elements inside square brackets:

cpp

**int** myArray[5]; *// Declares an integer array with 5 elements*

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

cpp

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

If you provide fewer initial values than the size, remaining elements are initialized to zero by default:

cpp

**int** numbers[5] = {1, 2}; *// Initializes as {1, 2, 0, 0, 0}*

## Difference Between Single-Dimensional and Multi-Dimensional Arrays

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Single-Dimensional Array** | **Multi-Dimensional Array** |
| **Structure** | Linear collection of elements | Array of arrays (e.g., 2D, 3D arrays) |
| **Indexing** | Accessed using a single index: array[index] | Accessed using multiple indices: array[row][column] |
| **Example Declaration** | int arr[5]; | int matrix[3][3]; |
| **Memory Layout** | Contiguous block of memory storing elements sequentially | Contiguous memory storing rows one after another |
| **Use Case** | Storing list or sequence of elements | Representing tables, matrices, grids, or higher-dimensional data |
| **Access Example** | arr[2] accesses the 3rd element | matrix[1][2] accesses element at 2nd row, 3rd column |

## Explanation

* **Single-Dimensional Array:**  
  Stores elements in a single line. For example, an array of integers representing scores:

cpp

**int** scores[5] = {90, 85, 88, 92, 75};

Access elements by a single index starting at 0.

* **Multi-Dimensional Array:**  
  Typically used to store data in a tabular form, such as a 2D matrix with rows and columns:

cpp

**int** matrix[3][3] = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

Access elements using two indices: one for row and one for column.

**Q18. Explain string handling in C++ with examples.**

In C++, **string handling** refers to the ways you can create, manipulate, and work with sequences of characters. There are two main approaches to handling strings in C++:

1. **C-style strings** (character arrays)
2. **std::string class** from the C++ Standard Library

## 1. C-Style Strings (Character Arrays)

These are arrays of characters terminated by a null character ('\0'). The <cstring> header provides several functions to manipulate these strings.

## Common C-string functions with examples:

* **strcpy(s1, s2)**: Copies string s2 into s1.

cpp

**char** s1[20] = "Good";

**char** s2[] = "Morning";

strcpy(s1, s2); *// s1 becomes "Morning"*

* **strcat(s1, s2)**: Concatenates string s2 at the end of s1.

cpp

**char** s1[20] = "Good";

**char** s2[] = "Morning";

strcat(s1, s2); *// s1 becomes "GoodMorning"*

* **strlen(s1)**: Returns the length of the string s1.
* **strcmp(s1, s2)**: Compares two strings lexicographically.
* **strchr(s1, ch)**: Finds the first occurrence of character ch in s1.
* **strstr(s1, s2)**: Finds the first occurrence of substring s2 in s1.

Example checking for a character in a string:

cpp

**char** str[] = "Good Morning";

**char** ch = 'a';

**if** (strchr(str, ch))

std::cout << "Character found\n";

**else**

std::cout << "Character not found\n";

These functions operate on raw character arrays and require careful management of memory and buffer sizes[1](https://www.scholarhat.com/tutorial/cpp/strings-in-cpp).

## 2. std::string Class

The C++ Standard Library provides the std::string class which simplifies string handling by managing memory automatically and providing many useful member functions.

## Basic operations with std::string:

* **Declaration and Initialization:**

cpp

#**include** <string>

std::string str1 = "Hello";

std::string str2 = "World";

* **Concatenation:**

cpp

std::string result = str1 + " " + str2; *// "Hello World"*

* **Length:**

cpp

size\_t len = result.length(); *// or result.size()*

* **Access characters:**

cpp

**char** firstChar = result[0];

**char** lastChar = result.back();

* **Substring extraction:**

cpp

std::string sub = result.substr(6, 5); *// "World"*

* **Comparison:**

cpp

**if** (str1 == str2) { */\* equal \*/* }

**else** **if** (str1 < str2) { */\* lex order \*/* }

* **Other useful functions:**
  + append(), insert(), erase(), replace(), find(), clear(), empty(), push\_back(), pop\_back(), and more[2](https://www.w3schools.com/cpp/cpp_ref_string.asp)[4](https://www.tutorialspoint.com/cplusplus/cpp_strings.htm)[5](https://www.demo2s.com/g/cpp/what-basic-operations-can-be-performed-on-cpp-strings-cover-common-operations-like-concat.html).

## Example of iterating over a string:

cpp

std::string s = "Hello";

**for** (**char** c : s) {

std::cout << c << " ";

}

*// Output: H e l l o*

## Summary

|  |  |  |
| --- | --- | --- |
| **Feature** | **C-Style Strings (char arrays)** | **std::string Class** |
| Memory management | Manual, prone to errors | Automatic, safer |
| Null-termination | Required | Not required |
| Functions | strcpy(), strcat(), strlen(), etc. | Member functions like append(), substr(), find(), etc. |
| Ease of use | More complex, error-prone | Easier and more intuitive |
| Flexibility | Limited | Highly flexible with many built-in operations |

**Q19. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.**

## Array Initialization in C++

Arrays in C++ can be initialized at the time of declaration by providing a list of values enclosed in braces {}. The compiler uses these values to set the initial contents of the array.

## 1D Array Initialization

* **Complete Initialization:** Provide values for all elements.

cpp

**int** numbers[] = {10, 20, 30, 40, 50}; *// size inferred as 5*

* **Partial Initialization:** Provide fewer values than the array size; remaining elements are set to zero.

cpp

**int** data[5] = {7, 14}; *// elements: 7, 14, 0, 0, 0*

* **Zero Initialization:** Initialize all elements to zero by specifying {0}.

cpp

**int** zeros[5] = {0}; *// all elements are 0*

* **Character Array Initialization:**

cpp

**char** greeting[] = "Hello"; *// size 6 (includes null terminator '\0')*

## 2D Array Initialization

* **Complete Initialization:** Provide values for all rows and columns.

cpp

**int** matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

* **Partial Initialization:** Provide fewer values; remaining elements are zero-initialized.

cpp

**int** matrix[2][3] = {

{1, 2}, *// third element in first row is 0*

{4} *// second and third elements in second row are 0*

};

* **Omitting Outer Braces (allowed but less clear):**

cpp

**int** matrix[2][3] = {1, 2, 3, 4, 5, 6};

## Examples

cpp

#**include** <iostream>

**using** **namespace** std;

**int** main() {

*// 1D array with complete initialization*

**int** arr1[] = {7, 3, 8, 7, 2};

cout << "arr1 elements: ";

**for** (**int** i = 0; i < 5; i++)

cout << arr1[i] << " ";

cout << endl;

*// 1D array with partial initialization*

**int** arr2[7] = {52, 78, 18, 92};

cout << "arr2 elements: ";

**for** (**int** i = 0; i < 7; i++)

cout << arr2[i] << " "; *// last 3 elements will be 0*

cout << endl;

*// 2D array initialization*

**int** matrix[2][3] = {

{1, 2, 3},

{4, 5} *// last element zero-initialized*

};

cout << "matrix elements:\n";

**for** (**int** i = 0; i < 2; i++) {

**for** (**int** j = 0; j < 3; j++)

cout << matrix[i][j] << " ";

cout << endl;

}

**return** 0;

}

## Summary

|  |  |  |
| --- | --- | --- |
| **Array Type** | **Initialization Syntax** | **Notes** |
| **1D array** | int arr[] = {1, 2, 3}; | Size inferred from initializer list |
|  | int arr[5] = {1, 2}; | Remaining elements zero-initialized |
|  | char str[] = "Hello"; | Includes null terminator |
| **2D array** | int mat[2][3] = {{1, 2, 3}, {4, 5, 6}}; | Complete initialization |
|  | int mat[2][3] = {{1, 2}, {4}}; | Partial initialization, missing elements zeroed |

**Q20. Explain string operations and functions in C++.**

In C++, **string operations** and **functions** are primarily provided by the std::string class from the Standard Library, which offers a rich set of member functions to manipulate text easily and safely.

## Common String Operations and Functions in C++

## 1. ****Concatenation****

Combine two or more strings using the + operator or append() function.

cpp

std::string str1 = "Hello, ";

std::string str2 = "World!";

std::string result = str1 + str2; *// Using +*

str1.append(str2); *// Using append()*

## 2. ****Accessing Characters****

Access individual characters using at() (with bounds checking) or the [] operator.

cpp

**char** firstChar = str1.at(0); *// 'H'*

**char** lastChar = str1.back(); *// last character*

## 3. ****Length and Size****

Get the number of characters using length() or size().

cpp

size\_t len = str1.length();

## 4. ****Substring Extraction****

Extract a portion of a string using substr(startIndex, length).

cpp

std::string sub = str1.substr(0, 5); *// "Hello"*

## 5. ****Comparison****

Compare strings lexicographically with compare() or operators like ==, <, >.

cpp

**if** (str1 == str2) { */\* equal \*/* }

**else** **if** (str1 < str2) { */\* str1 comes before str2 \*/* }

## 6. ****Searching****

Find the first or last occurrence of a character or substring using find() and rfind().

cpp

size\_t pos = str1.find("lo"); *// position of substring "lo"*

size\_t lastPos = str1.rfind('o'); *// last occurrence of 'o'*

## 7. ****Insertion and Erasure****

Insert or remove parts of a string with insert() and erase().

cpp

str1.insert(5, " C++ "); *// Insert " C++ " at index 5*

str1.erase(5, 5); *// Remove 5 characters starting at index 5*

## 8. ****Replacing****

Replace part of a string with new content using replace().

cpp

str1.replace(6, 5, "Everyone"); *// Replace 5 characters at index 6*

## 9. ****Clearing and Checking Empty****

Clear the string or check if it is empty.

cpp

str1.clear(); *// Empties the string*

**bool** isEmpty = str1.empty();

## 10. ****Conversion to C-Style String****

Get a null-terminated C-style string using c\_str().

cpp

**const** **char**\* cstr = str1.c\_str();

## 11. ****Push and Pop Characters****

Add or remove characters at the end.

cpp

str1.push\_back('!');

str1.pop\_back();

## Example Demonstrating Some String Functions

cpp

#**include** <iostream>

#**include** <string>

**int** main() {

std::string greeting = "Hello";

greeting.append(" World");

std::cout << greeting << std::endl; *// Hello World*

std::cout << "Length: " << greeting.length() << std::endl; *// 11*

std::cout << "Character at 1: " << greeting.at(1) << std::endl; *// e*

size\_t pos = greeting.find("World");

**if** (pos != std::string::npos)

std::cout << "'World' found at position: " << pos << std::endl;

std::string sub = greeting.substr(6, 5);

std::cout << "Substring: " << sub << std::endl; *// World*

greeting.replace(6, 5, "C++");

std::cout << "After replace: " << greeting << std::endl; *// Hello C++*

greeting.insert(5, ",");

std::cout << "After insert: " << greeting << std::endl; *// Hello, C++*

greeting.erase(5, 1);

std::cout << "After erase: " << greeting << std::endl; *// Hello C++*

greeting.push\_back('!');

std::cout << "After push\_back: " << greeting << std::endl; *// Hello C++!*

greeting.pop\_back();

std::cout << "After pop\_back: " << greeting << std::endl; *// Hello C++*

**return** 0;

}

## Summary of Key String Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
| append() | Adds characters or another string at the end |
| at() | Returns character at specified index with bounds check |
| back() | Accesses last character |
| clear() | Removes all characters |
| compare() | Compares two strings lexicographically |
| find() | Finds first occurrence of substring or character |
| insert() | Inserts characters or substring at a given position |
| length() | Returns string length |
| replace() | Replaces part of the string |
| substr() | Extracts a substring |
| empty() | Checks if string is empty |
| c\_str() | Returns C-style null-terminated string |
| push\_back() | Adds a character at the end |
| pop\_back() | Removes last character |

These functions make std::string a powerful and flexible tool for string manipulation in C++ programs.

* **6. Introduction to Object-Oriented Programming**

Q21. Explain the key concepts of Object-Oriented Programming (OOP).

Object-Oriented Programming (OOP) is a programming paradigm centered around the concept of **objects**, which encapsulate data and behavior. It organizes software design around **classes** (blueprints) and **objects** (instances), enabling modular, reusable, and maintainable code.

## Key Concepts of OOP

1. **Classes and Objects**
   * **Class:** A blueprint or template that defines the attributes (data) and methods (functions/behaviors) common to all objects of that type.
   * **Object:** An instance of a class representing a specific entity with its own state.  
     Example: A Car class defines properties like make, model, and methods like startEngine(). Each Car object represents a specific car with actual values.
2. **Encapsulation**  
   Encapsulation means bundling data (attributes) and methods that operate on the data into a single unit (class), and restricting direct access to some of the object's components. This protects the internal state and hides complexity, exposing only necessary interfaces.  
   Example: Private variables in a class can only be accessed or modified through public getter/setter methods, preventing unauthorized access.
3. **Inheritance**  
   Inheritance allows a new class (derived or child class) to inherit properties and behaviors from an existing class (base or parent class), promoting code reuse and hierarchical relationships.  
   Example: A SportsCar class can inherit from the Car class and add additional features like turboBoost().
4. **Polymorphism**  
   Polymorphism means "many forms" — it allows objects of different classes related by inheritance to be treated as objects of a common base class, with the ability to override or extend behaviors. It enables dynamic method binding and flexible code.  
   Example: A base class Shape with a method draw(). Derived classes like Circle and Rectangle provide their own implementation of draw(). Calling draw() on a Shape\* pointer will invoke the correct subclass method.
5. **Abstraction**  
   Abstraction involves hiding complex implementation details and showing only the essential features of an object. It simplifies interaction with objects by exposing only relevant information and interfaces.  
   Example: A Car class exposes methods like start() without revealing the complex engine mechanics inside.

## Summary Table

|  |  |  |
| --- | --- | --- |
| **Concept** | **Description** | **Example** |
| **Class** | Blueprint defining attributes and methods | class Car { make, model; startEngine(); } |
| **Object** | Instance of a class with actual data | Car myCar("Toyota", "Corolla") |
| **Encapsulation** | Hiding internal data, exposing only necessary interfaces | Private variables with public getters/setters |
| **Inheritance** | Deriving new classes from existing ones to reuse code | class SportsCar : public Car { ... } |
| **Polymorphism** | Ability to treat objects of different classes uniformly | Virtual functions and method overriding |
| **Abstraction** | Simplifying interface by hiding complex details | Using methods to interact without knowing internals |

## Why OOP?

* **Modularity:** Code is organized into discrete classes and objects.
* **Reusability:** Inheritance and polymorphism enable code reuse.
* **Maintainability:** Encapsulation and abstraction make code easier to manage and evolve.
* **Scalability:** OOP supports building complex systems by modeling real-world entities.

Q22. What are classes and objects in C++? Provide an example.

## Classes and Objects in C++

**Classes** and **objects** are fundamental concepts in C++ and form the basis of object-oriented programming.

**Class:**

* A class is a user-defined data type that acts as a blueprint or template for creating objects.
* It defines the properties (also called attributes or data members) and behaviors (also called methods or member functions) that the objects created from the class will have.
* For example, a class called Car might define attributes like color, model, and year, and methods like drive() and brake()[2](https://www.w3schools.com/cpp/cpp_classes.asp)[4](https://www.programiz.com/cpp-programming/object-class)[5](https://eng.libretexts.org/Courses/Delta_College/C_-_Data_Structures/01:_OOP_Concepts/1.08:_Classes_and_Objects)[6](https://hyperskill.org/university/cpp/cpp-classesobjects).

**Object:**

* An object is an instance of a class. When you create an object, you allocate memory for the data defined in the class and can use the methods to manipulate that data.
* Each object can have different values for its attributes, but they all share the structure and behaviors defined by the class[1](https://unstop.com/blog/difference-between-class-and-object)[2](https://www.w3schools.com/cpp/cpp_classes.asp)[4](https://www.programiz.com/cpp-programming/object-class)[5](https://eng.libretexts.org/Courses/Delta_College/C_-_Data_Structures/01:_OOP_Concepts/1.08:_Classes_and_Objects)[6](https://hyperskill.org/university/cpp/cpp-classesobjects).

## Example in C++

Here is a simple example demonstrating a class and how to create and use objects:

cpp

#**include** <iostream>

**using** **namespace** std;

*// Define a class named Car*

**class** Car {

**public**:

string brand;

string model;

**int** year;

**void** drive() {

cout << "The car is driving." << endl;

}

};

**int** main() {

*// Create an object of Car*

Car car1;

*// Assign values to the object's attributes*

car1.brand = "Ford";

car1.model = "Mustang";

car1.year = 2023;

*// Access the object's attributes and method*

cout << car1.brand << " " << car1.model << " " << car1.year << endl;

car1.drive();

**return** 0;

}

**Explanation:**

* The Car class defines three attributes (brand, model, year) and one method (drive()).
* In main(), car1 is an object (instance) of the Car class.
* We assign values to car1's attributes and call its drive() method[2](https://www.w3schools.com/cpp/cpp_classes.asp)3[4](https://www.programiz.com/cpp-programming/object-class)[5](https://eng.libretexts.org/Courses/Delta_College/C_-_Data_Structures/01:_OOP_Concepts/1.08:_Classes_and_Objects)[6](https://hyperskill.org/university/cpp/cpp-classesobjects).

Q23. What is inheritance in C++? Explain with an example.

## Inheritance in C++

**Inheritance** in C++ is a core object-oriented programming concept that allows one class (called the derived or child class) to inherit properties and behaviors (attributes and methods) from another class (called the base or parent class)[1](https://www.w3schools.com/cpp/cpp_inheritance.asp)[6](https://www.programiz.com/cpp-programming/inheritance)[5](https://www.scholarhat.com/tutorial/cpp/inheritance-in-cpp-programming). This promotes code reusability and helps establish a hierarchical relationship between classes[2](https://www.ibm.com/docs/en/zos/2.4.0?topic=reference-inheritance-c-only)[6](https://www.programiz.com/cpp-programming/inheritance).

## Key Points

* **Base Class (Parent):** The class whose properties are inherited.
* **Derived Class (Child):** The class that inherits from the base class.
* **Syntax:** Inheritance is specified using a colon (:) followed by an access specifier (usually public) and the base class name.

## Example

Here’s a simple example demonstrating inheritance in C++:

cpp

#**include** <iostream>

**using** **namespace** std;

*// Base class*

**class** Animal {

**public**:

**void** eat() {

cout << "I can eat!" << endl;

}

**void** sleep() {

cout << "I can sleep!" << endl;

}

};

*// Derived class*

**class** Dog : **public** Animal {

**public**:

**void** bark() {

cout << "I can bark! Woof woof!!" << endl;

}

};

**int** main() {

Dog dog1;

dog1.eat(); *// Inherited from Animal*

dog1.sleep(); *// Inherited from Animal*

dog1.bark(); *// Defined in Dog*

**return** 0;

}

**Output:**

text

I can eat!

I can sleep!

I can bark! Woof woof!!

In this example, the Dog class inherits the eat() and sleep() methods from the Animal class and adds its own method, bark()[6](https://www.programiz.com/cpp-programming/inheritance)[1](https://www.w3schools.com/cpp/cpp_inheritance.asp)[5](https://www.scholarhat.com/tutorial/cpp/inheritance-in-cpp-programming).

## Why Use Inheritance?

* To promote code reuse by sharing common functionality between classes.
* To implement an is-a relationship (e.g., a Dog is an Animal)[6](https://www.programiz.com/cpp-programming/inheritance).
* To extend or specialize the behavior of existing classes without modifying them[2](https://www.ibm.com/docs/en/zos/2.4.0?topic=reference-inheritance-c-only).

Q24. What is encapsulation in C++? How is it achieved in classes?

## Encapsulation in C++

**Encapsulation** in C++ is the concept of bundling data (attributes) and methods (functions) that operate on that data into a single unit, typically a class. This approach helps organize code, improves maintainability, and protects internal object states from unintended interference or misuse[1](https://www.udacity.com/blog/2021/09/cpp-encapsulation-an-overview.html)[4](https://www.programiz.com/cpp-programming/encapsulation)[6](https://www.mycplus.com/tutorials/cplusplus-programming-tutorials/encapsulation/)[8](https://www.codecademy.com/learn/learn-object-oriented-programming-with-c-plus-plus/modules/encapsulation-in-c-plus-plus/cheatsheet).

## How Encapsulation is Achieved in Classes

Encapsulation is primarily achieved using access specifiers:

* **Private:** Data members declared as private cannot be accessed directly from outside the class. This restricts direct access and enforces data hiding.
* **Public:** Public methods (often called getters and setters) are provided to allow controlled access to private data members[2](https://www.w3schools.com/cpp/cpp_encapsulation.asp)[5](https://beginnersbook.com/2017/09/cpp-encapsulation/).

## Example

cpp

#**include** <iostream>

**using** **namespace** std;

**class** Employee {

**private**:

**int** salary; *// Private data member*

**public**:

*// Setter function to set value*

**void** setSalary(**int** s) {

salary = s;

}

*// Getter function to get value*

**int** getSalary() {

**return** salary;

}

};

**int** main() {

Employee emp;

emp.setSalary(50000); *// Set salary using setter*

cout << emp.getSalary(); *// Get salary using getter*

**return** 0;

}

In this example, salary is private and cannot be accessed directly from outside the class. Instead, setSalary() and getSalary() provide controlled access to the data member[2](https://www.w3schools.com/cpp/cpp_encapsulation.asp)[5](https://beginnersbook.com/2017/09/cpp-encapsulation/).

## Summary

* Encapsulation groups related data and functions in a class.
* It restricts direct access to some of an object's components, which is known as data hiding.
* Access is provided through public methods, ensuring better control and security of data[1](https://www.udacity.com/blog/2021/09/cpp-encapsulation-an-overview.html)[2](https://www.w3schools.com/cpp/cpp_encapsulation.asp)[5](https://beginnersbook.com/2017/09/cpp-encapsulation/)[6](https://www.mycplus.com/tutorials/cplusplus-programming-tutorials/encapsulation/).