**1.Introduction To C++**

1.What Are The Key Differences Between Procedural Programming And Objectorientedprogramming (OOP)?

|  |  |  |
| --- | --- | --- |
| **Feature** | **Procedural Programming (POP)** | **Object-Oriented Programming (OOP)** |
| **Basic Unit Of Program** | Functions | Objects (Instances Of Classes) |
| **Approach** | Top-Down | Bottom-Up |
| **Focus** | On Functions And Logic | On Data And Objects |
| **Data Access** | Data Is Usually Global And Can Be Accessed Anywhere | Data Is Encapsulated Within Objects (Private/Protected) |
| **Security** | Less Secure (No Data Hiding) | More Secure (Uses Encapsulation And Abstraction) |
| **Reusability** | Low (Code Is Tightly Coupled) | High (Can Reuse Classes Via Inheritance & Polymorphism) |
| **Modularity** | Less Modular | Highly Modular (Organized Into Objects/Classes) |
| **Examples** | C, Pascal | C++, Java, Python (OOP Style) |
| **Ease Of Maintenance** | Difficult To Manage In Large Projects | Easier To Manage And Scale Large Applications |
| **Inheritance / Polymorphism** | Not Supported | Fully Supported |

**2.List And Explain The Main Advantages Of OOP Over POP.**

|  |  |  |
| --- | --- | --- |
| **No.** | **Advantage** | **Explanation** |
| 1. | **Encapsulation** | OOP Binds Data And Code Together Inside Objects, Keeping Data Safe From Outside Interference. This Makes The Code More Secure And Easier To Manage. |
| 2. | **Data Hiding** | Only Relevant Parts Of The Data And Methods Are Exposed; Others Remain Hidden Using Access Specifiers (Private, Protected, Public). |
| 3. | **Reusability (Inheritance)** | Existing Classes Can Be Extended To Create New Ones, Saving Time And Effort By Reusing Tested Code. |
| 4. | **Polymorphism** | OOP Allows The Same Function Name Or Operator To Behave Differently Based On Context. This Makes Code More Flexible And Dynamic. |
| 5. | **Modularity** | Code Is Divided Into Smaller, Self-Contained Objects Or Classes. Each Class Handles Its Own Functionality, Making The System Easier To Debug And Update. |
| 6. | **Maintainability** | Well-Structured OOP Code Is Easier To Maintain, Upgrade, And Scale Over Time, Especially In Large Projects. |
| 7. | **Real-World Modeling** | OOP Models Real-World Entities (Like A Car, Student, Etc.), Which Makes The Design More Intuitive And Easier To Understand. |
| 8. | **Flexibility And Extensibility** | You Can Easily Add New Features Without Affecting Existing Code, Thanks To Inheritance And Polymorphism. |
| 9. | **Code Reusability With Less Redundancy** | Since OOP Promotes Reusable Classes And Functions, It Reduces Duplication Of Code. |

**3.Explain The Steps Involved In Setting Up A C++ Development Environment.**

1. **Choose A Code Editor Or IDE**

**Dev C++**

Code::Blocks

Visual Studio Code

2.**Install A C++ Compiler**

3. **Set Up The IDE**

**For Dev C++:**

* Download From Https://Sourceforge.Net/Projects/Orwelldevcpp/
* Install And Open It
* File → New → Source File
* Save The File As .Cpp
* Press F11 To Compile And Run

4. **Write Your First Program**

#Include <Iostream>

Using Namespace Std;

Int Main() {

Cout << "Hello, World!" << Endl;

Return 0;

}

**Tips:**

* Always **Save Files With .Cpp Extension**.
* Make Sure Your **Compiler Is Properly Linked** To The IDE.
* Practice Simple Programs To Verify Setup.

**4. What Are The Main Input/Output Operations In C++? Provide Examples.**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Keyword** | **Description** |
| **Input** | Cin | Takes Input From The Keyboard |
| **Output** | Cout | Displays Output On The Screen |

#Include <Iostream>

Using Namespace Std;

Int Main() {

String Name;

Int Age;

Cout << "Enter Your Name: ";

Cin >> Name;

Cout << "Enter Your Age: ";

Cin >> Age;

Cout << "Hello, " << Name << "! You Are " << Age << " Years Old." << Endl;

Return 0;

}

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

**1.What Are The Different Data Types Available In C++? Explain With Examples.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Type** | **Description** | **Example Value** | **Example Code** |
| Int | Integer Numbers | 10, -5, 0 | Int Age = 20; |
| Float | Floating-Point Numbers (Decimal) | 3.14, -2.5 | Float Pi = 3.14; |
| Double | Double-Precision Float | 3.14159 | Double Area = 3.14159; |
| Char | Single Character | 'A', 'B', '3' | Char Grade = 'A'; |
| Bool | Boolean Values | True, False | Bool Ispassed = True; |
| Void | No Value (Used In Functions) | — | Void Showmessage(); |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int Age = 21;

Float Height = 5.9;

Char Grade = 'A';

Bool Isstudent = True;

Double Pi = 3.1415926535;

Cout << "Age: " << Age << Endl;

Cout << "Height: " << Height << Endl;

Cout << "Grade: " << Grade << Endl;

Cout << "Is Student: " << Isstudent << Endl;

Cout << "Pi: " << Pi << Endl;

Return 0;

}

**2. Explain The Difference Between Implicit And Explicit Type Conversion In C++.**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Implicit Type Conversion** | **Explicit Type Conversion** |
| **Trigger** | Automatically Performed By The Compiler. | Manually Done By The Programmer Using Casting. |
| **Loss Of Precision** | Safe Conversions (Usually No Data Loss). | May Lead To Loss Of Precision Or Data Truncation. |
| **Syntax** | No Special Syntax Required. | Requires Casting Syntax (C-Style Or C++-Style). |
| **Example** | Int Num = 5; Double Result = Num; | Double Pi = 3.14159; Int Intpi = Static\_Cast<Int>(Pi); |

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

**Implicit Type Conversion** Is Automatically Performed By The **Compiler** When You Assign A Value Of One Data Type To A Variable Of Another Data Type. This Happens When The Conversion Is **Safe** And Does Not Lose Any Data. It’s Also Known As **Type Coercion**.

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

**Explicit Type Conversion** (Also Known As **Type Casting**) Occurs When You Manually Instruct The Compiler To Convert One Type Into Another. This Is Necessary When The Conversion Might **Lose Data** Or When It Requires **Manual Control** Over The Conversion Process.

**3. What Are The Different Types Of Operators In C++? Provide Examples Of Each.**

**1. Arithmetic Operators.**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Addition | A + B |
| - | Subtraction | A - B |
| \* | Multiplication | A \* B |
| / | Division (Integer) | A / B |
| % | Modulus (Remainder) | A % B |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int A = 10, B = 3;

Cout << "Sum: " << A + B << Endl; // Addition

Cout << "Difference: " << A - B << Endl; // Subtraction

Cout << "Product: " << A \* B << Endl; // Multiplication

Cout << "Quotient: " << A / B << Endl; // Division

Cout << "Remainder: " << A % B << Endl; // Modulus

Return 0;

}

**2. Relational (Comparison) Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Equal To | A == B |
| != | Not Equal To | A != B |
| > | Greater Than | A > B |
| < | Less Than | A < B |
| >= | Greater Than Or Equal | A >= B |
| <= | Less Than Or Equal | A <= B |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int A = 10, B = 20;

Cout << (A == B) << Endl; // False

Cout << (A != B) << Endl; // True

Cout << (A > B) << Endl; // False

Cout << (A < B) << Endl; // True

Return 0;

}

**3. Logical Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| && | Logical AND | A && B |
|  |  |  |
| ! | Logical NOT | !A |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Bool A = True, B = False;

Cout << (A && B) << Endl; // Logical AND

Cout << (A || B) << Endl; // Logical OR

Cout << (!A) << Endl; // Logical NOT

Return 0;

}

**4. Assignment Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple Assignment | A = B |
| += | Add And Assign | A += B (Equivalent To A = A + B) |
| -= | Subtract And Assign | A -= B |
| \*= | Multiply And Assign | A \*= B |
| /= | Divide And Assign | A /= B |
| %= | Modulus And Assign | A %= B |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int A = 10, B = 5;

A += B; // A = A + B

Cout << "A += B: " << A << Endl; // Output: 15

A -= B; // A = A - B

Cout << "A -= B: " << A << Endl; // Output: 10

Return 0;

}

**5. Bitwise Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| & | Bitwise AND | A & B |
| ` | ` | Bitwise OR |
| ^ | Bitwise XOR | A ^ B |
| ~ | Bitwise NOT | ~A |
| << | Left Shift | A << 1 |
| >> | Right Shift | A >> 1 |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int A = 5, B = 3; // A = 0101, B = 0011 In Binary

Cout << (A & B) << Endl; // Bitwise AND (Output: 1)

Cout << (A | B) << Endl; // Bitwise OR (Output: 7)

Cout << (A ^ B) << Endl; // Bitwise XOR (Output: 6)

Cout << (~A) << Endl; // Bitwise NOT (Output: -6)

Cout << (A << 1) << Endl; // Left Shift (Output: 10)

Cout << (A >> 1) << Endl; // Right Shift (Output: 2)

Return 0;

}

**6. Conditional (Ternary) Operator**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ? : | Ternary (Conditional) | Condition ? Expr1 : Expr2 |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int A = 10, B = 5;

Int Max = (A > B) ? A : B; // If A > B, Max = A, Else Max = B

Cout << "Max: " << Max << Endl; // Output: 10

Return 0;

}

**7. Increment/Decrement Operators**

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| ++ | Increment (Add 1) | A++ Or ++A |
| -- | Decrement (Subtract 1) | A-- Or --A |

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int A = 5;

Cout << A++ << Endl; // Post-Increment: Output 5, A Becomes 6

Cout << ++A << Endl; // Pre-Increment: Output 7

Return 0;

}

**4. Explain The Purpose And Use Of Constants And Literals In C++.**

**1. Constants In C++**

**Constants** Are Named Values That Are Defined Once And Cannot Be Modified During The Execution Of The Program. They Are Often Used To Represent Values That Should Remain Fixed Throughout The Program, Such As Mathematical Constants, Configuration Values, Or Limits That Should Not Change.

**Types Of Constants:**

* **Constant Variables (Const)**: A Constant That Is Created Using The Const Keyword.
* **#Define Constants**: Constants That Are Defined Using The Preprocessor Directive

Example:

#Include <Iostream>

Using Namespace Std;

//#Define PI 3.14159 // Define Constant PI Using #Define

Int Main() {

Const Int MAX\_USERS = 100; // Constant Integer

Cout << "Max Users: " << MAX\_USERS << Endl;

// MAX\_USERS = 150; // Error: Cannot Modify A Constant Variable

Return 0;

}

**2. Literals In C++**

**Literals** Are Fixed Values That Appear Directly In The Source Code. They Are The Actual Values Assigned To Variables Or Used In Expressions And Operations. They Can Be Of Various Types Like Integer Literals, Floating-Point Literals, Character Literals, Etc.

#Include <Iostream>

Using Namespace Std;

Int Main() {

// Integer Literal

Int Age = 25;

// Floating-Point Literal

Double Pi = 3.14159;

// Character Literal

Char Grade = 'A';

// String Literal

String Message = "Hello, World!";

// Boolean Literal

Bool Isactive = True;

// Null Pointer Literal

Int\* Ptr = Nullptr;

// Output The Values

Cout << "Age: " << Age << Endl;

Cout << "Pi: " << Pi << Endl;

Cout << "Grade: " << Grade << Endl;

Cout << "Message: " << Message << Endl;

Cout << "Is Active: " << Isactive << Endl;

Cout << "Pointer: " << Ptr << Endl;

Return 0;

}

**3. Control Flow Statements**

**1. What Are Conditional Statements In C++? Explain The If-Else And Switch Statements**

**Conditional Statements** Are Used To Make Decisions In Your Program. These Statements Allow Your Code To Execute Different Blocks Of Code Based On Certain Conditions. The Most Commonly Used Conditional Statements Are:

### 1.If Statement

Int X = 10;

If (X > 5) {

Cout << "X Is Greater Than 5";

}

**2. If-Else Statement**

Int X = 3;

If (X > 5) {

Cout << "X Is Greater Than 5";

} Else {

Cout << "X Is Not Greater Than 5";

}

### 3.If-Else-If Ladder

Int X = 20;

If (X == 10) {

Cout << "X Is 10";

} Else If (X == 20) {

Cout << "X Is 20";

} Else {

Cout << "X Is Neither 10 Nor 20";

}

**4. Switch Statement**

Int Day = 3;

Switch (Day) {

Case 1:

Cout << "Monday";

Break;

Case 2:

Cout << "Tuesday";

Break;

Case 3:

Cout << "Wednesday";

Break;

Default:

Cout << "Invalid Day";

}

**2. What Is The Difference Between For, While, And Do-While Loops In C++?**

**For**, **While**, And **Do-While** Loops Are Used To Repeatedly Execute A Block Of Code As Long As A Certain Condition Is True. The Difference Lies In How And **When** The Condition Is Checked, And **Where** Initialization And Updates Occur.

**1.For Loop**

Used When The Number Of Iterations Is Known.

For (Int I = 1; I <= 5; I++) {

Cout << I << " ";

}

**2.While Loop**

Used When The Number Of Iterations Is **Not Known In Advance**.

Int I = 1;

While (I <= 5) {

Cout << I << " ";

I++;

}

3.Do-While Loop

Used When You Want The Loop To Run **At Least Once**, Even If The Condition Is False.

Int I = 1;

Do {

Cout << I << " ";

I++;

} While (I <= 5);

**3.How Are Break And Continue Statements Used In Loops? Provide Examples.**

**1.Break Statement**

The Break Statement **Exits The Loop Immediately**, Regardless Of The Condition.

For (Int I = 1; I <= 10; I++) {

If (I == 5) {

Break; // Exit Loop When I Is 5

}

Cout << I << " ";

}

**2. Continue Statement**

The Continue Statement **Skips The Current Iteration** And Jumps To The Next One.

For (Int I = 1; I <= 5; I++) {

If (I == 3) {

Continue; // Skip When I Is 3

}

Cout << I << " ";

}

**3.Used In Other Loops Too**

Int I = 1;

While (I <= 5) {

If (I == 4) {

I++;

Continue; // Skip Printing 4

}

If (I == 6) {

Break;

}

Cout << I << " ";

I++;

}

**4.Explain Nested Control Structures With An Example.**

**Nested Control Structures** Mean Using One Control Structure (Like If, For, While, Etc.) **Inside Another**. You Can Nest: If Inside If,For Inside For,If Inside For, And Vice Versa, Loops Inside Loops, Etc.

## Example 1: Nested If Statements

Int Age = 25;

Int Marks = 85;

If (Age >= 18) {

If (Marks >= 80) {

Cout << "Eligible For Scholarship";

} Else {

Cout << "Not Eligible For Scholarship";

}

} Else {

Cout << "Not Eligible Due To Age";

}

Example 2: Nested For Loop (For Printing Patterns)

For (Int I = 1; I <= 3; I++) {

For (Int J = 1; J <= I; J++) {

Cout << "\* ";

}

Cout << Endl;

}

Example 3: Loop + If (Nested Loop With Condition)

For (Int I = 1; I <= 5; I++) {

If (I % 2 == 0) {

Cout << I << " Is Even\N";

} Else {

Cout << I << " Is Odd\N";

}

}

**4.Functions And Scope**

**1. What Is A Function In C++? Explain The Concept Of Function Declaration, Definition, And Calling.**

A **Function** In C++ Is A **Block Of Reusable Code** That Performs A Specific Task. Functions Help Break Your Program Into Smaller, Manageable Pieces, Which Improves Readability, Reusability, And Debugging.

|  |  |
| --- | --- |
| **Part** | **Description** |
| **Declaration** | Tells The Compiler About The Function's Name, Return Type, And Parameters. |
| **Definition** | Contains The Actual Code (Body) Of The Function. |
| **Calling** | Tells The Program To Execute The Function. |

## Types Of Functions

1. **Library Functions** – Predefined (Like Cout, Sqrt(), Main()).
2. **User-Defined Functions** – Created By The Programmer.

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

// Function Declaration

Int Multiply(Int, Int);

// Main Function

Int Main() {

Int Result = Multiply(4, 5); // Function Call

Cout << "Product: " << Result;

Return 0;

}

// Function Definition

Int Multiply(Int A, Int B) {

Return A \* B;

}

**2.What Is The Scope Of Variables In C++? Differentiate Between Local And Global Scope.**

The **Scope** Of A Variable Refers To **Where** In The Program The Variable Can Be **Accessed Or Used**.

In C++, Variables Can Be:

1. **Local** – Declared Inside A Function Or Block
2. **Global** – Declared Outside Of All Functions

## 1. ****Local Variables****

Declared **Inside** A Function, Loop, Or Block ({ })

**Accessible Only Within** That Block

**Created When** The Block Is Entered

**Destroyed When** The Block Is Exited

\*\*Example\*\*

Void Show() {

Int A = 10; // Local Variable

Cout << "A = " << A;

}

## 2. ****Global Variables****

Declared **Outside** All Functions

**Accessible** From Any Function Within The Same File

**Lifespan** Is The Entire Program Run

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

Int X = 100; // Global Variable

Void Display() {

Cout << "X = " << X;

}

Int Main() {

Cout << "In Main, X = " << X << Endl;

Display();

Return 0;

}

|  |  |  |
| --- | --- | --- |
| **Feature** | **Local Variable** | **Global Variable** |
| Declared In | Inside A Function Or Block | Outside All Functions |
| Scope | Limited To The Block/Function | Entire File/Program |
| Lifetime | Created And Destroyed With Block | Exists Throughout Program |
| Default Value | Garbage (Undefined) | 0 (If Not Initialized Manually) |
| Accessibility | Not Accessible Outside The Block | Accessible In All Functions |

\*\*Example\*\*

Int X = 50; // Global

Int Main() {

Int X = 10; // Local

Cout << "Local X = " << X << Endl;

Cout << "Global X = " << ::X << Endl;

Return 0;

}

**3.Explain Recursion In C++ With An Example.**

**Recursion** Is A Programming Technique Where A Function **Calls Itself** To Solve A Problem.

It Works By **Breaking Down A Big Problem Into Smaller Sub-Problems**, Each Time Calling The Same Function.

**Structure Of A Recursive Function**

Every Recursive Function Must Have:

**Base Case** – A Condition To **Stop** Recursion.

**Recursive Case** – Where The Function **Calls Itself**.

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

Int Factorial(Int N) {

If (N == 0 || N == 1) // Base Case

Return 1;

Else

Return N \* Factorial(N - 1); // Recursive Call

}

Int Main() {

Int Num = 5;

Cout << "Factorial Of " << Num << " Is " << Factorial(Num);

Return 0;

}

**4.What Are Function Prototypes In C++? Why Are They Used?**

**What Are Function Prototypes In C++?**

A **Function Prototype** Is A **Declaration** Of A Function That Tells The Compiler:

* The **Function Name**
* Its **Return Type**
* The **Number And Types Of Parameters**

It Does **Not** Contain The Body Of The Function.

Syntax Of A Function Prototype

Return\_Type Function\_Name(Parameter\_List);

Example:

Int Add(Int, Int); // Function Prototype

|  |  |
| --- | --- |
| **Purpose** | **Explanation** |
| **Tells The Compiler** | So It Knows How The Function Will Be Used In Main() |
| **Allows Calling Before Definition** | Enables You To Use The Function **Before It's Defined** |
| **Helps With Type Checking** | Ensures Arguments Passed Match Expected Types |
| **Improves Code Readability** | Gives A Summary Of All Functions At The Top |

**5. Arrays And Strings**

**1. What Are Arrays In C++? Explain The Difference Between Single-Dimensional And Multi-Dimensional Arrays.**

An **Array** In C++ Is A **Collection Of Elements** Of The **Same Data Type** Stored In **Contiguous Memory Locations**. It Allows Storing Multiple Values Using A Single Variable Name.

Syntax:

Datatype Array\_Name[Size];

Example:

Int Numbers[5]; // Array Of 5 Integers

Accessing Elements

Numbers[0] = 10; // First Element

Cout << Numbers[0]; // Output: 10

|  |  |  |
| --- | --- | --- |
| **Feature** | **1D Array** | **Multi-Dimensional Array** |
| Structure | A Single List Of Elements | An Array Of Arrays (E.G., Table/Grid) |
| Syntax | Int A[5]; | Int Matrix[3][3]; |
| Access Elements | A[2] | Matrix[1][2] |
| Common Use Case | Storing List Of Values | Storing Tables, Matrices, Grids |

Example Of Single-Dimensional Array:

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int Marks[5] = {90, 85, 78, 92, 88};

For (Int I = 0; I < 5; I++) {

Cout << "Mark " << I+1 << ": " << Marks[I] << Endl;

}

Return 0;

}

Example Of Multi-Dimensional Array (2D):

#Include <Iostream>

Using Namespace Std;

Int Main() {

Int Matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

For (Int I = 0; I < 2; I++) {

For (Int J = 0; J < 3; J++) {

Cout << Matrix[I][J] << " ";

}

Cout << Endl;

}

Return 0;

}

**2. Explain String Handling In C++ With Examples.**

In C++, A **String** Is A Sequence Of Characters. There Are Two Main Ways To Handle Strings:

1. **C-Style Strings** (Character Arrays)
2. **C++ String Class** (Part Of The Standard Library)

**1.C-Style Strings**

A C-Style String Is Simply A **Character Array** Terminated By A **Null Character** ('\0'), Which Indicates The End Of The String.

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

Int Main() {

Char Str1[] = "Hello";

Char Str2[] = "World";

// Concatenation (Manually With Loops Or Strcat)

Cout << "Concatenated: " << Str1 << " " << Str2 << Endl;

// Length Of String (Using Strlen Function)

Cout << "Length Of Str1: " << Strlen(Str1) << Endl;

Return 0;

}

**2.C++ String Class**

The C++ Standard Library Provides A More Powerful And Flexible String Class, Which Automatically Manages Memory, Allows Easy String Manipulation, And Offers Several Built-In Functions.

\*\*Example\*\*

#Include <Iostream>

#Include <String>

Using Namespace Std;

Int Main() {

String Str1 = "Hello";

String Str2 = "World";

// Concatenation

String Str3 = Str1 + " " + Str2;

Cout << "Concatenated: " << Str3 << Endl; // Output: Hello World

// Length

Cout << "Length Of Str1: " << Str1.Length() << Endl; // Output: 5

// Accessing Characters

Cout << "First Character: " << Str1[0] << Endl; // Output: H

Cout << "Second Character: " << Str1.At(1) << Endl; // Output: E

// Substring

String Substr = Str3.Substr(6, 5); // Starting From Index 6, Take 5 Characters

Cout << "Substring: " << Substr << Endl; // Output: World

Return 0;

}

**3. How Are Arrays Initialized In C++? Provide Examples Of Both 1D And 2D Arrays.**

In C++, **Arrays** Can Be Initialized In Several Ways, Depending On Whether They Are **Single-Dimensional** (1D) Or **Multi-Dimensional** (E.G., 2D Arrays).

**1.Single-Dimensional Array (1D)**

A Single-Dimensional Array Is Like A **List** Of Elements Of The Same Type.

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

Int Main() {

// Initializing An Array With 5 Integers

Int Arr[5] = {1, 2, 3, 4, 5};

// Printing Elements Of The Array

For (Int I = 0; I < 5; I++) {

Cout << Arr[I] << " ";

}

Return 0;

}

2.**Multi-Dimensional Array (2D Array)**

A Multi-Dimensional Array Is An Array Of Arrays. For Example, A 2D Array Can Be Thought Of As A **Matrix** With Rows And Columns.

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

Int Main() {

// Initializing A 2D Array With 2 Rows And 3 Columns

Int Arr[2][3] = {

{1, 2, 3}, // First Row

{4, 5, 6} // Second Row

};

// Printing Elements Of The 2D Array

For (Int I = 0; I < 2; I++) {

For (Int J = 0; J < 3; J++) {

Cout << Arr[I][J] << " ";

}

Cout << Endl;

}

Return 0;

}

4. Explain String Operations And Functions In C++.

1. String Declaration And Initialization

#Include <Iostream>

#Include <String>

Using Namespace Std;

Int Main() {

String Str = "Hello, World!";

Cout << Str << Endl; // Output: Hello, World!

Return 0;

}

|  |  |
| --- | --- |
| **Function** | **Description** |
| .Append() | Adds A String To The End Of Another String. |
| .Substr() | Extracts A Substring From The Original String. |
| .Find() | Finds The First Occurrence Of A Substring. |
| .Replace() | Replaces Part Of The String With Another String. |
| .Erase() | Removes Characters From The String. |
| .Insert() | Inserts A String At A Specified Position. |
| .Compare() | Compares Two Strings Lexicographically. |
| .Length()/.Size() | Returns The Length Of The String. |
| .At() | Accesses A Character At A Specific Index. |

6. Introduction To Object-Oriented Programming

1. Explain The Key Concepts Of Object-Oriented Programming (OOP).

Object-Oriented Programming (OOP) Is A Programming Paradigm Centered Around The Concept Of **Objects**, Which Are Instances Of **Classes**. It Provides A Way To Structure Programs By Bundling Related Data And Functions Together, Making Code More Reusable, Scalable, And Easier To Maintain.

**1.Class And Object**

**Class**: A Blueprint Or Template For Creating Objects.

**Object**: An Instance Of A Class, Containing Actual Values.

**\*\*Example\*\***

Class Car {

Public:

String Brand;

Int Year;

};

Int Main() {

Car Mycar;

Mycar.Brand = "Toyota";

Mycar.Year = 2020;

}

**2.Encapsulation**

Wrapping Data (Variables) And Methods (Functions) Into A Single Unit (Class).

Access To Data Is Controlled Using **Access Specifiers**: Private, Public, And Protected.

Promotes **Data Hiding** And **Security**.

\*\*Example\*\*

Class Bankaccount {

Private:

Double Balance;

Public:

Void Deposit(Double Amount) {

Balance += Amount;

}

Double Getbalance() {

Return Balance;

}

};

#### 3.Abstraction

Hiding Complex Implementation Details And Showing Only Essential Features To The User.

Achieved Using **Abstract Classes** Or **Interfaces** (Via Pure Virtual Functions).

\*\*Example\*\*

Class Animal {

Public:

Virtual Void Makesound() = 0; // Pure Virtual Function

};

Class Dog : Public Animal {

Public:

Void Makesound() {

Cout << "Woof!" << Endl;

}

};

#### 4.Inheritance

#### A Mechanism To Derive A New Class (Child) From An Existing Class (Parent).

#### Promotes Code Reusability And Hierarchical Classification.

\*\*Example\*\*

Class Animal {

Public:

Void Eat() {

Cout << "Eating..." << Endl;

}

};

Class Dog : Public Animal {

Public:

Void Bark() {

Cout << "Barking..." << Endl;

}

};

**5.Polymorphism**

Ability Of A Function Or Object To Behave In Multiple Ways.

Types:

**Compile-Time Polymorphism**: Function Overloading And Operator Overloading.

**Run-Time Polymorphism**: Function Overriding Using Virtual Functions.

\*\*Example\*\*

// Function Overloading

Class Print {

Public:

Void Show(Int I) {

Cout << "Integer: " << I << Endl;

}

Void Show(String S) {

Cout << "String: " << S << Endl;

}

};

2. What Are Classes And Objects In C++? Provide An Example.

**Class:**

A **Class** In C++ Is A User-Defined Data Type That Acts As A **Blueprint** For Creating **Objects**. It Defines **Data Members** (Variables) And **Member Functions** (Methods) That Operate On Those Variables.

**Object:**

An **Object** Is An **Instance Of A Class**. It Is Created Using The Class, And It Holds Actual Values In Memory.

\*\*Example\*\*

#Include <Iostream>

Using Namespace Std;

// Define A Class Named Car

Class Car {

Public:

String Brand;

Int Year;

// Member Function To Display Car Details

Void Display() {

Cout << "Brand: " << Brand << ", Year: " << Year << Endl;

}

};

Int Main() {

// Create An Object Of Class Car

Car Mycar;

// Assign Values To Data Members

Mycar.Brand = "Toyota";

Mycar.Year = 2022;

// Call The Member Function

Mycar.Display();

Return 0;

}

3. What Is Inheritance In C++? Explain With An Example.

**Inheritance** Is One Of The Core Concepts Of **Object-Oriented Programming (OOP)** In C++. It Allows A Class (Called The **Derived Class** Or **Child Class**) To **Inherit** Properties And Behaviors (Data Members And Member Functions) From Another Class (Called The **Base Class** Or **Parent Class**).

**Why Use Inheritance?**

* Promotes **Code Reusability**
* Supports **Hierarchical Classification**
* Enables **Extending Functionalities** Of Existing Code Without Modifying It

**Types Of Inheritance In C++:**

1. **Single Inheritance**
2. **Multiple Inheritance**
3. **Multilevel Inheritance**
4. **Hierarchical Inheritance**
5. **Hybrid Inheritance**

Example: Single Inheritance

#Include <Iostream>

Using Namespace Std;

// Base Class

Class Animal {

Public:

Void Eat() {

Cout << "This Animal Eats Food." << Endl;

}

};

// Derived Class

Class Dog : Public Animal {

Public:

Void Bark() {

Cout << "The Dog Barks." << Endl;

}

};

Int Main() {

Dog Mydog;

Mydog.Eat(); // Inherited From Animal

Mydog.Bark(); // Own Method

Return 0;

}

4. What Is Encapsulation In C++? How Is It Achieved In Classes?

**Encapsulation** Is The Process Of **Binding Data And Methods** (Functions) That Operate On That Data Into A **Single Unit**, Typically A **Class**.

**Key Features Of Encapsulation:**

* Protects Internal Object State From Unintended Interference
* Provides **Controlled Access** Using **Access Specifiers**:
  + Private: Accessible Only Inside The Class
  + Public: Accessible From Outside The Class
  + Protected: Accessible In The Class And Its Derived Classes

**How Is Encapsulation Achieved?**

Encapsulation Is Implemented Using **Classes**, Where:

* **Data Members** Are Usually Marked Private
* **Public Getter/Setter Functions** Are Used To Access And Modify Those Private Members

Example: Encapsulation In C++

#Include <Iostream>

Using Namespace Std;

Class Student {

Private:

Int Rollnumber; // Private Data Member

String Name;

Public:

// Setter Function To Set Values

Void Setdata(Int R, String N) {

Rollnumber = R;

Name = N;

}

// Getter Function To Display Values

Void Displaydata() {

Cout << "Roll No: " << Rollnumber << ", Name: " << Name << Endl;

}

};

Int Main() {

Student S1;

// Accessing Private Data Through Public Methods

S1.Setdata(101, "Jaydipsinh");

S1.Displaydata();

Return 0;

}