Module #3

Introduction to OOPS Programming:

1.Introduction to C++

THEORY EXERCISE:

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

| **Feature** | **Procedural Programming** | **Object-Oriented Programming (OOP)** |
| --- | --- | --- |
| **Main focus** | Functions (procedures) | Objects (data + methods) |
| **Data security** | Weak (data often global) | Strong (encapsulation, access control) |
| **Code reusability** | Limited (via functions) | High (inheritance, polymorphism) |
| **Program structure** | Top-down | Bottom-up |
| **Examples** | C, Pascal | Java, Python, C++ |
|  | | |

2. List and explain the main advantages of OOP over POP.

1. Code Reusability – Once you create a class, you can use it again and again without writing the same code.

2. Data Security – With encapsulation, data is protected. No one can change it without permission.

3. Easy to Maintain – If you want to change something, you don’t need to touch the whole program. Just update the class.

4. Real-world Modeling – OOP uses objects like car, employee, etc., which are easy to understand and work with.

5. Flexibility – With polymorphism, one function can do different tasks based on the object.

6. Best for Large Projects – OOP is perfect for big software. POP is simple but not good for managing large code.

In short:

OOP = Safe, reusable, easy to maintain, real-life style

POP = Simple but hard to manage in big projects

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

1. Choose an IDE or Text Editor:

Examples: Code::Blocks, Dev C++, Visual Studio, or editors like VS Code.

2. Download and Install the IDE:

Go to the official website of your chosen IDE and download the installer. Then run it to install.

3. Install a C++ Compiler:

Most IDEs come with built-in compilers (like GCC or MinGW). If not, you need to install one separately.

4. Configure the IDE:

Make sure the IDE knows where the compiler is located. Usually, the IDE auto-configures this.

5. Create a New C++ Project:

Open the IDE and start a new C++ project or file.

6. Write Your Code:

Start coding inside the editor.

7. Build/Compile the Program:

Use the IDE’s build or compile option to turn your code into an executable.

8. Run the Program:

After successful compilation, run your program inside the IDE or terminal.

9. Fix Errors (if any):

Check for errors or warnings, fix them, and rebuild.

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

| **Stream** | **Purpose** | **Operator Used** | **Example** |
| --- | --- | --- | --- |
| cout | Standard output (screen) | << | cout << "Hello"; |
| cin | Standard input (keyboard) | >> | cin >> x; |
| cerr | Standard error (unbuffered) | << | cerr << "Error"; |
| clog | Standard log (buffered) | << | clog << "Log info"; |

Example:

#include <iostream>

using namespace std;

int main() {

cout << "Program started..." << endl;

clog << "Processing data..." << endl;

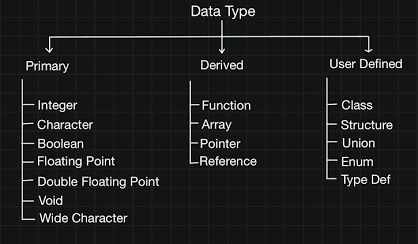
return 0;

}

2. Variables, Data Types, and Operators:

THEORY EXERCISE:

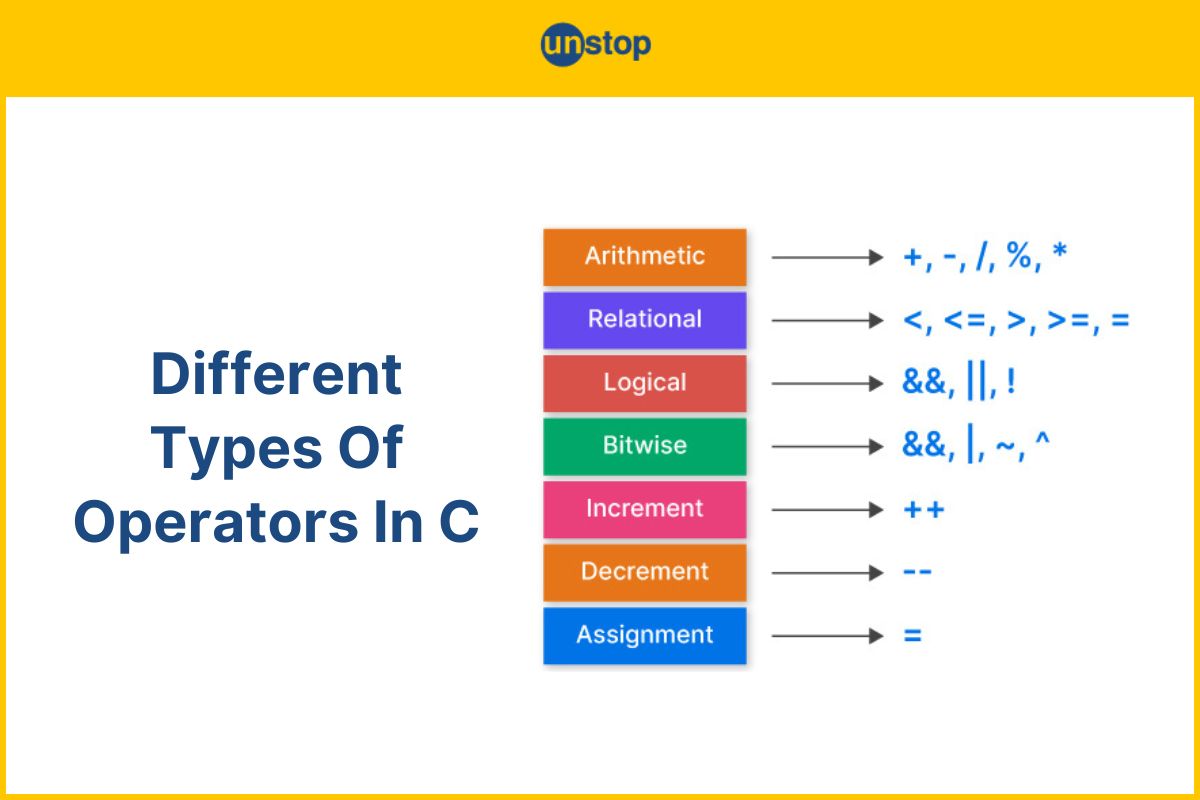
1. What are the different data types available in C++? Explain with examples.



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

| **Feature** | **Implicit Conversion** | **Explicit Conversion** |
| --- | --- | --- |
| **Who does it?** | Compiler automatically | Programmer manually |
| **Control** | Less control | Full control |
| **Syntax** | No special syntax | Requires cast syntax |
| **Risk** | Usually safe (widening conversions) | May cause data loss (narrowing conversions) |
| **Example** | int x = 5; double y = x; | double pi = 3.14; int n = (int)pi; |

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



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

| **Feature** | **Constant** | **Literal** |
| --- | --- | --- |
| **Definition** | A named fixed value | A direct fixed value written in code |
| **Changeable?** | Cannot be changed after definition | Always fixed |
| **Usage** | Declared with const or #define | Directly used in code |
| **Example** | Const int SIZE = 10; | int x = 10; (10 is a literal) |

1. Control Flow Statements:

THEORY EXERCISE:

1. What are conditional statements in C++? Explain the if-else and switch statements.

if (condition) {

// code if condition is true

} else {

// code if condition is false

}

Example:

cpp

int age = 18;

if (age >= 18)

{

cout << "You can vote.";

}

else

{

cout << "You are too young to vote.";

}

---

2. switch statement

Used when you have multiple conditions based on the value of a single variable.

Syntax:

cpp

switch (variable)

{

case value1:

// code

break;

case value2:

// code

break;

default:

// code if no case matches

}

Example:

cpp

int day = 3;

switch (day)

{

case 1: cout << "Monday"; break;

case 2: cout << "Tuesday"; break;

case 3: cout << "Wednesday"; break;

default: cout << "Invalid day";

}

Summary:

- if-else is best for range-based or complex conditions.

- switch is cleaner when checking fixed values of a single variable.

1. What is the difference between for, while, and do-while loops in C++?

1. for loop

- Used when you know how many times to repeat something.

- Initialization, condition, and update are in one line.

Syntax:

cpp

for (int i = 1; i <= 5; i++) {

cout << i << " ";

}

---

2. while loop

- Used when the number of repetitions is not known in advance.

- Condition is checked first — if false at the beginning, loop may never run.

Syntax:

cpp

int i = 1;

while (i <= 5)

{

cout << i << " ";

i++;

}

---

3. do-while loop

- Like while, but the loop runs at least once even if the condition is false.

- Condition is checked after the body runs.

Syntax:

cpp

int i = 1;

do

{

cout << i << " ";

i++;

} while (i <= 5);

1. How are break and continue statements used in loops? Provide examples.

1. \*break Statement\*

- Used to exit a loop immediately.

- Skips all remaining iterations and jumps out of the loop.

Example:

cpp

for (int i = 1; i <= 10; i++) {

if (i == 5)

break; // loop stops when i is 5

cout << i << " ";

}

// Output: 1 2 3 4

---

2. \*continue Statement\*

- Skips the current iteration and moves to the next one.

- Loop continues running, but skips code after continue for that iteration.

Example:

cpp

for (int i = 1; i <= 5; i++) {

if (i == 3)

continue; // skip printing 3

cout << i << " ";

}

// Output: 1 2 4 5

1. Explain nested control structures with an example.

Example: Nested if statement

int a = 5, b = 10;

if (a < b)

{

if (a % 2 == 1)

{

cout << "a is smaller than b and odd";

}

}

---

Example: Nested loop (for loop)

Print a right-angled triangle of stars:

cpp

for (int i = 1; i <= 5; i++)

{

for (int j = 1; j <= i; j++)

{

cout << "\* ";

}

cout << endl;

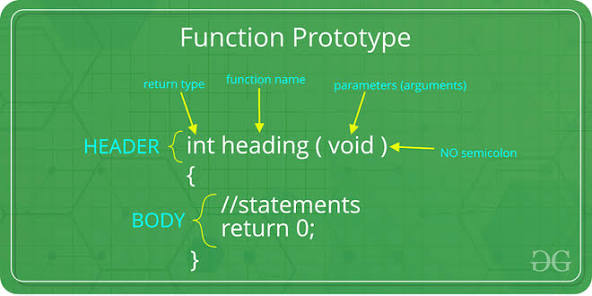
}

1. Functions and Scope:

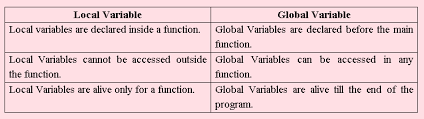
THEORY EXERCISE:

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

A function in C++ is a block of code that performs a specific task and can be executed (or "called") from other parts of a program



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



#include <iostream>

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

void anotherFunc() {

cout << "Inside anotherFunc: " << globalVar << std::endl;

}

int main() {

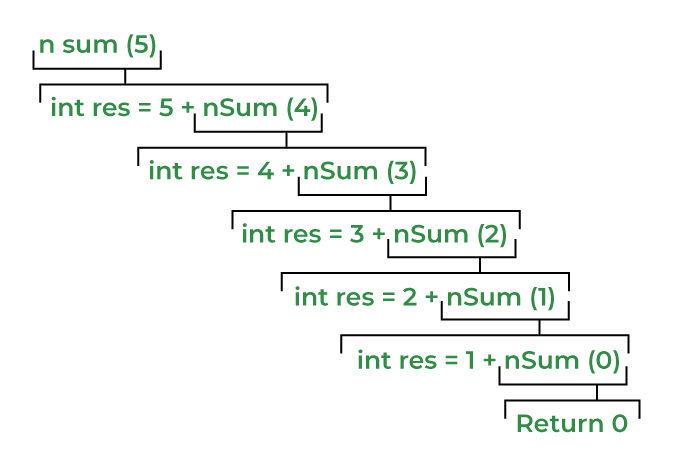
cout << "Inside main: " << globalVar << std::endl;

anotherFunc();

return 0;

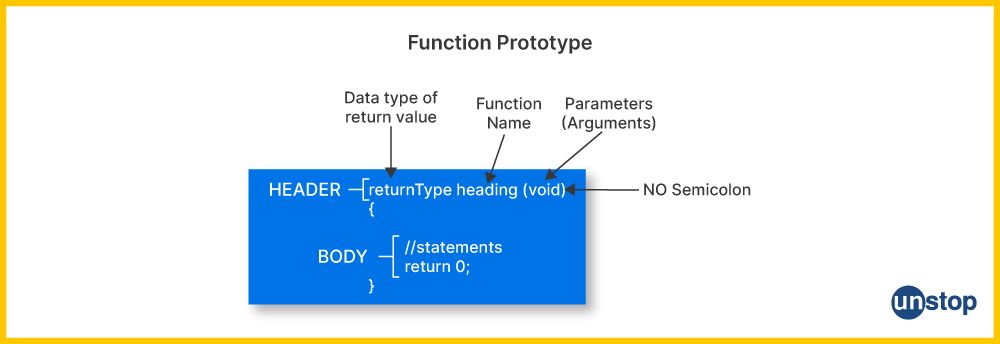
}

1. Explain recursion in C++ with an example.



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

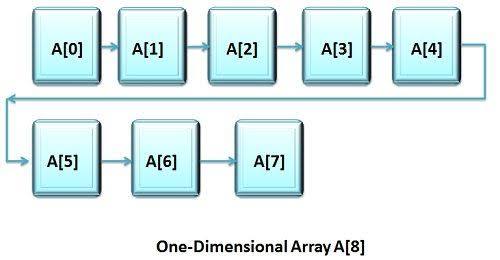
A function prototype in C++ is a declaration that informs the compiler about a function's existence before it is called.

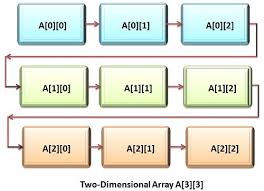


1. Arrays and Strings:

LAB EXERCISES:

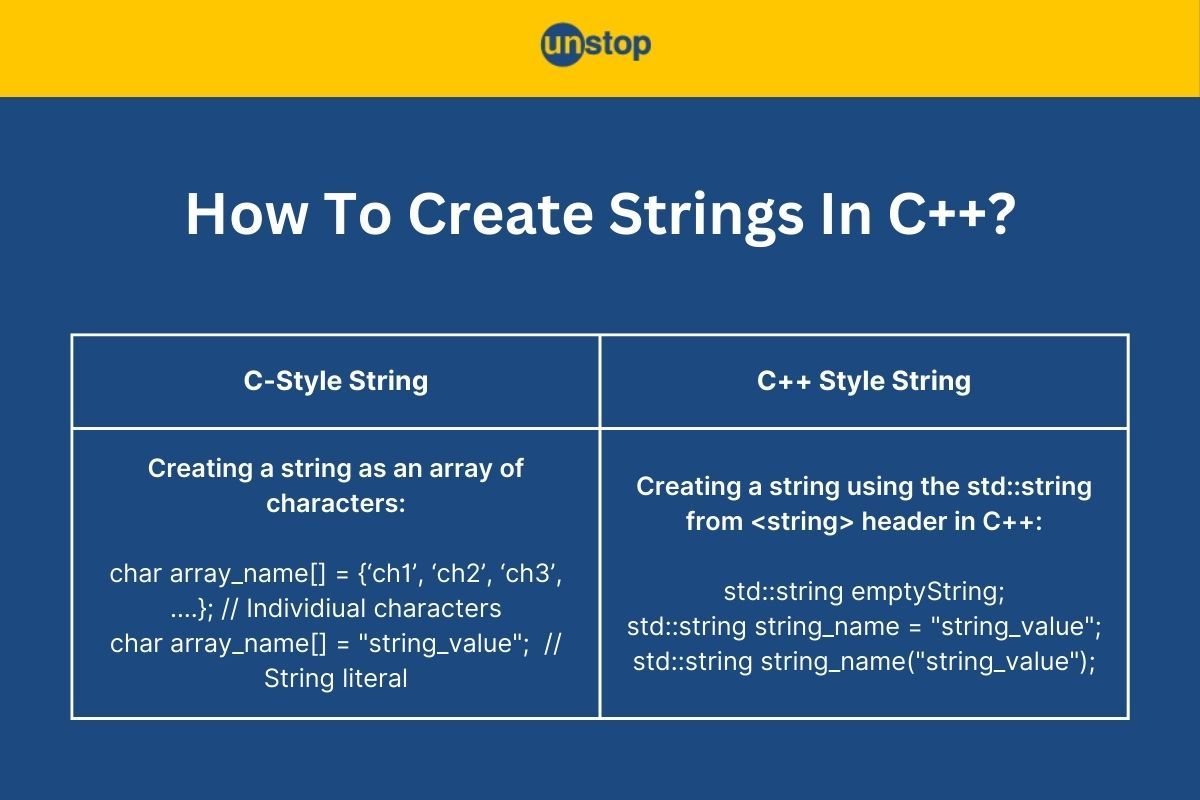
1. What are arrays in C++? Explain the difference between single-dimensional and multi- dimensional arrays.





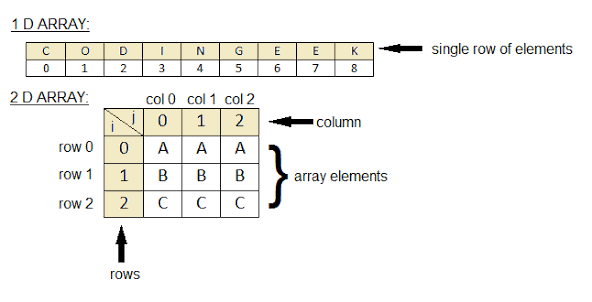
1. Explain string handling in C++ with examples.

C++ offers robust string handling primarily through the std::string class, found in the <string> header.

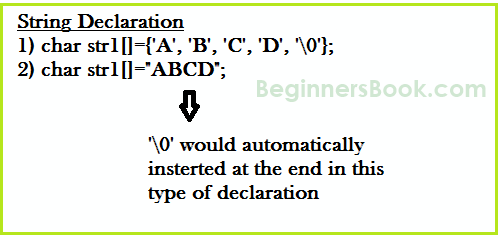


1. How are arrays initialized in C++? Provide examples of both 1D and 2D arrays.

Arrays in C++ are typically initialized using an initializer list with curly braces {} at the time of declaration. If an array is not fully initialized, the uninitialized elements are set to 0 for static arrays or to an undefined value for local arrays.



1. Explain string operations and functions in C++.



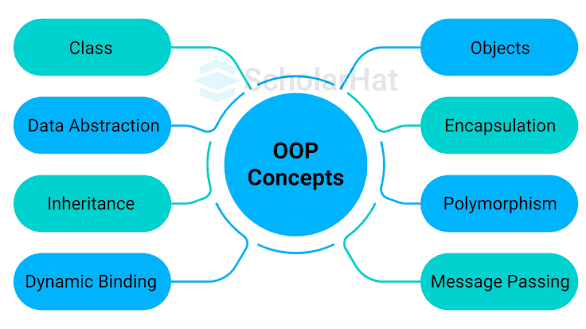
**String is an array of characters**. In this guide, we learn how to declare strings, how to work with strings in C programming and how to use the pre-defined string handling functions.

6.Introduction to Object-Oriented Programming.

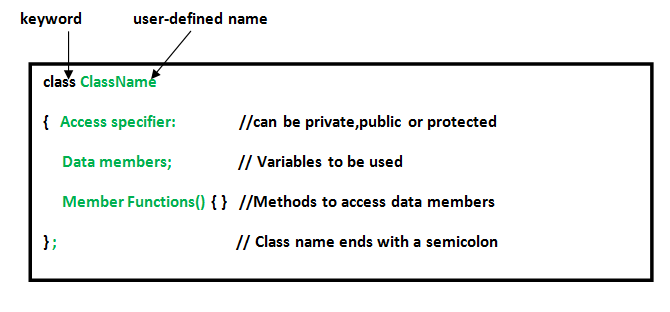
THEORY EXERCISE:

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

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around data, or objects, rather than functions and logic.

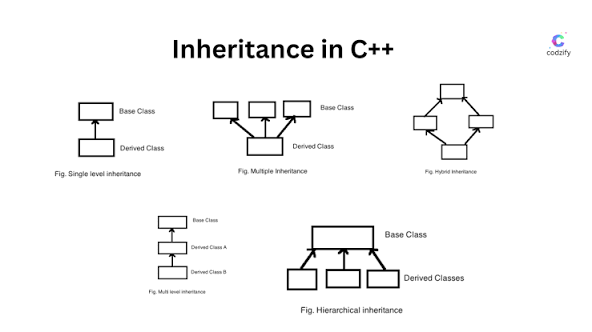


1. What are classes and objects in C++? Provide an example.



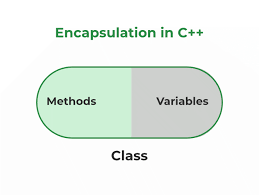
C++, a class is a blueprint for creating objects, while an object is an instance of a class. Classes are fundamental to Object-Oriented Programming (OOP), as they allow you to bundle related data and functions into a single, user-defined data type.

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



Inheritance is a fundamental principle of Object-Oriented Programming (OOP) in C++ that allows a new class, or derived class, to inherit properties and characteristics from an existing class, or base class.

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



Encapsulation in C++ is a fundamental principle of object-oriented programming (OOP) that involves bundling data (member variables) and the functions (member methods) that operate on that data into a single unit, which is a class.