Module #3 Introduction to OOPS Programming

1. Introduction to C++ :

THEORY EXERCISE:

1. **What are the key differences between Procedural Programming and ObjectOrientedProgramming (OOP)?**

Ans : Procedural Programming:

1. Focus on procedures: The program is broken down into procedures or functions that perform specific tasks.

2. Data is separate from procedures: Data is stored in separate variables or data structures, and procedures operate on that data.

3. No encapsulation: Data is accessible from anywhere in the program.

4. No inheritance: Code reuse is achieved through copy-paste or by writing separate functions.

5. No polymorphism: Functions with the same name must have different parameters to avoid conflicts.

Object-Oriented Programming (OOP):

1. Focus on objects: The program is organized around objects that contain data and procedures that operate on that data.

2. Data and procedures are bundled together: Objects encapsulate data and procedures, making it harder for other parts of the program to access or modify the data directly.

3. Encapsulation: Objects hide their internal implementation details from the outside world.

4. Inheritance: Objects can inherit properties and behavior from parent objects, promoting code reuse.

5. Polymorphism: Objects can take on multiple forms, depending on the context in which they are used.

Key benefits of OOP:

1. Modularity: OOP promotes modular code that is easier to maintain and extend.

2. Reusability: OOP enables code reuse through inheritance and polymorphism.

3. Easier maintenance: OOP's encapsulation and abstraction make it easier to modify and extend code without affecting other parts of the program.

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

Ans :

Advantages of OOP over POP:

1. Modularity: OOP allows for modular code, where each object is a self-contained unit that can be easily modified or replaced without affecting other parts of the program.

2. Reusability: OOP enables code reuse through inheritance and polymorphism, reducing the amount of code that needs to be written and maintained.

3. Easier Maintenance: OOP's encapsulation and abstraction make it easier to modify and extend code without affecting other parts of the program.

4. Improved Code Organization: OOP promotes a more organized and structured approach to coding, making it easier to understand and navigate large codebases.

5. Better Error Handling: OOP's encapsulation and exception handling mechanisms make it easier to handle errors and exceptions in a more robust and reliable way.

6. Enhanced Security: OOP's encapsulation and data hiding mechanisms make it more difficult for unauthorized access or modification of sensitive data.

7. Improved Scalability: OOP's modular and reusable code makes it easier to scale up or down to meet changing requirements.

8. Faster Development: OOP's reusable code and modular design make it possible to develop software faster and more efficiently.

9. Easier Testing: OOP's modular design and encapsulation make it easier to test individual components or objects without affecting other parts of the program.

10. Improved Readability: OOP's structured and organized approach to coding makes it easier to read and understand code, reducing the risk of errors and misinterpretation.

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

Ans :

Step 1: Choose a Text Editor or IDE

- A text editor or Integrated Development Environment (IDE) is necessary for writing, editing, and managing C++ code.

- Popular choices for C++ development include:

  - Visual Studio Code (VS Code)

  - Visual Studio (VS)

  - dev c++

Step 2: Install a C++ Compiler

- A C++ compiler is required to translate C++ code into machine code that can be executed by the computer.

- Popular C++ compilers include:

  - GCC (GNU Compiler Collection)

  - Clang

  - Visual Studio Compiler (for Windows)

  - Xcode Compiler (for macOS)

- On Linux or macOS, GCC or Clang can be installed using the package manager.

- On Windows, Visual Studio or MinGW can be installed to provide a C++ compiler.

Step 3: Install a Build System (Optional)

- A build system helps manage the compilation process, especially for larger projects.

- Popular build systems for C++ include:

- CMake
- Meson
- Autotools

- If you're using an IDE, it may have a built-in build system or integration with a popular build system.

Step 4: Install a Debugger (Optional)

- A debugger helps identify and fix errors in the code.
- Popular debuggers for C++ include:
  - GDB (GNU Debugger)
  - LLDB (Low-Level Debugger)
  - Visual Studio Debugger (for Windows)
- If you're using an IDE, it may have a built-in debugger or integration with a popular debugger.

Step 5: Verify the Installation

- Once all the necessary tools are installed, verify that they're working correctly.
- Write a simple C++ program, compile it, and run it to ensure that everything is set up correctly.

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

Ans :

Input Operations

1. cin: The cin object is used to read input from the standard input stream (usually the keyboard).

```
#include <iostream>
int main() {
int num;
std::cout << "Enter a number: ";
std::cin >> num;
std::cout << "You entered: " << num << std::endl;
return 0;
}
```

2. **getline**: The `getline` function is used to read a line of text from the standard input stream.

```
#include <iostream>
#include <string>
int main() {
    std::string name;
    std::cout << "Enter your name: ";
    std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!" << std::endl;
    return 0;
```

```
}
```

3. *scanf*: The `scanf` function is used to read formatted input from the standard input stream.

```
#include <cstdio>
int main() {
int num;
printf("Enter a number: ");
scanf("%d", &num);
printf("You entered: %d\n", num);
return 0;
}
```

Output Operations

1. *cout*: The cout object is used to write output to the standard output stream (usually the screen).

```
#include <iostream>
int main() {
    int num = 10;
    std::cout << "The number is: " << num << std::endl;
    return 0;
}
```

2. printf: The printf function is used to write formatted output to the standard output stream.

```
#include <cstdio>
int main() {
int num = 10;
printf("The number is: %d\n", num);
return 0;
}
```

3. **cerr**: The `cerr` object is used to write error messages to the standard error stream (usually the screen).

```
#include <iostream>
int main() {
   std::cerr << "An error occurred!" << std::endl;
   return 1;
}
```

File Input/Output Operations

1. *ifstream*: The `ifstream` class is used to read from a file.

```
#include <iostream>
#include <fstream>
```

```
int main() {
std::ifstream file("example.txt");
if (file.is_open()) {
std::string line;
while (std::getline(file, line)) {
std::cout << line << std::endl;
}
file.close();
} else {
std::cout << "Unable to open file!" << std::endl;
}
return 0;
}
```

2. *ofstream*: The ofstream class is used to write to a file.

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream file("example.txt");
    if (file.is_open()) {
        file << "Hello, World!" << std::endl;
        file.close();
```

```
    } else {
        std::cout << "Unable to open file!" << std::endl;
    }
    return 0;
}
```

2. Variables, Data Types, and Operators :

THEORY EXERCISE:

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

Ans :

1. Built-in Data Types

These are the fundamental data types provided by C++.

a) Integer (int)

Used to store whole numbers.

```
int age = 25;
```

b) Floating-Point (float, double)

Used to store decimal numbers.

```
float pi = 3.14;      // 4 bytes
double price = 99.99; // 8 bytes
```

c) Character (char)

Used to store single characters (enclosed in single quotes).

```
char grade = 'A';
```

d) Boolean (bool)

Stores true or false values.

```
bool isPassed = true;
```

e) void

Represents an empty type, mainly used for functions that do not return a value.

```
void showMessage() {
   cout << "Hello, World!";
}
```

---

2. Derived Data Types

Derived from fundamental data types.

a) Arrays

A collection of elements of the same type.

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

b) Pointers

Stores the memory address of another variable.

```
int num = 10;
int* ptr = &num;
```

c) References

An alias for another variable.

```
int x = 5;
int& ref = x;
```

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

Ans :

Implicit Type Conversion :

Implicit type conversion, also known as automatic type conversion, occurs when the compiler automatically converts a value or expression from one data type to another without any explicit casting. This type of conversion is performed by the compiler when the data type of a value or expression is not compatible with the operation being performed.

Example of Implicit Type Conversion :

```
int x = 10;
double y = x;  // Implicit conversion from int to double
```

Explicit Type Conversion :

Explicit type conversion, also known as casting, occurs when a programmer explicitly converts a value or expression from one data type to another using a casting operator. This type of conversion is performed by the programmer to ensure that the data type of a value or expression is compatible with the operation being performed.

Example of Explicit Type Conversion

```
double x = 10.5;
int y = (int)x;  // Explicit conversion from double to int
```

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

Ans :

1. Arithmetic Operators :

Arithmetic operators are used to perform mathematical operations.

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulus)
- ++ (increment)
- -- (decrement)

Example:

```
int x = 10;
int y = 3;

int sum = x + y;  // sum = 13
int difference = x - y;  // difference = 7
int product = x * y;  // product = 30
int quotient = x / y;  // quotient = 3
int remainder = x % y;  // remainder = 1
```

2. Assignment Operators :

Assignment operators are used to assign a value to a variable.

- = (assignment)
- += (addition assignment)
- -= (subtraction assignment)
- *= (multiplication assignment)
- /= (division assignment)
- %= (modulus assignment)

Example:

```
int x = 10;
x = 20;  // x = 20
x += 5;  // x = 25
x -= 3;  // x = 22
x *= 2;  // x = 44
x /= 2;  // x = 22
x %= 3;  // x = 1
```

3. Comparison Operators :

Comparison operators are used to compare two values.

- == (equal to)
- != (not equal to)
- > (greater than)

- < (less than)

- >= (greater than or equal to)

- <= (less than or equal to)

Example:

```
int x = 10;

int y = 20;


bool isEqual = (x == y);  // isEqual = false

bool isNotEqual = (x != y);  // isNotEqual = true

bool isGreater = (x > y);  // isGreater = false

bool isLess = (x < y);  // isLess = true

bool isGreaterOrEqual = (x >= y);  // isGreaterOrEqual = false

bool isLessOrEqual = (x <= y);  // isLessOrEqual = true
```

4. Logical Operators :

Logical operators are used to perform logical operations.

- && (logical and)

- || (logical or)

- ! (logical not)

Example:

```
bool x = true;
bool y = false;

bool andResult = x && y;  // andResult = false
bool orResult = x || y;  // orResult = true
bool notResult = !x;  // notResult = false
```

5. Bitwise Operators :

Bitwise operators are used to perform bitwise operations.

- & (bitwise and)
- | (bitwise or)
- ^ (bitwise xor)
- ~ (bitwise not)
- << (left shift)
- >> (right shift)

Example:

```
int x = 10;  // binary: 1010
int y = 6;  // binary: 0110

int andResult = x & y;  // andResult = 2 (binary: 0010)
int orResult = x | y;  // orResult = 14 (binary: 1110)
int xorResult = x ^ y;  // xorResult = 12 (binary: 1100)
```

int notResult = ~x; // notResult = -11 (binary: 11111111111111111111111111110101)

int leftShiftResult = x << 2; // leftShiftResult = 40 (binary: 101000)

int rightShiftResult = x >> 1; // rightShiftResult = 5 (binary: 101)

6. Member Access Operators :

Member access operators are used to access members of a class or struct.

- . (dot operator)
- -> (arrow operator)
- :: (scope resolution operator)

Example:

```
struct Person {
   int age;
   char name[20];
};

Person person;
person.age = 30;  // access member using dot operator
Person* personPtr = &person;
personPtr->age = 30;  // access member using arrow operator

int Person::age = 30
```

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

Ans :

Constants :

Constants are values that are assigned a name and remain unchanged throughout the program. Constants can be:

- Integer constants: Represent whole numbers, e.g., const int MAX_SIZE = 100;
- Floating-point constants: Represent decimal numbers, e.g., const double PI = 3.14159;
- Character constants: Represent single characters, e.g., const char DELIMITER = ',';
- String constants: Represent sequences of characters, e.g., const std::string COPYRIGHT = "2022 MyCompany";
- Enum constants: Represent a set of named values, e.g., enum Color { RED, GREEN, BLUE };

Literals :

Literals are values that are represented directly in the code, without being assigned a name. Literals can be:

- Integer literals: Represent whole numbers, e.g., 123
- Floating-point literals: Represent decimal numbers, e.g., 3.14159
- Character literals: Represent single characters, e.g., 'A'
- String literals: Represent sequences of characters, e.g., "Hello, World!"
- Boolean literals: Represent true or false values, e.g., true or false

Literals are used directly in expressions and statements.

Purpose and Use :

Constants and literals serve several purposes:

- Readability: Constants and literals make the code more readable by providing a clear understanding of the values being used.

- Maintainability: Constants can be easily modified or updated in one place, without affecting the entire program.

- Reusability: Constants can be reused throughout the program, reducing code duplication.

- Type safety: Constants and literals help ensure type safety by preventing incorrect assignments or operations.

3. Control Flow Statements :

THEORY EXERCISE:

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

Ans :

1. if-else Statement :

The if-else statement is used to execute a block of code if a condition is true, and another block of code if the condition is false.

Syntax :

```
if (condition) {
   // code to execute if condition is true
```

```
} else {
   // code to execute if condition is false
}
```

Example :

```
int x = 10;
if (x > 5) {
   std::cout << "x is greater than 5";
} else {
   std::cout << "x is less than or equal to 5";
}
```

2. switch Statement :

The switch statement is used to execute a block of code based on the value of a variable or expression.

Syntax :

```
switch (expression) {
   case value1:
      // code to execute if expression equals value1
      break;
   case value2:
      // code to execute if expression equals value2
      break;
```

```
    default:
        // code to execute if expression does not equal any value
        break;
}
```

Example :

```
int day = 2;
switch (day) {
    case 1:
        std::cout << "Monday";
        break;
    case 2:
        std::cout << "Tuesday";
        break;
    case 3:
        std::cout << "Wednesday";
        break;
    default:
        std::cout << "Invalid day";
        break;
}
```

Nested if-else Statements :

if-else statements can be nested to create more complex conditional logic.

Example

```
int x = 10;
int y = 5;
if (x > 5) {
   if (y > 5) {
      std::cout << "Both x and y are greater than 5";
   } else {
      std::cout << "x is greater than 5, but y is not";
   }
} else {
   std::cout << "x is not greater than 5";
}
```

Conditional Operator (Ternary Operator) :

The conditional operator, also known as the ternary operator, is a shorthand way of writing a simple if-else statement.

Syntax :

condition ? expression_if_true : expression_if_false

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

Ans :

1. for Loop :

A for loop is used to iterate over a block of code for a specified number of times. It consists of three parts: initialization, condition, and increment/decrement.

Syntax :

```
for (initialization; condition; increment/decrement) {
   // code to execute
}
```

Example :

```
for (int i = 0; i < 5; i++) {
   std::cout << i << " ";
}
```

2. while Loop :

A while loop is used to execute a block of code as long as a specified condition is true.

Syntax :

```
while (condition) {
   // code to execute
}
```

Example :

```
int i = 0;
while (i < 5) {
   std::cout << i << " ";
   i++;
}
```

3. do-while Loop :

A do-while loop is similar to a while loop, but it executes the code block at least once before evaluating the condition.

Syntax :

```
do {
   // code to execute
} while (condition);
```

Example :

```
int i = 0;
do {
```

```
    std::cout << i << " ";
    i++;
} while (i < 5);
```

Key differences :

- Initialization: for loops have an initialization part, while while and do-while loops do not.

- Condition evaluation: for and while loops evaluate the condition before executing the code block, while do-while loops evaluate the condition after executing the code block.

- Usage: for loops are commonly used for iterating over arrays or vectors, while while and do-while loops are used for more general-purpose looping.

3. How are break and continue statements used in loops?

Ans :

1. break Statement :

The break statement is used to exit a loop prematurely. When a break statement is encountered, the loop is terminated, and the program continues to execute the code after the loop.

Example: Using break in a for Loop

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        break;
```

```
    }
    cout << i << " ";
}
```

In this example, the loop will only iterate from i = 0 to i = 2, and then exit the loop when i equals 3.

Example: Using break in a while Loop

```
int i = 0;
while (i < 5) {
    if (i == 3) {
        break;
    }
    cout << i << " ";
    i++;
}
```

Example: Using break in a switch Statement

```
int x = 2;
switch (x) {
    case 1:
        cout << "x is 1";
```

```
        break;
    case 2:
        cout << "x is 2";
        break;
    default:
        cout << "x is not 1 or 2";
        break;
}
```

2. continue Statement :

The continue statement is used to skip the current iteration of a loop and move on to the next iteration.

Example: Using continue in a for Loop

```
for (int i = 0; i < 5; i++) {
    if (i == 3) {
        continue;
    }
    cout << i << " ";
}
```

Example: Using continue in a while Loop

```
int i = 0;
while (i < 5) {
   if (i == 3) {
      i++;
      continue;
   }
   cout << i << " ";
   i++;
}
```

4. Provide examples. 4. Explain nested control structures with an example.

Ans :

Nested if-else Statement :

```
int x = 10;
int y = 5;

if (x > 5) {
   if (y > 5) {
      std::cout << "Both x and y are greater than 5";
   } else {
      std::cout << "x is greater than 5, but y is not";
   }
```

```
} else {
   std::cout << "x is not greater than 5";
}
```

Nested Loop :

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 2; j++) {
      std::cout << i << " " << j << std::endl;
   }
}
```

Nested switch Statement :

```
int x = 2;
int y = 1;

switch (x) {
   case 1:
      switch (y) {
         case 1:
            std::cout << "x is 1 and y is 1";
            break;
         case 2:
            std::cout << "x is 1 and y is 2";
```

```
            break;
        }
        break;
    case 2:
        std::cout << "x is 2";
        break;
}
```

4. Functions and Scope :

THEORY EXERCISE:

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

Ans :

a function is a block of code that performs a specific task and can be called multiple times from different parts of a program.

Function Declaration :

A function declaration, also known as a function prototype, is a statement that specifies the function's name, return type, and parameter list. It tells the compiler that a function with the specified name and parameters exists, and that it will be defined later.

Syntax :

return-type function-name(parameter-list);

Example :

```
int add(int, int);
```

This declares a function named add that takes two int parameters and returns an int value.

Function Definition :

A function definition is the actual implementation of the function. It includes the function's body, which contains the code that performs the desired task.

Syntax :

```
return-type function-name(parameter-list) {
   // function body
}
```

Example :

```
int add(int x, int y) {
   return x + y;
}
```

This defines the add function, which takes two int parameters, adds them together, and returns the result.

Function Calling :

A function call is a statement that invokes a function, passing any required arguments to it.

Syntax :

function-name(argument-list);

Example :

```
int result = add(5, 3);
```

This calls the add function, passing 5 and 3 as arguments, and assigns the returned result to the result variable.

Key Concepts :

Here are some key concepts related to functions in C++:

- Function signature: The combination of a function's name, return type, and parameter list.
- Function overloading: The ability to define multiple functions with the same name but different parameter lists.
- Function default arguments: The ability to specify default values for function arguments, which can be omitted when calling the function.

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

Ans :

In C++, the scope of a variable refers to the region of the program where the variable is defined and can be accessed.

Local Scope:

Variables with local scope are defined within a block, such as a function, loop, or conditional statement. They are only accessible within that block and are destroyed when the block is exited.

Example :

```
void myFunction() {
   int x = 10;  // local variable
   std::cout << x;
}

int main() {
   myFunction();
   // x is not accessible here
   return 0;
}
```

Global Scope :

Variables with global scope are defined outside of any block, typically at the top of a source file. They are accessible from any part of the program and retain their values until the program terminates.

Example :

```
int x = 10;  // global variable
void myFunction() {
   std::cout << x;  // accesses the global variable
}

int main() {
   myFunction();
   std::cout << x;  // accesses the global variable
   return 0;
}
```

Key differences :

- Accessibility: Local variables are only accessible within the block where they are defined, while global variables are accessible from any part of the program.

- Lifetime: Local variables are destroyed when the block is exited, while global variables retain their values until the program terminates.

- Namespace: Local variables are part of the block's namespace, while global variables are part of the global namespace.

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

Ans :

Recursion is a programming technique where a function calls itself repeatedly until it reaches a base case that stops the recursion.

How Recursion Works :

Here's a step-by-step explanation of how recursion works:

1. Function Call: A function calls itself, passing in arguments if necessary.

2. Base Case: The function checks if it has reached a base case, which is a condition that stops the recursion.

3. Recursive Call: If the base case is not met, the function calls itself again, passing in updated arguments if necessary.

4. Return: The function returns the result of the recursive call, or the base case value if the recursion has stopped.

Factorial Function :

Here's an example of a recursive function that calculates the factorial of a given number:

#include <iostream>

```
// Recursive function to calculate factorial
int factorial(int n) {
    // Base case: factorial of 0 or 1 is 1
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive call: n! = n * (n-1)!
    else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    std::cout << "Factorial of " << num << " is " << factorial(num);
    return 0;
}
```

Advantages and Disadvantages of Recursion :

Advantages:

- Elegant solutions: Recursion can provide elegant solutions to complex problems.

- Easy to implement: Recursive functions can be easy to implement, especially for problems with a recursive structure.

Disadvantages:

- Inefficient: Recursion can be inefficient, especially for large problems, since each recursive call creates a new stack frame.

- Risk of stack overflow: Deep recursion can cause a stack overflow, especially if the recursive function calls itself too many times.

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

Ans :

In C++, a function prototype, also known as a function declaration, is a statement that specifies the function's name, return type, and parameter list. It tells the compiler that a function with the specified name and parameters exists, and that it will be defined later.

Why Use Function Prototypes?

Function prototypes are used for several reasons:

1. Forward Declaration: Function prototypes allow you to declare a function before it's defined. This is useful when you want to use a function before its definition.

2. Compiler Checking: When you provide a function prototype, the compiler checks the function call against the prototype to ensure that the correct number and types of arguments are passed.

3. Function Overloading: Function prototypes enable function overloading, which allows multiple functions with the same name to be defined, as long as they have different parameter lists.

Syntax :

The syntax for a function prototype is:

return-type function-name(parameter-list);

Example :

Here's an example of a function prototype:

int add(int, int);

This prototype declares a function named add that takes two int parameters and returns an int value.

5. Arrays and Strings

THEORY EXERCISE:

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

Ans :

In C++, an array is a collection of elements of the same data type stored in contiguous memory locations.

Single-Dimensional Arrays :

A single-dimensional array, also known as a one-dimensional array, is a collection of elements of the same data type stored in a single row or column.

Syntax :

data-type array-name[array-size];

Example

int scores[5];

This declares an array named scores that can hold 5 integer values.

Multi-Dimensional Arrays :

A multi-dimensional array is an array of arrays. It can be thought of as a table with rows and columns.

Syntax :

data-type array-name[array-size1][array-size2]...;

Example :

int matrix[2][3];

This declares a 2x3 matrix, which is an array of 2 rows and 3 columns.

Key differences :

Here are the key differences between single-dimensional and multi-dimensional arrays:

- Number of indices: Single-dimensional arrays have one index, while multi-dimensional arrays have multiple indices.

- Memory layout: Single-dimensional arrays are stored in contiguous memory locations, while multi-dimensional arrays are stored in a row-major or column-major layout.

- Accessing elements: Single-dimensional arrays are accessed using a single index, while multi-dimensional arrays are accessed using multiple indices.

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

Ans :

C++ provides a powerful way to work with strings using the `std::string` class, which is part of the standard library. Here's a breakdown of string handling in C++ with examples:

1. Creating and Initializing Strings :

* Direct Initialization:

```
std::string myString = "Hello, world!";
```

* Empty String:

```
std::string emptyString; // Creates an empty string
```

* Initialization from a Character Array:

```
char charArray[] = "C++ is fun";
std::string str(charArray);
```

2. Accessing String Characters :

* Operator[]: You can access individual characters using the square bracket operator.

```
std::string greeting = "Hello";
char firstChar = greeting[0]; // 'H'
```

* at() Method: This method is similar to the `[]` operator, but it provides bounds checking for safety.

```
std::string name = "Alice";
char secondChar = name.at(1); // 'l'
```

3. Concatenation and Assignment :

* + Operator: You can concatenate (join) strings using the `+` operator.

```
std::string firstName = "John";
std::string lastName = "Doe";
```

```
std::string fullName = firstName + " " + lastName; // "John Doe"
```

* += Operator: This operator concatenates and assigns the result back to the original string.

```
std::string message = "Welcome";
message += " to C++!";
```

* Assignment Operator (=): Assigns a new value to a string.

```
std::string str1 = "Hello";
std::string str2 = str1; // str2 now contains "Hello"
```

4. String Length and Size :

* length() Method: Returns the number of characters in the string.

```
std::string word = "Python";
int length = word.length(); // length = 6
```

* size() Method:

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

Ans:

In C++, arrays can be initialized in several ways. Here are some examples:

1D Array Initialization :

Here are some examples of initializing 1D arrays:

1. Initialization with a fixed size and values :

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

2. Initialization with a fixed size and default values :

```
int arr[5] = {};  // all elements initialized to 0
```

3. Initialization without specifying size

```
int arr[] = {10, 20, 30, 40, 50};  // size is determined by the number of elements
```

2D Array Initialization :

Here are some examples of initializing 2D arrays:

1. Initialization with a fixed size and values :

```
int arr[2][3] = {{10, 20, 30}, {40, 50, 60}};
```

2. Initialization with a fixed size and default values

int arr[2][3] = {}; // all elements initialized to 0

3. Initialization without specifying size (only for the outer array) :

int arr[][3] = {{10, 20, 30}, {40, 50, 60}}; // size of the outer array is determined by the number of inner arrays

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

Ans:

In C++, strings are a sequence of characters enclosed in double quotes. String operations and functions are used to manipulate and process strings.

String Operations :

1. Concatenation

Concatenation is the process of joining two or more strings together.

Example :

string str1 = "Hello";

string str2 = "World";

string result = str1 + " " + str2;

cout << result << std::endl; // Output: "Hello World"

2. Comparison

String comparison involves checking whether two strings are equal or not.

Example :

```
string str1 = "Hello";
string str2 = "Hello";
if (str1 == str2) {
   cout << "Strings are equal." << std::endl;
} else {
   cout << "Strings are not equal." << std::endl;
}
```

String Functions :

1. length()

The length() function returns the length of a string.

Example :

```
string str = "Hello";
int length = str.length();
cout << "Length: " << length << std::endl;  // Output: "Length: 5"
```

2. find()

The find() function finds the first occurrence of a substring within a string.

Example :

```
string str = "Hello World";
string substr = "World";
```

```
size_t pos = str.find(substr);
if (pos != std::string::npos) {
   cout << "Substring found at position " << pos << std::endl;
} else {
   cout << "Substring not found." << std::endl;
}
```

3. substr()

The substr() function extracts a substring from a string.

Example :

```
string str = "Hello World";
string substr = str.substr(6, 5);  // Extract 5 characters starting from position 6
cout << "Substring: " << substr << std::endl;  // Output: "World"
```

4. replace()

The replace() function replaces a substring with another substring.

Example :

```
string str = "Hello World";
string old_substr = "World";
string new_substr = "Universe";
str.replace(str.find(old_substr), old_substr.length(), new_substr);
cout << "Modified string: " << str << std::endl;  // Output: "Hello Universe"
```

6. Introduction to Object-Oriented Programming :

THEORY EXERCISE:

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

Ans :

the key concepts of OOP:

1. Classes and Objects :

- Class: A blueprint or template that defines the properties and behavior of an object.

- Object: An instance of a class, which has its own set of attributes (data) and methods (functions).

2. Inheritance :

- Inheritance: The process by which one class can inherit the properties and behavior of another class.

- Parent class (or superclass): The class being inherited from.

- Child class (or subclass): The class that inherits from the parent class.

3. Polymorphism :

- Polymorphism: The ability of an object to take on multiple forms, depending on the context in which it is used.

- Method overriding: When a child class provides a different implementation of a method that is already defined in its parent class.

- Method overloading: When multiple methods with the same name can be defined, but with different parameter lists.

4. Encapsulation :

- Encapsulation: The concept of bundling data and methods that operate on that data within a single unit (i.e., a class).

- Data hiding: The practice of hiding the implementation details of an object from the outside world, and only exposing the necessary information through public methods.

5. Abstraction :

- Abstraction: The practice of showing only the necessary information to the outside world while hiding the implementation details.

- Abstract class: A class that cannot be instantiated on its own and is meant to be inherited by other classes.

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

Ans :

a class is a blueprint or template that defines the properties and behavior of an object. An object, on the other hand, is an instance of a class, which has its own set of attributes (data) and methods (functions).

Class Definition :

A class definition typically includes:

- Data members (attributes): Variables that are part of the class.

- Member functions (methods): Functions that are part of the class and operate on the data members.

Object Creation :

To create an object, you need to instantiate a class using the new keyword or by declaring a variable of the class type.

Example :

Here's an example of a simple Person class:

```
#include <iostream>
#include <string>

// Class definition
class Person {
public:
    // Constructor
    Person(std::string name, int age) {
        this->name = name;
        this->age = age;
    }

    // Member functions
    void displayInfo() {
        cout << "Name: " << name << std::endl;
        cout << "Age: " << age << std::endl;
    }
```

```
    // Getter and setter for name
    string getName() { return name; }
    void setName(string name) { this->name = name; }

    // Getter and setter for age
    int getAge() { return age; }
    void setAge(int age) { this->age = age; }

private:
    // Data members
    string name;
    int age;
};

int main() {
    // Create an object
    Person person("John Doe", 30);

    // Call member functions
    person.displayInfo();

    // Use getter and setter
    person.setName("Jane Doe");
    person.setAge(31);
    cout << "Name: " << person.getName() << endl;
```

```
    cout << "Age: " << person.getAge() << endl;
}
```

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

Ans :

Inheritance in C++ is a mechanism that allows you to create new classes (derived classes) based on existing classes (base classes). The derived class inherits the properties and methods of the base class, allowing you to reuse code and create a hierarchical relationship between classes.

Here's an example:

```
#include <iostream>

// Base class: Vehicle
class Vehicle {
public:
    std::string brand;
    int year;

    void setBrand(std::string newBrand) {
        brand = newBrand;
    }

    void setYear(int newYear) {
        year = newYear;
    }
```

```
    void displayInfo() {
        std::cout << "Brand: " << brand << std::endl;
        std::cout << "Year: " << year << std::endl;
    }
};

// Derived class: Car (inherits from Vehicle)
class Car : public Vehicle {
public:
    std::string model;

    void setModel(std::string newModel) {
        model = newModel;
    }

    void displayCarInfo() {
        std::cout << "Model: " << model << std::endl;
        displayInfo(); // Call the displayInfo() method from the base class
    }
};

int main() {
    // Creating a Car object
    Car myCar;
```

```
    // Setting values for the Car object
    myCar.setBrand("Toyota");
    myCar.setYear(2023);
    myCar.setModel("Camry");

    // Displaying information about the Car
    myCar.displayCarInfo();

    return 0;
}
```

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

Ans :

Encapsulation in C++ is a fundamental object-oriented programming principle that combines data (member variables) and the functions (member functions) that operate on that data into a single unit called a class. It restricts direct access to the data from outside the class, protecting the data's integrity and ensuring it's modified only through controlled methods.

Here's how encapsulation is achieved in classes:

1. Data Hiding: Declare member variables as `private`. This prevents direct access to them from outside the class.

2. Accessor Methods: Provide public member functions (often called "getters" and "setters") to access and modify the private data. These methods act as controlled gateways to the data.

Access Specifiers

There are three access specifiers in C++:

1. public: Members declared as public are accessible from anywhere.

2. protected: Members declared as protected are accessible within the class and its derived classes.

3. private: Members declared as private are accessible only within the class.

Example :

```
#include <iostream>
#include <string>
Using namespace std ;
class Employee {
private:
   string name;
   int age;
   double salary;

public:
   // Constructor
   Employee(string name, int age, double salary) {
      this->name = name;
```

```
        this->age = age;
        this->salary = salary;
    }

    // Getter methods
    string getName() { return name; }
    int getAge() { return age; }
    double getSalary() { return salary; }

    // Setter methods
    void setName(string name) { this->name = name; }
    void setAge(int age) { this->age = age; }
    void setSalary(double salary) { this->salary = salary; }
};

int main() {
    Employee employee("John Doe", 30, 50000.0);

    // Accessing employee data using getter methods
    cout << "Name: " << employee.getName() << endl;
    cout << "Age: " << employee.getAge() << endl;
    cout << "Salary: " << employee.getSalary() << endl;

    // Modifying employee data using setter methods
    employee.setSalary(55000.0);
```

```
    cout << "Updated Salary: " << employee.getSalary() << endl;
}
```