# 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)?

Introduction:
Procedural Programming vs. Object-Oriented Programming (OOP):

1. Paradigm Type:
   - Procedural Programming: Follows a linear, top-down approach.
   - Object-Oriented Programming: Uses objects that encapsulate data and behavior.

2. Data Handling:
   - Procedural Programming: Focuses on functions that operate on raw data.
   - Object-Oriented Programming: Encourages the bundling of data and methods within objects.

3. Encapsulation:
   - Procedural Programming: Lacks strong encapsulation mechanisms.
   - Object-Oriented Programming: Emphasizes data hiding and encapsulation to restrict access to data.

4. Inheritance:
   - Procedural Programming: Does not support inheritance.

- Object-Oriented Programming: Supports the concept of inheritance for code reuse and structuring.

5. Polymorphism:
   - Procedural Programming: Does not inherently support polymorphism.
   - Object-Oriented Programming: Enables polymorphism, allowing objects of different classes to respond to the same message.

6. Code Reusability:
   - Procedural Programming: Relies on functions for code reuse.
   - Object-Oriented Programming: Promotes code reuse through inheritance and composition.

7. Modularity:
   - Procedural Programming: Lacks inherent modularity features.
   - Object-Oriented Programming: Encourages modular design through classes and objects.

Procedural Programming focuses on procedures or routines that operate on data, while Object-Oriented Programming revolves around objects that contain data and methods.

OOP offers benefits like encapsulation, inheritance, and polymorphism, which enhance code organization and reusability compared to Procedural Programming.

Developers often choose between these paradigms based on factors like project requirements, scalability, and maintainability.

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

Object Oriented Programming (OOP) offers several advantages over Procedural Programming (POP):

1. Modularity:
OOP allows code to be organized into modular units called objects, making it easier to understand, maintain, and reuse code.

2. Encapsulation:
OOP hides the internal workings of an object from the outside world, allowing for better control over data access and manipulation.

3. Inheritance:
OOP supports inheritance, a mechanism that enables the creation of new classes based on existing classes, leading to code reuse and a more hierarchical organization of code.

4. Polymorphism:
OOP allows objects of different classes to be treated as objects of a common superclass, improving code flexibility and extensibility.

5. Abstraction:
OOP provides abstraction by allowing developers to focus on the essential features of an object while hiding the implementation details, leading to a clearer separation of concerns.

6. Encapsulation and Data Hiding:
OOP promotes data hiding, protecting data within an object and exposing only the necessary methods to interact with that data,

enhancing security and reducing system complexity.

7. Extensibility:
OOP allows for the easy addition of new features through the creation of new classes or by modifying existing ones, promoting scalability and adaptability.

8. Code Reusability:
OOP emphasizes the creation of reusable components through classes and objects, reducing redundancy and improving development efficiency.

The main advantages of OOP over POP lie in its ability to promote modularity, encapsulation, inheritance, polymorphism, abstraction, data hiding, extensibility, and code reusability, all of which contribute to more efficient, maintainable, and flexible software development.

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

Step 1: Choose a C++ Compiler
Select a C++ compiler based on your operating system. Popular choices include GCC for Linux, Clang for macOS, and MinGW for Windows.

Step 2: Install a Text Editor or Integrated Development Environment (IDE)
Choose a text editor like Visual Studio Code or an IDE like Code::Blocks or CLion for a more feature-rich coding experience.

Step 3: Install Build Tools
Install build tools like CMake or Make to manage the build process of your C++ projects efficiently.

Step 4: Set Up Compiler Paths
Make sure the compiler paths are properly set in your system environment variables to enable the IDE or text editor to locate the compiler during the build process.

Step 5: Create a Project
Set up a new C++ project in your IDE or text editor, specifying the project name, location, and type.

Step 6: Write and Compile Your Code
Start writing your C++ code in the editor and compile it using the build tools configured in your environment.

Step 7: Debug and Test Your Code
Utilize the debugging features of your IDE or text editor to identify and fix any errors in your code. Test your program thoroughly to ensure it functions as expected.

Step 8: Build and Run Your Application
Once your code is error-free, build the project and run the compiled executable to test its functionality.

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

In C++, the main input/output operations are facilitated through the iostream library. The primary input operations include reading data from the standard input stream (cin) and other input sources like files. Commonly used input functions are ">>" for extracting data from the input stream and getline() for reading entire lines of text.

Example of reading input from the user using cin:

```
//cpp
int num;
cout << "Enter a number: ";
cin >> num;
```

---

Example of reading input from a file:

```
//cpp
ifstream inputFile("data.txt");
string line;
while (getline(inputFile, line)) {
    cout << line << endl;
}
inputFile.close();
```

—----------

On the other hand, output operations involve displaying data to the standard output stream (cout) or saving data to files. The primary output operation includes using the insertion operator "<<" to output data.

Example of outputting data to the console using cout:

//cpp

int result = 42;

cout << "The answer is: " << result << endl;

—----------

Example of outputting data to a file:

//cpp

ofstream outputFile("output.txt");

outputFile << "Hello from C++!";

outputFile.close();

These input/output operations in C++ are crucial for interacting with users, reading external data, and saving program outputs. Understanding how to effectively utilize these operations is fundamental for developing versatile and interactive C++ applications.

## 2. Variables, Data Types, and Operators

## [ THEORY EXERCISE ]

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

In C++, there are several data types categorized into different groups: fundamental, derived, user-defined, and abstract data types.

Fundamental data types include integers (int, short, long, and long long), floating-point numbers (float and double), characters (char), and boolean (bool).

Derived data types consist of arrays, pointers, references, and functions.

User-defined data types are structures and classes, while abstract data types are defined by programmers based on their requirements.

some examples:

1. Fundamental Data Types:
   - int myInteger = 10;
   - float myFloat = 3.14;
   - char myChar = 'A';
   - bool myBoolean = true;

2. Derived Data Types:
   - int myArray[5] = {1, 2, 3, 4, 5};
   - int* myPointer = &myInteger;

- int& myReference = myInteger;

3. User-Defined Data Types:
   - struct Person {
     string name;
     int age;
     };
   - class Rectangle {
     int width, height;
     };

4. Abstract Data Types:
   - Queue: A data structure that follows the First In First Out (FIFO) principle.
   - Stack: A data structure that follows the Last In First Out (LIFO) principle.

Understanding the different data types available in C++ is crucial for effectively managing and manipulating data in your programs.

By utilizing appropriate data types, programmers can optimize memory usage, improve code readability, and enhance the overall functionality of their software solutions.

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

In C++, type conversion plays a crucial role in manipulating data of different types. There are two main types of type conversion: implicit

and explicit.

Implicit type conversion, also known as automatic type conversion or coercion, occurs when the compiler automatically converts one data type to another without requiring any explicit instructions from the programmer. This conversion is done to facilitate operations involving mismatched data types. For example, when performing arithmetic operations between an integer and a floating-point number, the integer is implicitly converted to a floating-point number to match the data types.

On the other hand, explicit type conversion, also known as type casting, is performed explicitly by the programmer using casting operators. This type of conversion requires a specific conversion directive to inform the compiler to change the data type of a variable. Explicit type conversion provides the programmer with more control over the conversion process and helps prevent unintended data loss or errors that may occur during implicit conversions.

It is important to note that while implicit type conversion can simplify code and make it more concise, it can also lead to unexpected behavior if not used carefully. Explicit type conversion, though more verbose, allows for clearer intent and can help in catching potential errors at compile time.

Implicit type conversion is automatic and performed by the compiler, while explicit type conversion is manual and requires the programmer's intervention. Understanding the differences between these types of type conversion is essential for writing safe and efficient C++ code.

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

In C++, operators are symbols that represent specific operations to be performed on operands.

1. Arithmetic Operators:
Examples:

- Addition (+): Adds two operands together.
- Subtraction (-): Subtracts the second operand from the first.
- Multiplication (*): Multiplies two operands.
- Division (/): Divides the first operand by the second.
- Modulus (%): Returns the remainder of the division of the first operand by the second.

2. Relational Operators:
Examples:

- Equal to (==): Checks if two operands are equal.
- Not equal to (!=): Checks if two operands are not equal.
- Greater than (>): Checks if the first operand is greater than the second.
- Less than (<): Checks if the first operand is less than the second.
- Greater than or equal to (>=): Checks if the first operand is greater than or equal to the second.
- Less than or equal to (<=): Checks if the first operand is less than or equal to the second.

3. Logical Operators:
Examples:

- Logical AND (&&): Returns true if both operands are true.
- Logical OR (||): Returns true if either operand is true.
- Logical NOT (!): Returns the negation of the operand's logical

state.

4. Bitwise Operators:
Examples:

- Bitwise AND (&): Performs a bitwise AND operation between two operands.
- Bitwise OR (|): Performs a bitwise OR operation between two operands.
- Bitwise XOR (^): Performs a bitwise XOR operation between two operands.
- Bitwise NOT (~): Flips the bits of the operand.

5. Assignment Operators:
Examples:

- Assign (=): Assigns the value of the right operand to the left operand.
- Add AND (+=): Adds the right operand to the left operand and assigns the result to the left operand.
- Subtract AND (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.
- Multiply AND (*=): Multiplies the left operand by the right operand and assigns the result to the left operand.

6. Increment and Decrement Operators:
Examples:

- Increment (++): Increases the value of the operand by 1.
- Decrement (--): Decreases the value of the operand by 1.

Understanding and mastering these operators is fundamental for efficient programming in C++. By applying these operators effectively, developers can manipulate data and control the flow of their programs

with precision and efficiency.

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

Constants and literals play a crucial role in C++ programming by providing a way to store and represent fixed values that do not change during the execution of a program. Constants are identifiers that hold values that cannot be modified once they are assigned, while literals are actual fixed values used in code.

The primary purpose of using constants is to improve the readability, maintainability, and efficiency of code. By assigning a meaningful name to a constant value, the code becomes more self-explanatory and easier to understand for other programmers. Additionally, constants help avoid magic numbers in code, which are hardcoded values that make code less flexible and harder to update.

On the other hand, literals are used to represent fixed values directly within code. They can be of different types, such as integer literals, floating-point literals, character literals, string literals, and boolean literals. Literals provide a convenient way to input values directly into code without the need for defining separate variables.

In C++, constants are typically declared using the 'const' keyword, which tells the compiler that the value should not be changed. Constants can also be defined using preprocessor directives like #define or by using 'constexpr' for compile-time evaluated constants.

Overall, constants and literals in C++ help in writing more structured,

readable, and maintainable code by encapsulating fixed values and making the code more expressive. By leveraging constants and literals effectively, programmers can enhance the robustness and efficiency of their C++ programs.

## 3. Control Flow Statements

## [ THEORY EXERCISE ]

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

Conditional statements in C++ allow for decision-making within a program based on specified conditions. Two common types of conditional statements in C++ are the if-else statement and the switch statement.

The if-else statement is used to execute a block Aof code if a specified condition is true, and another block of code if the condition is false. It has the following structure:

—----------

```
if (condition) {
    // code to be executed if the condition is true
} else {
```

```
    // code to be executed if the condition is false
}
```

---

The switch statement allows a variable to be tested for equality against a list of values and execute different blocks of code based on which value matches the variable. It has the following structure:

---

```
switch (expression) {
    case value1:
        // code to be executed if expression equals value1
        break;
    case value2:
        // code to be executed if expression equals value2
        break;
    .
    .
    .
    default:
        // code to be executed if expression does not match any value
}
```

---

When using the switch statement, it is important to include a `break` statement after each case to prevent fall-through behavior where

multiple case blocks are executed. The `default` case is optional and executes if none of the other cases match the expression.

The if-else statement is more flexible and can accommodate complex conditional logic, while the switch statement is more suitable when comparing a single expression against multiple values. Both statements play a crucial role in controlling the flow of a program based on specific conditions, enhancing its flexibility and functionality.

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

In C++, for, while, and do-while loops are essential control flow structures used to repeatedly execute a block of code until a specified condition is met. The key differences lie in their syntax, initialization, condition evaluation, and execution characteristics.

1. For Loop:

- Initialization, condition, and increment statements are set up within the loop structure.
- Ideal for iterating over a range with a fixed number of iterations.
- Structure: for(initialization; condition; increment) { // code block}

2. While Loop:

- Condition is evaluated before executing the code block, so it may not execute at all if the condition is initially false.
- Suitable when the number of iterations is uncertain.
- Structure: while(condition) { // code block}

3. Do-While Loop:

- Similar to the while loop, but the code block is executed at least once before checking the condition for further iterations.
- More appropriate when the loop body must execute at least once.
- Structure: do { // code block} while(condition);

for loops are best for a fixed number of iterations, while loops are useful for indeterminate loops based on a condition, and do-while loops ensure at least one execution before checking the condition.

Understanding the nuances of each loop type allows programmers to effectively control the flow of their code and optimize performance based on different iteration requirements.

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

Break and continue statements are used in loops to control the flow of the loop iteration.

The "break" statement is used to immediately exit a loop when a specific condition is met. It helps in terminating the loop prematurely and is commonly used when a certain condition is fulfilled. Here is an example in Python:

—----------

```
for i in range(1, 11):
    if i == 5:
```

```
        break
    print(i)
```

—----------

In this example, the loop will stop when the value of "i" becomes 5, and the output will be: 1, 2, 3, 4.

On the other hand, the "continue" statement is used to skip the current iteration and move to the next one in the loop. It allows you to bypass certain iterations based on specific conditions. Here is an example in Java:

—----------

```
for (int i = 1; i <= 5; i++) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println(i);
}
```

—----------

In this example, the loop will skip even numbers and only print odd numbers from 1 to 5.

Both "break" and "continue" statements are essential tools in loop control flow, allowing for more precise control over how the loop iterates and processes data. By strategically using these statements, developers can tailor the behavior of loops to suit their specific needs and optimize the efficiency of their code.

## 4. Explain nested control structures with an example.

Nested control structures refer to the concept of having one control structure nested within another. This allows for complex decision-making and looping patterns in programming. An example of nested control structures is having an if statement nested within a while loop.

Consider the following pseudocode example:

—----------

```
while x < 10
    if x is even
        output "x is even"
    else
        output "x is odd"
    end if
    x = x + 1
end while
```

—----------

In this example, we have a while loop that continues to run as long as the value of x is less than 10. Within the while loop, we have an if statement that checks if the value of x is even or odd, and outputs the corresponding message. This demonstrates how nested control structures can be used to create more complex and intricate logic in programming.

By nesting control structures, programmers can create more sophisticated algorithms and decision-making processes, allowing for greater flexibility and efficiency in coding. Understanding how to effectively utilize nested control structures is essential for mastering programming concepts and developing robust and elegant code.

## 4. Functions and Scope

## [ THEORY EXERCISE ]

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

In C++, a function is a block of code that performs a specific task. It is a self-contained unit of code that can be reused throughout a program. Functions help in organizing code, making it more readable, maintainable, and efficient.

Function Declaration:

A function declaration is also known as a function prototype. It specifies the function's name, return type, and parameters without providing the actual implementation details. This allows the compiler to know the function's existence and how it should be called before the function is defined. Function declarations are typically placed at the beginning of the program or in header files.

Example of function declaration:

```
//cpp

int add(int a, int b);
```

—----------

Function Definition:

A function definition provides the actual implementation of the function. It includes the function header (return type, name, parameters) followed by the function body (code that defines the task performed by the function). Functions can be defined before or after they are called within the program.

Example of function definition:

```
//cpp

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

—----------

Function Calling:

Function calling, also known as invoking a function, is the process of using a function to perform a specific task. To call a function, its name must be followed by parentheses containing any necessary arguments. The function is executed when it is called, and the result (if any) is returned to the calling code.

Example of function calling:

```
//cpp

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

—----------

Functions in C++ are essential building blocks that enable modular programming and code reusability. The separation of function declaration, definition, and calling allows for a well-structured and organized program. By understanding these concepts, programmers can create efficient and scalable code that is easier to debug and maintain.

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

Variables in C++ can have either local or global scope. Local variables are declared within a specific block of code, such as within a function, and they are only accessible within that block. Local variables are temporary and are typically used for storing data that is only needed for a short period of time.

On the other hand, global variables are declared outside of any function and are accessible throughout the entire program. Global variables can be accessed by any part of the program, making them useful for storing data that needs to be shared between different functions or modules. However, global variables should be used sparingly as they can lead to issues such as naming conflicts and difficulties in debugging.

One key difference between local and global variables is their lifetime. Local variables exist only within the block in which they are declared and are destroyed when the block is exited. Global variables, on the other hand, exist for the entire duration of the program's execution.

Another important concept related to variable scope in C++ is the concept of scope resolution. When a variable with the same name exists in both

local and global scopes, the local variable takes precedence within its scope. To access the global variable in such cases, the scope resolution operator (::) can be used to explicitly specify the scope from which the variable should be accessed.

understanding the scope of variables in C++ is crucial for writing efficient and maintainable code. By distinguishing between local and global scope, programmers can effectively manage data within their programs and avoid conflicts that can lead to errors. By utilizing proper scope resolution techniques, programmers can ensure that variables are accessed correctly and that their programs run smoothly.

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

Recursion in C++ is a powerful programming technique where a function calls itself directly or indirectly to solve a problem. The key concept behind recursion is breaking down a complex problem into simpler subproblems that can be solved iteratively. This allows for elegant and concise solutions to problems that exhibit recursive properties.

Here is an example of recursion in C++ to calculate the factorial of a given number:

```
//cpp
#include <iostream>

int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
```

```
    } else {
        return n * factorial(n - 1);
    }
}

int main() {
    int num = 5;
    int result = factorial(num);
    std::cout << "Factorial of " << num << " is: " << result << std::endl;

    return 0;

}—----------------------------------------------------------------
```

In this example, the `factorial` function calculates the factorial of a number by recursively calling itself with a smaller input until the base case is reached (n = 0 or n = 1), at which point the recursion stops. This demonstrates how recursion can simplify the solution to a complex problem by breaking it down into smaller, manageable subproblems.

recursion in C++ is a powerful tool that leverages the concept of functions calling themselves to solve problems more effectively and elegantly. By understanding its key principles and applying it appropriately, programmers can write efficient and concise code to tackle a wide range of computational challenges.

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

In C++, function prototypes are declarations that specify the function's signature (name, parameters, and return type) without providing the actual function implementation. They act as a blueprint for the compiler, informing it about the existence and structure of the function before it is defined in the program.

Function prototypes are crucial for several reasons:

1. Early Declaration: By using function prototypes at the beginning of a program or file, developers can declare the functions they intend to use throughout the codebase, allowing them to define functions in any order they prefer.

2. Cross-File Communication: Function prototypes enable communication between different source files by providing a way for functions defined in one file to be called from another. This is essential for building modular and maintainable codebases.

3. Type and Parameter Checking: Function prototypes enforce type checking, ensuring that functions are called with the correct number and type of arguments. This helps catch errors at compile time rather than runtime, leading to more robust and bug-free code.

4. Improving Code Readability: Function prototypes act as documentation for the functions in a program, making it easier for other developers (and the programmer themselves) to understand the function interface without needing to inspect the function definition.

By using function prototypes in C++, developers promote code organization, enhance maintainability, and improve overall code quality by catching errors early in the development process. Incorporating function prototypes is a best practice in C++ programming and contributes to the creation of more efficient and structured software.

## 5. Arrays and Strings

## [ THEORY EXERCISE ]

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

Arrays in C++ are a fundamental data structure that allows storing elements of the same data type in contiguous memory locations. They provide a way to efficiently access and manipulate multiple elements using a single identifier.

Single-dimensional arrays in C++ are used to store elements in a linear sequence, accessed using a single index. They offer a simple way to manage a collection of data, with each element accessible by its position within the array.

On the other hand, multi-dimensional arrays in C++ extend this concept by storing elements in a grid-like structure with multiple dimensions. This enables organizing data in rows and columns, or even higher dimensions, such as a matrix. Accessing elements in a multi-dimensional array requires specifying multiple indices corresponding to each dimension.

The key difference between single-dimensional and multi-dimensional arrays lies in their structure and the way data is accessed. Single-dimensional arrays provide a linear sequence of elements accessed by a single index, while multi-dimensional arrays organize data in a grid-like format with multiple indices to access specific elements.

When working with arrays in C++, it is important to consider the memory layout, indexing, and access mechanisms to efficiently manipulate data. Understanding the differences between single-dimensional and multi-dimensional arrays is essential for effectively implementing algorithms and managing complex datasets in C++ programs.

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

String handling in C++ involves manipulating sequences of characters. The standard way to work with strings in C++ is by using the `std::string` class from the Standard Template Library (STL).

Here are some key concepts and examples of string handling in C++:

1. String Initialization:

//cpp

#include <iostream>

#include <string>

int main() {

    std::string str1 = "Hello,";

    std::string str2(" World!");

```
    std::string str3;

    str3 = str1 + str2;

    std::cout << str3 << std::endl; // Output: Hello, World!

    return 0;
}
```

—----------

2. String Concatenation:

```
//cpp
#include <iostream>
#include <string>

int main() {
    std::string str1 = "Hello,";
    std::string str2 = " World!";

    str1 += str2;

    std::cout << str1 << std::endl; // Output: Hello, World!

    return 0;
```

```
}
```

---

3. String Comparison:

```
//cpp
#include <iostream>
#include <string>

int main() {
    std::string str1 = "apple";
    std::string str2 = "orange";

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

    return 0;
}
```

---

4. String Length:

```
//cpp
```

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

int main() {
    std::string str = "Hello, World!";

     std::cout << "Length of string: " << str.length() << std::endl; // Output: Length of string: 13

    return 0;
}
```

---

5. String Substring:

```
//cpp
#include <iostream>
#include <string>

int main() {
    std::string str = "Hello, World!";

    std::string sub = str.substr(7, 5); // Extract "World"

    std::cout << sub << std::endl; // Output: World
```

```
    return 0;
}
```

---

you can effectively handle strings in C++ using the `std::string` class. Experimenting with these concepts will provide a solid foundation for manipulating strings in your C++ programs.

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

In C++, arrays are initialized by specifying the data type of the elements and the size of the array.

For 1D arrays:

```
//cpp
// Initializing a 1D array of integers with size 5
int arr1D[5] = {1, 2, 3, 4, 5};

// Initializing a 1D array of strings with size 3
string strArr1D[3] = {"apple", "banana", "orange"};
```

---

For 2D arrays:

```
//cpp
```

```
// Initializing a 2D array of integers with size 3x3
int arr2D[3][3] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Initializing a 2D array of characters with size 2x4
char charArr2D[2][4] = {
    {'a', 'b', 'c', 'd'},
    {'e', 'f', 'g', 'h'}
};
```

—----------

It is important to note that in C++, arrays are zero-indexed, meaning the first element of an array is at index 0. When initializing arrays, the size specified must be a compile-time constant known at the time of compilation. Additionally, arrays in C++ do not automatically have bounds checking, so care must be taken to ensure that array elements are accessed within their defined boundaries to avoid undefined behavior.

Overall, initializing arrays in C++ involves specifying the data type, the size, and the values of the elements, providing a versatile and powerful tool for handling collections of data in a structured manner.

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

In C++, string operations and functions play a crucial role in manipulating and working with strings efficiently. Strings in C++ are a sequence of characters stored in a specific order, making them a fundamental data type in programming. There are various built-in functions and operators specifically designed to facilitate string handling in C++.

Key String Operations in C++:

1. Concatenation: The concatenation operator (+) allows merging two strings into one.

2. Comparison: Strings can be compared using relational operators such as ==, !=, >, <, >=, <=.

3. Substring: The substr() function is used to extract a portion of a string based on the specified position and length.

4. Length: The length() function returns the number of characters in a string.

5. Accessing Characters: Individual characters in a string can be accessed using array notation or the at() function.

6. Finding Characters: The find() function locates a specific character or substring within a string.

7. Insertion and Deletion: The insert() function is used to insert characters into a string, while the erase() function removes characters.

Key String Functions in C++:

1. stoi() and to_string(): Used to convert strings to integers and vice versa.

2. getline(): Reads an entire line from input, including whitespaces.

3. find_first_of() and find_last_of(): Locate the first or last occurrence of a set of characters in a string.

4. replace(): Replaces a portion of a string with another string.

5. c_str(): Returns a pointer to a null-terminated character array representing the string.

6. swap(): Exchanges the content of two strings.

It is essential to understand and leverage these string operations and functions in C++ to effectively work with strings in programming tasks. By mastering these concepts, developers can efficiently handle and manipulate strings to build robust and functional applications.

## 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 centered around the concept of "objects." The key concepts of OOP include:

1. Classes and Objects: Classes are blueprints for creating objects, defining their properties (attributes) and behaviors (methods). Objects are instances of classes, representing real-world entities.

2. Encapsulation: Encapsulation refers to the bundling of data (attributes) and methods (behaviors) within a class. It helps in hiding the internal implementation details of an object and only revealing necessary information.

3. Inheritance: Inheritance allows a class (subclass) to inherit properties and behaviors from another class (superclass). This promotes code reusability and hierarchy among classes.

4. Polymorphism: This concept allows objects to be treated as instances of their superclass, enabling different classes to be used interchangeably, leading to flexibility and extensibility in code.

5. Abstraction: Abstraction involves hiding complex implementation details and providing a simplified interface for interacting with objects. It focuses on what an object does rather than how it does it.

6. Modularity: OOP promotes the decomposition of a system into smaller, manageable modules (classes), thus improving code organization, maintainability, and reusability.

Object-Oriented Programming enhances code quality, promotes scalability, and fosters a more intuitive way of modeling real-world entities in software development. It remains a fundamental paradigm widely used in modern programming languages such as Java, C++, and Python.

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

In C++, classes are user-defined data types that encapsulate data and behavior. Objects are instances of these classes, representing real-world entities with attributes (data members) and actions (member functions). Classes serve as blueprints, defining the structure and behavior of objects, while objects hold data specific to their class and can perform operations defined by the class.

Example:

```
//cpp

#include <iostream>


class Rectangle {

private:

    double length;

    double width;


public:

    Rectangle(double l, double w) : length(l), width(w) {}


    double calculateArea() {

        return length * width;

    }


    void setDimensions(double l, double w) {

        length = l;
```

```
        width = w;
    }
};

int main() {
    Rectangle myRectangle(5.0, 3.0);
      std::cout << "Area of rectangle: " << myRectangle.calculateArea() << std::endl;

    myRectangle.setDimensions(4.0, 2.0);
 std::cout << "Updated area of rectangle: " << myRectangle.calculateArea() << std::endl;
}
```

—----------

Objects such as `myRectangle` are instances of the `Rectangle` class and hold specific length and width values. The member function `calculateArea()` computes the area of the rectangle based on its dimensions, demonstrating the behavior associated with the class. Through objects, we can create multiple instances of the `Rectangle` class with varying dimensions, each independent of the others.

## 3. Explain string handling in C++ with examples.

In C++, inheritance is a fundamental object-oriented programming concept that allows a class to inherit properties and behaviors from another class. It enables the creation of a hierarchy where classes can be organized in a parent-child relationship, promoting code reusability and modularity.

Consider the following example:

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

// Base class
class Shape {
public:
    virtual double area() {
        cout << "Calculating area of Shape" << endl;
        return 0;
    }
};

// Derived class
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}
```

```
    double area() override {
        cout << "Calculating area of Circle" << endl;
        return 3.14159 * radius * radius;
    }
};

// Derived class
class Rectangle : public Shape {
private:
    double width, height;

public:
    Rectangle(double w, double h) : width(w), height(h) {}

    double area() override {
        cout << "Calculating area of Rectangle" << endl;
        return width * height;
    }
};

int main() {
    Shape* shape1 = new Circle(5);
    Shape* shape2 = new Rectangle(4, 6);
```

```
    cout << "Area of Circle: " << shape1->area() << endl;
    cout << "Area of Rectangle: " << shape2->area() << endl;

    delete shape1;
    delete shape2;

}
```

---

In this example, we have a base class `Shape` with a virtual function `area()`. We then define two derived classes `Circle` and `Rectangle` that inherit from the `Shape` class. Each derived class provides its own implementation of the `area()` function.

By utilizing inheritance, we are able to reuse the `area()` function implementation in the `Shape` class and extend it in the derived classes `Circle` and `Rectangle`, resulting in a structured and modular design. Through polymorphism and dynamic binding, we can invoke the appropriate `area()` function based on the object's actual type at runtime, demonstrating the power and flexibility of inheritance in C++.

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

Encapsulation in C++ is the fundamental concept of bundling data (attributes) and methods (functions) that operate on that data within a single unit, called a class. It allows for data hiding and only exposing the necessary interfaces to interact with the class, promoting data security and modularity in code design.

Achieving encapsulation in C++ is done by defining a class with private, protected, and public access specifiers. Private members can only be accessed within the class itself, protecting the data from external manipulation. Protected members are accessible within the class and its derived classes, while public members are accessible from outside the class and are used as the interface through which external code interacts with the class.

By defining the internal data members as private and providing public methods to manipulate or access that data, encapsulation ensures that the internal state of the object remains consistent and is not directly tampered with. This information hiding principle allows for better control over access to data, reducing the risk of unintended modifications that could lead to bugs and errors.

In summary, encapsulation in C++ is achieved by utilizing access specifiers to control the visibility of class members, thereby encapsulating the data and methods within a class to enforce data protection, enhance code readability, and promote reusable and modular code design practices.