Introduction to oops using c++

Object-Oriented Programming (OOP) is a powerful programming paradigm that enhances the way we design, structure, and organize code. In C++, OOP is a fundamental concept, and it revolves around the concept of "objects." Let's delve into an introduction to OOP in C++:

Object-Oriented Programming in C++: An Introduction

C++ is a versatile and widely-used programming language known for its ability to blend both procedural and object-oriented programming paradigms. Object-Oriented Programming (OOP) is a fundamental approach in C++, where programs are constructed as a collection of objects, each representing a real-world entity or an abstract concept.

Key Concepts of OOP in C++:

Classes and Objects: In C++, a class is like a blueprint or a template that defines the structure and behavior of objects. Objects are instances of classes, and they encapsulate data (attributes) and functions (methods) that operate on that data. For example, a "Car" class can describe the properties and actions of different car objects.

Encapsulation: Encapsulation is the concept of bundling data (attributes) and the methods that operate on that data within a single unit, the class. This helps in hiding the internal details of how an object works and provides a well-defined interface to interact with it. Access control keywords like public, private, and protected determine the visibility of class members.

Inheritance: Inheritance allows you to create a new class (derived or child class) based on an existing class (base or parent class). The derived class inherits the properties and methods of the base class, enabling code reuse and specialization. It promotes the "is-a" relationship. For instance, a "SportsCar" class can inherit from the "Car" class.

Polymorphism: Polymorphism enables objects of different classes to be treated as objects of a common base class. It allows for method overriding, where derived classes can provide their own implementation of inherited methods. This promotes the flexibility and extensibility of your code. It is closely associated with the "virtual" keyword and dynamic binding.

Abstraction: Abstraction involves simplifying complex systems by modeling classes at a high level of generality. It focuses on essential features while hiding non-essential details. It helps developers manage the complexity of larger programs by providing a clear and concise view of the interactions between objects.

Composition: Composition is a way to build complex objects by combining simpler objects or components. It enables you to create more flexible and modular code by assembling objects to achieve a specific functionality.

Benefits of OOP in C++:

Modularity: OOP promotes code modularity, making it easier to understand, maintain, and extend.
Reusability: You can reuse existing classes and objects in new projects or extend them for specific requirements.
Encapsulation: Encapsulation enhances data security and integrity by restricting access to data members.
Inheritance: Inheritance supports code reuse and the creation of specialized classes.
Polymorphism: Polymorphism allows for flexible and dynamic code execution.
In summary, Object-Oriented Programming in C++ offers a structured and organized approach to software development, emphasizing the use of objects, classes, and their interactions to build robust, maintainable, and extensible applications. It is a powerful paradigm that has played a significant role in modern software engineering.

Data Types

In C++, data types are used to define the type of data that a variable can hold. C++ provides a variety of data types to accommodate different types of data, such as integers, floating-point numbers, characters, and more. Here are some common data types in C++ with examples:

Integer Data Types:

int: Represents integers.

int age = 25;
short: Represents short integers.

short distance = 100;
long: Represents long integers.

long population = 1000000L;
unsigned int: Represents non-negative integers.

unsigned int count = 10;
Floating-Point Data Types:

float: Represents single-precision floating-point numbers.

float pi = 3.14159f;
double: Represents double-precision floating-point numbers (more precision than float).

double price = 99.99;
long double: Represents extended-precision floating-point numbers (more precision than double).

long double high_precision = 1234.567890123456789L;
Character Data Type:

char: Represents a single character.

char grade = 'A';
Boolean Data Type:

bool: Represents boolean values (true or false).

bool isRaining = true;
Enumeration Data Type:

enum: Represents a set of named integer constants.

enum Color { Red, Green, Blue };
Color selectedColor = Green;
User-Defined Data Types:

struct: Defines a composite data type composed of multiple members.

```
struct Point {
   int x;
   int y;
};
Point p1 = {3, 4};
```

class: Defines a user-defined class type with data members and member functions.

```
class Student {
public:
   int id;
   std::string name;
};
Student s1;
s1.id = 101;
s1.name = "Alice";
```

Pointers:

int*: Represents a pointer to an integer.

int* ptr = nullptr;
char*: Represents a pointer to a character.

char* str = "Hello";
Arrays:

int[]: Represents an array of integers.

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

std::string: Represents a string of characters.

std::string greeting = "Hello, World!";
These are some of the common data types in C++. Remember that C++ allows for user-defined data types through classes and structs, which can be customized to suit your specific needs.

If else if else

In C++, the if, else if, and else statements are used for decision-making and conditional execution of code. These statements allow you to control the flow of your program based on specified conditions. Here are examples of how to use if, else if, and else statements in C++:

if Statement:

The if statement is used to execute a block of code if a condition is true.

```
int age = 25;

if (age >= 18) {
   std::cout << "You are an adult." << std::endl;
}
```

In this example, the code inside the if block will execute because age is greater than or equal to 18.

else if Statement:

The else if statement allows you to check multiple conditions sequentially.

```
int score = 75;

if (score >= 90) {
   std::cout << "You got an A." << std::endl;
} else if (score >= 80) {
   std::cout << "You got a B." << std::endl;
} else if (score >= 70) {
   std::cout << "You got a C." << std::endl;
} else {
   std::cout << "You need to improve." << std::endl;
}
```

In this example, the program checks the score against multiple conditions and prints the corresponding message based on the condition that evaluates to true. If none of the conditions are true, the else block is executed.

else Statement:

The else statement is used to specify a block of code to execute when the if condition (or any preceding else if conditions) is false.

```
int temperature = 28;

if (temperature >= 30) {
   std::cout << "It's hot outside." << std::endl;
} else {
   std::cout << "It's not very hot today." << std::endl;
}
```

In this example, if the temperature is less than 30, the code inside the else block will be executed.

Remember that you can nest if, else if, and else statements to handle more complex conditional logic. The order of conditions is important, as the program will execute the first block whose condition is true and then exit the entire if-else structure.

Loops

In C++, loops are used to execute a block of code repeatedly as long as a specified condition is true or for a fixed number of iterations. There are three main types of loops in C++: for, while, and do-while. Here are examples of each type of loop:

for Loop:

The for loop is typically used when you know the number of iterations in advance.

```
for (int i = 1; i <= 5; i++) {
   std::cout << "Iteration " << i << std::endl;
}
```

This for loop will execute the code block five times, printing "Iteration 1" through "Iteration 5" to the console.

while Loop:

The while loop is used when you want to repeat a block of code as long as a condition is true.

```
int count = 1;
while (count <= 5) {
   std::cout << "Count: " << count << std::endl;
   count++;
}
```

This while loop will keep executing as long as count is less than or equal to 5.

do-while Loop:

The do-while loop is similar to the while loop, but it guarantees that the loop body will execute at least once because the condition is checked after the loop body.

```
int x = 1;
do {
   std::cout << "Value of x: " << x << std::endl;
   x++;
} while (x <= 5);
```

This do-while loop will execute the code block once for x = 1, and then it will continue as long as x is less than or equal to 5.

Additionally, you can use control statements like break and continue to control the flow within loops:

break: It exits the loop prematurely when a certain condition is met.

```
for (int i = 1; i <= 10; i++) {
   if (i == 5) {
      break; // Exit the loop when i reaches 5
   }
   std::cout << "Value of i: " << i << std::endl;
}
```

continue: It skips the current iteration and continues with the next iteration of the loop.

```
for (int i = 1; i <= 5; i++) {
   if (i == 3) {
      continue; // Skip iteration when i is 3
   }
   std::cout << "Value of i: " << i << std::endl;
}
```

These examples demonstrate the basic usage of loops in C++. Depending on your specific programming tasks, you may choose the loop type that best fits your needs.

Functions

In C++, functions are blocks of code that can be defined and called to perform specific tasks. Functions allow you to modularize your code, making it more organized and easier to maintain. Here are examples of how to define and use functions in C++:

Function Declaration and Definition:

In C++, you need to declare a function before using it. Here's how you declare and define a simple function:

```
// Function declaration (prototype)
int add(int a, int b);

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

In this example, we declare a function add that takes two integers as parameters and returns their sum. The function is defined later in the code.

Function Call:

To use a function, you need to call it with appropriate arguments:

```
int result = add(5, 3);
std::cout << "Sum: " << result << std::endl;
```

This code calls the add function with the arguments 5 and 3 and assigns the result to the result variable.

Function with No Parameters and No Return Value:

You can define functions that don't take parameters or return values:

```
void greet() {
   std::cout << "Hello, World!" << std::endl;
}
```

To call this function, simply use its name:

greet(); // Call the greet function
Function with Default Arguments:

C++ allows you to specify default values for function parameters:

```
int multiply(int a, int b = 2) {
   return a * b;
}
```

If you don't provide a value for b, it will default to 2:

```
int result1 = multiply(3, 4); // result1 is 12
int result2 = multiply(5);    // result2 is 10 (uses the default value)
```

Function Overloading:

C++ supports function overloading, which allows you to define multiple functions with the same name but different parameter lists:

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

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

The appropriate function is called based on the argument types:

```
int intSum = add(5, 3);         // Calls the int version
double doubleSum = add(2.5, 1.5); // Calls the double version
```

Recursive Functions:

C++ also supports recursive functions, which call themselves. Here's an example of a recursive function to calculate factorial:

```
unsigned long long factorial(int n) {
   if (n == 0 || n == 1) {
      return 1;
   } else {
      return n * factorial(n - 1);
   }
}
```

To use this recursive function:

```
unsigned long long result = factorial(5); // result is 120
```

These examples demonstrate the basic concepts of functions in C++, including declaration, definition, calling, and different parameter scenarios. Functions are a fundamental building block of C++ programs, allowing you to write organized and reusable code.

Switch Case

In C++, the switch statement is used for decision-making based on the value of an expression. It provides an efficient way to choose among multiple alternatives. Here's an example of how to use the switch statement in C++:

```
#include <iostream>

int main() {
   int choice;

   std::cout << "Menu:\n";
   std::cout << "1. Option 1\n";
   std::cout << "2. Option 2\n";
   std::cout << "3. Option 3\n";
   std::cout << "Enter your choice: ";
   std::cin >> choice;

   switch (choice) {
      case 1:
         std::cout << "You selected Option 1." << std::endl;
         break;

      case 2:
         std::cout << "You selected Option 2." << std::endl;
         break;

      case 3:
         std::cout << "You selected Option 3." << std::endl;
         break;

      default:
         std::cout << "Invalid choice. Please select a valid option." << std::endl;
         break;
   }

   return 0;
}
```

In this example:

We declare an integer variable choice to store the user's input.

We display a menu to the user and prompt them to enter their choice.

The switch statement is used to evaluate the value of choice. Depending on the value of choice, different code blocks are executed. If none of the cases matches the value of choice, the default case is executed.

Each case represents a possible value for choice. When a case is matched, the code inside that case block is executed, and then the break statement is used to exit the switch statement. This prevents the subsequent cases from being executed.

The default case is executed when none of the case values matches the value of choice. It is used to handle invalid or unexpected inputs.

When you run this program, it will execute the code block associated with the user's choice or display an error message if the choice is invalid.

Here's a sample run of the program:

Menu:
1. Option 1
2. Option 2
3. Option 3
Enter your choice: 2
You selected Option 2.
The switch statement is a useful tool for handling multiple cases and making decisions based on the value of an expression efficiently.

Pointers

In C++, a pointer is a variable that stores the memory address of another variable. Pointers are a powerful feature of the C++ language and are commonly used for tasks like dynamic memory allocation, passing parameters by reference, and working with arrays and data structures. Here are examples of how to declare, initialize, and use pointers in C++:

Declaring and Initializing Pointers:

You declare a pointer using an asterisk * before the variable name. To initialize a pointer, you can assign it the address of an existing variable using the address-of operator & or allocate memory for it using new.

```
int x = 42;           // Declare and initialize an integer variable
int* ptr = &x;        // Declare and initialize a pointer to an integer
```

In this example, ptr is a pointer to an integer, and it is initialized with the address of the x variable.

Dereferencing Pointers:

To access the value pointed to by a pointer, you use the dereference operator *.

```
int y = *ptr;  // y is now equal to 42 (the value pointed to by ptr)
```

Dynamic Memory Allocation:

Pointers are often used to allocate memory dynamically using the new operator.

```
int* dynamicInt = new int; // Allocate memory for an integer
*dynamicInt = 10;         // Store a value in the dynamically allocated memory
delete dynamicInt;        // Release the allocated memory to prevent memory leaks
```

Dynamic memory allocation allows you to create objects with a lifetime that extends beyond the scope of their declaration.

Pointer Arithmetic:

Pointers can be used for pointer arithmetic, such as incrementing or decrementing the pointer to access adjacent memory locations.

```
int array[] = {10, 20, 30, 40};
int* pArray = array;    // Point to the start of the array

// Accessing array elements using pointer arithmetic
int secondElement = *(pArray + 1);  // Accesses the second element (20)
```

Pointer to Functions:

Pointers can also point to functions, allowing you to call functions dynamically.

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

int (*funcPtr)(int, int);  // Declare a pointer to a function
funcPtr = &add;            // Assign the address of the 'add' function

int result = funcPtr(3, 4); // Call the 'add' function through the pointer
```

Null Pointers:

Pointers can be set to null using the nullptr keyword, which indicates that they do not point to any valid memory address.

```
int* nullPtr = nullptr;
```

Passing Pointers as Function Parameters:

Pointers can be passed to functions to modify the values they point to.

```
void modifyValue(int* ptr) {
   *ptr = 100;
}

int value = 42;
modifyValue(&value); // Pass the address of 'value'
```

After calling modifyValue, the value variable will have a value of 100.

Arrays and Pointers:

In C++, arrays and pointers are closely related. An array name can be used as a pointer to the first element of the array.

```
int arr[] = {1, 2, 3, 4, 5};
int* arrPtr = arr;  // arrPtr points to the first element of arr
```

You can use pointer arithmetic to traverse the array.

These examples illustrate various aspects of working with pointers in C++. Pointers are essential for low-level memory manipulation and for creating flexible and efficient code. However, they require careful handling to avoid memory-related issues like segmentation faults and memory leaks.

Structures

In C++, a structure is a user-defined data type that allows you to group together variables of different data types under a single name. Each variable within a structure is called a member, and you can access these members using the dot (.) operator. Structures are typically used to represent a collection of related data. Here are examples of how to define and use structures in C++:

Defining a Structure:

To define a structure, you use the struct keyword followed by the structure name and a list of member variables enclosed in curly braces.

```
struct Student {
   int rollNumber;
   std::string name;
   double gpa;
};
```

In this example, we've defined a structure named Student with three members: rollNumber, name, and gpa.

Creating Structure Variables:

You can create variables of the structure type by specifying the structure name followed by the variable name.

```
Student student1; // Declare a variable of type Student
student1.rollNumber = 101;
student1.name = "Alice";
student1.gpa = 3.7;
```

Here, we've created a student1 variable of type Student and initialized its members.

Accessing Structure Members:

You can access the members of a structure variable using the dot (.) operator.

```
std::cout << "Roll Number: " << student1.rollNumber << std::endl;
std::cout << "Name: " << student1.name << std::endl;
```

```
std::cout << "GPA: " << student1.gpa << std::endl;
```

Array of Structures:

You can create an array of structure variables to store multiple records of the same type.

```
Student students[3]; // Array of 3 Student records

students[0].rollNumber = 101;
students[0].name = "Alice";
students[0].gpa = 3.7;

students[1].rollNumber = 102;
students[1].name = "Bob";
students[1].gpa = 3.9;

students[2].rollNumber = 103;
students[2].name = "Charlie";
students[2].gpa = 3.5;
```

Passing Structures to Functions:

You can pass structures to functions by value or by reference. Here's an example of passing a structure by value:

```
void displayStudent(Student s) {
   std::cout << "Roll Number: " << s.rollNumber << std::endl;
   std::cout << "Name: " << s.name << std::endl;
   std::cout << "GPA: " << s.gpa << std::endl;
}

Student student2 = {201, "David", 3.8};
displayStudent(student2);
```

Nested Structures:

You can nest structures inside other structures to represent complex data.

```
struct Address {
   std::string street;
   std::string city;
```

```
    std::string state;
};

struct Person {
    std::string name;
    Address address;
    int age;
};
```

In this example, the Person structure contains an Address structure as one of its members.

Initializing Structures:

You can initialize structures when declaring them.

```
Student student3 = {301, "Eve", 3.5};
```

You can also use the assignment operator to initialize structure members individually.

```
Student student4;
student4.rollNumber = 401;
student4.name = "Frank";
student4.gpa = 3.2;
```

These examples demonstrate the basics of using structures in C++ to create custom data types for organizing and managing related data. Structures are commonly used in C++ programs to represent entities with multiple attributes.

Array

In C++, an array is a collection of elements of the same data type, stored in contiguous memory locations and accessed using an index. Arrays provide a way to group multiple values of the same type under a single variable name. Here are examples of how to declare, initialize, and use arrays in C++:

Declaring and Initializing Arrays:

You can declare an array by specifying the data type of its elements, followed by the array name and the size of the array enclosed in square brackets.

```
int numbers[5]; // Declares an integer array with 5 elements
```

You can also initialize the array elements during declaration:

```
int scores[] = {90, 85, 78, 92, 88}; // Initializes an integer array without specifying the size
```

In this case, the size of the array is automatically determined by the number of elements provided in the initializer.

Accessing Array Elements:

Array elements are accessed using an index starting from 0 for the first element.

```
int firstScore = scores[0]; // Accesses the first element (90)
int thirdScore = scores[2]; // Accesses the third element (78)
```

Remember that array indices are zero-based, so the index 0 corresponds to the first element, 1 to the second element, and so on.

Modifying Array Elements:

You can assign new values to array elements using the assignment operator.

```
scores[1] = 95; // Modifies the second element (85 to 95)
```

Iterating Through Arrays:

You can use loops, such as for or while, to iterate through the elements of an array.

```
for (int i = 0; i < 5; i++) {
   std::cout << "Element " << i << ": " << scores[i] << std::endl;
}
```

This loop prints each element of the scores array along with its index.

Array Size:

You can obtain the size of an array using the sizeof operator.

```
int arraySize = sizeof(scores) / sizeof(scores[0]);
```

sizeof(scores) gives the total size of the array in bytes, and sizeof(scores[0]) gives the size of one element. Dividing the total size by the size of one element gives you the number of elements in the array.

Multidimensional Arrays:

C++ supports multidimensional arrays, such as 2D arrays for representing tables of data.

```
int matrix[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

You can access elements in a 2D array using two indices, one for the row and another for the column.

```
int element = matrix[1][2]; // Accesses element in the second row and third column (6)
```

Arrays as Function Parameters:

You can pass arrays to functions. When an array is passed to a function, it is treated as a pointer to its first element.

```
void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      std::cout << arr[i] << " ";
   }
   std::cout << std::endl;
```

}

This function takes an integer array and its size as parameters and prints all the elements of the array.

int myArray[] = {10, 20, 30, 40, 50};
printArray(myArray, 5); // Calls the function to print the array

These examples demonstrate the fundamentals of working with arrays in C++. Arrays are widely used for storing and manipulating collections of data in C++ programs.

Strings

In C++, a string is a sequence of characters represented as an object of the std::string class from the C++ Standard Library. Strings in C++ provide a convenient way to work with text data. Here are examples of how to declare, initialize, and use strings in C++:

Declaring and Initializing Strings:

To use strings in C++, you need to include the <string> header.

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

int main() {
   std::string greeting = "Hello, World!";
   std::string name;
   name = "Alice";

   std::cout << greeting << std::endl;
   std::cout << "My name is " << name << std::endl;

   return 0;
}
```

In this example, we declare and initialize a std::string variable named greeting with the string "Hello, World!" and another variable name is assigned a string later.

Concatenating Strings:

You can concatenate strings using the + operator.

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

The fullName variable now contains "John Doe."

Getting String Length:

To find the length (number of characters) of a string, you can use the length() or size() member function of the std::string class.

```
std::string sentence = "This is a sample sentence.";
int length = sentence.length(); // or sentence.size()
```

Accessing Characters in a String:

You can access individual characters in a string using the subscript operator [].

```
char firstChar = sentence[0]; // Accesses the first character 'T'
```

Substring:

You can extract a substring from a string using the substr() member function.

```
std::string phrase = "The quick brown fox";
std::string sub = phrase.substr(4, 5); // Extracts "quick" (starting at index 4, length 5)
```

Comparing Strings:

You can compare two strings using comparison operators like ==, !=, <, >, <=, and >=.

```
std::string str1 = "apple";
std::string str2 = "banana";

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

Searching for Substrings:

You can search for substrings within a string using the find() member function.

```
std::string text = "C++ programming is fun!";
size_t found = text.find("programming"); // Searches for "programming"
```

The found variable will contain the index where "programming" is found, or std::string::npos if not found.

String Input and Output:

You can read and write strings from/to the standard input and output streams.

```
std::string userInput;
std::cout << "Enter your name: ";
std::cin >> userInput;
std::cout << "Hello, " << userInput << "!" << std::endl;
```

This code prompts the user for input and then displays a greeting.

These examples illustrate some common operations with strings in C++. The std::string class provides many other useful methods for working with strings, making it a versatile and powerful tool for handling text data in C++ programs.

Function Overloading

Function overloading in C++ allows you to define multiple functions with the same name but different parameter lists. The compiler determines which function to call based on the number and types of arguments provided when the function is called. Function overloading is a form of polymorphism and helps improve code readability and maintainability. Here are examples of function overloading in C++:

```
#include <iostream>

// Function to add two integers
int add(int a, int b) {
   return a + b;
}

// Overloaded function to add two doubles
double add(double a, double b) {
   return a + b;
}

// Overloaded function to concatenate two strings
std::string add(std::string a, std::string b) {
   return a + b;
}

// Overloaded function to concatenate an int and a string
std::string add(int a, std::string b) {
   return std::to_string(a) + b;
}

int main() {
   int sum1 = add(5, 3);
   double sum2 = add(2.5, 1.5);
   std::string result1 = add("Hello, ", "World!");
   std::string result2 = add(42, " is the answer.");

   std::cout << "Sum of integers: " << sum1 << std::endl;
   std::cout << "Sum of doubles: " << sum2 << std::endl;
   std::cout << "Concatenated strings: " << result1 << std::endl;
   std::cout << "Concatenated int and string: " << result2 << std::endl;

   return 0;
```

}
In this example:

We define four overloaded functions named add with the same name but different parameter lists. The first two add functions accept integers and doubles, respectively, and return their sum. The third add function concatenates two strings, and the fourth add function concatenates an integer and a string.

In the main function, we call these overloaded functions with different argument types. The compiler determines which version of the add function to call based on the argument types provided.

Function overloading allows us to use the same function name for operations that logically make sense, even if they involve different data types.

This concept of function overloading is a form of compile-time polymorphism, as the compiler decides at compile time which function to call based on the arguments provided.

Function Templates

Function templates in C++ allow you to write functions that can work with different data types without having to write separate functions for each type. Templates provide a way to create generic functions and classes. Here are examples of function templates in C++:

Example 1: A Simple Function Template

```
#include <iostream>

// Function template to find the maximum of two values of any type
template <typename T>
T max(T a, T b) {
   return (a > b) ? a : b;
}

int main() {
   int num1 = 5, num2 = 10;
   double dbl1 = 3.14, dbl2 = 2.71;

   // Call the max function template with different types
   int maxInt = max(num1, num2);
   double maxDouble = max(dbl1, dbl2);

   std::cout << "Max integer: " << maxInt << std::endl;
   std::cout << "Max double: " << maxDouble << std::endl;

   return 0;
}
```

In this example:

We define a function template max that takes two arguments of any type T and returns the maximum of the two values.

In the main function, we call the max template with both integer and double values. The compiler generates specialized versions of the max function for each data type as needed.

Example 2: Function Template with Multiple Types

```
#include <iostream>
```

```
// Function template to swap two values of any type
template <typename T1, typename T2>
void swapValues(T1 &a, T2 &b) {
   T1 temp = a;
   a = static_cast<T1>(b);
   b = static_cast<T2>(temp);
}

int main() {
   int num1 = 5;
   double num2 = 3.14;

   std::cout << "Before swapping: num1 = " << num1 << ", num2 = " << num2 << std::endl;

   // Call the swapValues function template with different types
   swapValues(num1, num2);

   std::cout << "After swapping: num1 = " << num1 << ", num2 = " << num2 << std::endl;

   return 0;
}
```

In this example:

We define a function template swapValues that takes two arguments of different types T1 and T2 and swaps their values. The static_cast is used to ensure type compatibility during the swap.

In the main function, we call the swapValues template with an integer and a double value. The template works with different data types, allowing us to swap values of different types.

Function templates are a powerful feature in C++ that enables you to write generic and reusable code for a wide range of data types while maintaining type safety. Templates are extensively used in the C++ Standard Library to implement container classes like vectors, lists, and maps, among others.

Scope of Variables

In C++, the scope of a variable defines where in your code that variable is accessible and can be used. Variable scope is determined by where the variable is declared, and it follows a set of rules that define when and where you can access the variable. There are two primary types of variable scope: local scope and global scope. Let's explore these concepts with examples:

Local Scope:

Variables declared inside a function or a block of code have local scope, which means they are only accessible within that function or block. They are not visible or accessible from outside that function or block.

```
#include <iostream>

void myFunction() {
   int x = 10;  // Local variable with scope inside myFunction
   std::cout << "Inside myFunction: " << x << std::endl;
}

int main() {
   myFunction();
   // The following line would produce an error because x is not accessible here:
   // std::cout << "Outside myFunction: " << x << std::endl;
   return 0;
}
```

In this example, x is a local variable with scope limited to the myFunction function. Attempting to access x outside of myFunction would result in a compilation error.

Global Scope:

Variables declared outside of any function or block have global scope. They are accessible from any part of the code, both inside functions and at the global level.

```
#include <iostream>

int globalVariable = 20;  // Global variable with global scope

void myFunction() {
   std::cout << "Inside myFunction: " << globalVariable << std::endl;
```

```
}

int main() {
   std::cout << "Inside main: " << globalVariable << std::endl;
   myFunction();
   return 0;
}
```

In this example, globalVariable is declared outside of any function, making it accessible from both main and myFunction.

Block Scope:

In C++, you can also create blocks of code within functions, and variables declared within such blocks have block scope. They are accessible only within the block where they are defined.

```
#include <iostream>

int main() {
   int a = 10;  // Block-scoped variable
   {
      int b = 20;  // Block-scoped variable within an inner block
      std::cout << "Inside inner block: " << a << ", " << b << std::endl;
   }
   // The following line would produce an error because b is not accessible here:
   // std::cout << "Outside inner block: " << a << ", " << b << std::endl;
   return 0;
}
```

In this example, a is declared within the main block, and b is declared within an inner block. Variables a and b have block scope and are only accessible within their respective blocks.

These examples illustrate the concepts of local scope, global scope, and block scope in C++. Understanding variable scope is crucial for managing the visibility and lifetime of variables in your programs, which helps prevent naming conflicts and manage resources efficiently.

Type aliases (typedef / using)

In C++, type aliases are used to create alternative names for existing data types. They provide a way to make code more readable and maintainable, especially when dealing with complex data types or long type names. There are two primary ways to create type aliases in C++: typedef and the using keyword. Let's explore both methods with examples:

1. typedef (Type Alias using typedef):

The typedef keyword is used to create a type alias for an existing data type. It introduces a new name for the type, making the code more readable.

```
#include <iostream>

// Creating a type alias for int
typedef int myInt;

int main() {
   myInt x = 42;
   std::cout << "x = " << x << std::endl;

   return 0;
}
```

In this example, we use typedef to create a type alias myInt for the int data type. Now, we can use myInt to declare variables instead of using int.

2. using (Type Alias using using):

In C++11 and later versions, the using keyword can also be used to create type aliases. The using syntax is considered more modern and flexible.

```
#include <iostream>

// Creating a type alias for double
using myDouble = double;

int main() {
   myDouble pi = 3.14159265359;
   std::cout << "pi = " << pi << std::endl;
```

```
    return 0;
}
```

In this example, we use using to create a type alias myDouble for the double data type. Like typedef, we can now use myDouble to declare variables of type double.

Type Aliases for Templates:

Type aliases are particularly useful when working with templates, where type names can become complex. Here's an example of using a type alias with a template:

```
#include <iostream>
#include <vector>

// Creating a type alias for a vector of integers
using IntVector = std::vector<int>;

int main() {
    IntVector numbers = {1, 2, 3, 4, 5};

    for (const auto& num : numbers) {
        std::cout << num << " ";
    }

    std::cout << std::endl;

    return 0;
}
```

In this example, we use a type alias IntVector to simplify the declaration of a std::vector<int>. This makes the code more readable when dealing with complex data structures.

Type aliases, whether created using typedef or using, help improve code readability and maintainability by giving meaningful names to types, especially in situations involving templates or when working with library types with long and complex names.

Unions

In C++, a union is a data structure that allows you to store different data types in the same memory location. Unlike structures (which allocate separate memory for each member), unions allocate memory only for the largest member, allowing you to use the same memory location for different types of data. Here are examples of unions in C++:

Example 1: Basic Union

```
#include <iostream>

union MyUnion {
   int intValue;
   double doubleValue;
};

int main() {
   MyUnion u;
   u.intValue = 42;

   std::cout << "Integer value: " << u.intValue << std::endl;

   u.doubleValue = 3.14159;

   // Accessing the same memory location, but interpreting it as a double
   std::cout << "Double value: " << u.doubleValue << std::endl;

   return 0;
}
```

In this example:

We define a union named MyUnion that contains two members: intValue (an integer) and doubleValue (a double).

In the main function, we create a variable u of type MyUnion and assign an integer value to it. We then access and print the integer value.

Afterward, we assign a double value to the same union variable u and access and print it. The same memory location is now interpreted as a double.

Example 2: Union with Structures

You can also use unions with structures to create complex data structures. Here's an example:

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

union ContactInfo {
   struct {
      std::string firstName;
      std::string lastName;
   } person;

   struct {
      std::string companyName;
      std::string email;
   } organization;
};

int main() {
   ContactInfo info;

   info.person.firstName = "John";
   info.person.lastName = "Doe";

   std::cout << "First Name: " << info.person.firstName << std::endl;
   std::cout << "Last Name: " << info.person.lastName << std::endl;

   info.organization.companyName = "ABC Inc.";
   info.organization.email = "john.doe@abc.com";

   std::cout << "Company Name: " << info.organization.companyName << std::endl;
   std::cout << "Email: " << info.organization.email << std::endl;

   return 0;
}
```

In this example:

We define a union ContactInfo that contains two structures: person and organization. Each structure represents different types of contact information.

We can assign and access values for the person's and organization's information using the union, sharing the same memory location.

It's important to note that while unions are powerful, they require careful handling, as there is no built-in way to know which member of the union is currently valid. Unions are often used in situations where memory efficiency is critical, and the programmer is responsible for keeping track of the type of data stored in the union.

Enumerated types (enum)

In C++, an enumerated type, often referred to as enum, is a user-defined data type that consists of a set of named integer constants. Enumerations are used to represent a set of distinct values, making the code more readable and self-documenting. Here are examples of enumerated types in C++:

Example 1: Basic Enum

```
#include <iostream>

// Declare an enumeration named 'Color'
enum Color {
   Red,   // 0
   Green,  // 1
   Blue   // 2
};

int main() {
   Color selectedColor = Green;

   if (selectedColor == Green) {
      std::cout << "The selected color is Green." << std::endl;
   }

   return 0;
}
```

In this example:

We declare an enumeration named Color, which defines three named constants: Red, Green, and Blue. These constants are assigned integer values starting from 0 by default (unless specified).

In the main function, we declare a variable selectedColor of type Color and initialize it with the value Green. We then compare selectedColor to Green to check if it's the selected color.

Example 2: Enum with Explicit Values

You can assign explicit values to enum constants:

```
#include <iostream>

// Declare an enumeration named 'Day' with explicit values
enum Day {
   Monday = 1,
   Tuesday = 2,
   Wednesday = 3,
   Thursday = 4,
   Friday = 5,
   Saturday = 6,
   Sunday = 7
};

int main() {
   Day today = Wednesday;
   std::cout << "Today is day " << today << std::endl;
   return 0;
}
```

Here, we assign explicit integer values to the days of the week.

Example 3: Enum Class (Scoped Enum)

C++11 introduced enum classes, also known as scoped enums, which provide better scoping and type safety:

```
#include <iostream>

// Declare an enum class named 'Season' with scoped values
enum class Season {
   Spring,
   Summer,
   Autumn,
   Winter
};

int main() {
   Season currentSeason = Season::Autumn;

   switch (currentSeason) {
      case Season::Spring:
         std::cout << "It's Spring!" << std::endl;
         break;
```

```
        case Season::Summer:
            std::cout << "It's Summer!" << std::endl;
            break;
        case Season::Autumn:
            std::cout << "It's Autumn!" << std::endl;
            break;
        case Season::Winter:
            std::cout << "It's Winter!" << std::endl;
            break;
    }

    return 0;
}
```

In this example, we declare an enum class Season with scoped values. Scoped enums provide better encapsulation and prevent name clashes.

Enumerated types are useful for improving code readability and maintainability by giving meaningful names to constants. They are commonly used to represent sets of related values, such as days of the week, months, or options in a menu.

Class

In C++, a class is a blueprint for creating objects. It defines a template for objects, specifying their attributes (data members) and behaviors (member functions). Classes are fundamental to object-oriented programming (OOP) and allow you to create structured and organized code. Here's an example of a class in C++:

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

// Define a class named 'Person'
class Person {
public: // Access specifier
   // Data members (attributes)
   std::string name;
   int age;

   // Member functions (methods)
   void introduce() {
      std::cout << "Hi, I'm " << name << " and I'm " << age << " years old." << std::endl;
   }

   void growOlder() {
      age++;
   }
};

int main() {
   // Create an object of the 'Person' class
   Person person1;

   // Set the data members of the object
   person1.name = "Alice";
   person1.age = 25;

   // Call member functions on the object
   person1.introduce();
   person1.growOlder();
   person1.introduce();

   return 0;
}
```

In this example:

We define a class named Person using the class keyword. Inside the class definition, we declare data members (name and age) and member functions (introduce and growOlder).

Data members represent the attributes or properties of objects, and member functions define their behaviors.

We create an object of the Person class named person1 in the main function.

We set the data members of the object using the dot (.) operator.

We call the member functions on the object to perform actions associated with it.

The public: access specifier indicates that data members and member functions are accessible from outside the class. There are also private and protected access specifiers for controlling access to class members.

This is a basic example of a C++ class, but classes can become much more complex and can have constructors, destructors, access control, inheritance, and other features that make them powerful tools for modeling real-world entities in software. Classes and objects are the foundation of object-oriented programming and are used extensively in C++ programs for organizing and encapsulating data and behavior.

Constructors

In C++, constructors are special member functions within a class that are automatically called when an object of the class is created. Constructors are used to initialize the object's data members and perform any necessary setup. Here are examples of constructors in C++:

1. Default Constructor:

A default constructor is a constructor with no parameters. It is automatically called when an object is created if no other constructor is explicitly defined. It initializes the object's data members to default values.

```
#include <iostream>

class MyClass {
public:
   // Default constructor
   MyClass() {
      std::cout << "Default constructor called." << std::endl;
      // Initialize data members here
   }
};

int main() {
   MyClass obj; // Creating an object calls the default constructor
   return 0;
}
```

In this example, when an object of MyClass is created using MyClass obj;, the default constructor is automatically called.

2. Parameterized Constructor:

A parameterized constructor is a constructor that accepts one or more parameters. It allows you to initialize the object's data members with specific values at the time of object creation.

```
#include <iostream>

class Student {
public:
   // Parameterized constructor
```

```
    Student(std::string name, int rollNumber) {
        this->name = name;
        this->rollNumber = rollNumber;
    }

    void displayInfo() {
        std::cout << "Name: " << name << ", Roll Number: " << rollNumber << std::endl;
    }

private:
    std::string name;
    int rollNumber;
};

int main() {
    Student student1("Alice", 101); // Creating an object with parameterized constructor
    student1.displayInfo();
    return 0;
}
```

In this example, the Student class has a parameterized constructor that takes name and rollNumber as parameters. When we create an object of Student using Student student1("Alice", 101);, the parameterized constructor is called to initialize the object's data members.

3. Copy Constructor:

A copy constructor is a constructor that creates a new object by copying the values of another object. It is invoked when one object is used to initialize another object of the same class.

```
#include <iostream>

class Point {
public:
    // Parameterized constructor
    Point(int x, int y) : x(x), y(y) {}

    // Copy constructor
    Point(const Point& other) {
        x = other.x;
        y = other.y;
    }
```

```
    void display() {
        std::cout << "Point(" << x << ", " << y << ")" << std::endl;
    }

private:
    int x, y;
};

int main() {
    Point p1(1, 2);
    Point p2 = p1; // Copy constructor is called here
    p1.display();
    p2.display();
    return 0;
}
```

In this example, the Point class has a copy constructor that allows us to create a new Point object p2 by copying the values from an existing Point object p1.

These are just some of the types of constructors in C++. Constructors provide a way to initialize objects with specific values and perform any necessary setup when objects are created. They are essential in defining the behavior of classes and objects in your C++ programs.

Overloading Constructors

In C++, you can overload constructors by defining multiple constructors with different parameter lists in the same class. Constructor overloading allows you to create objects with different initializations based on the arguments provided during object creation. Here are examples of constructor overloading in C++:

1. Overloading Constructors with Different Parameters:

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

class Person {
public:
   // Parameterized constructor
   Person(std::string n, int a) : name(n), age(a) {
      std::cout << "Parameterized constructor called." << std::endl;
   }

   // Default constructor (no parameters)
   Person() : name("Unknown"), age(0) {
      std::cout << "Default constructor called." << std::endl;
   }

   void displayInfo() {
      std::cout << "Name: " << name << ", Age: " << age << std::endl;
   }

private:
   std::string name;
   int age;
};

int main() {
   // Creating objects using different constructors
   Person person1("Alice", 25); // Parameterized constructor
   Person person2;              // Default constructor

   // Displaying object information
   person1.displayInfo();
   person2.displayInfo();
```

```
    return 0;
}
```

In this example:

The Person class has two constructors: a parameterized constructor that takes name and age as arguments and a default constructor that initializes the object with default values.

When objects are created, the appropriate constructor is called based on the arguments provided during object creation.

2. Overloading Constructors with Default Values:

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

class Book {
public:
    // Constructor with default values
    Book(std::string t = "Unknown", std::string a = "Unknown", int y = 0)
        : title(t), author(a), year(y) {
        std::cout << "Constructor called." << std::endl;
    }

    void displayInfo() {
        std::cout << "Title: " << title << ", Author: " << author << ", Year: " << year << std::endl;
    }

private:
    std::string title;
    std::string author;
    int year;
};

int main() {
    // Creating objects using different constructors
    Book book1("The Catcher in the Rye", "J.D. Salinger", 1951);
    Book book2; // Using default values

    // Displaying object information
    book1.displayInfo();
    book2.displayInfo();
```

```
    return 0;
}
```

In this example:

The Book class has a constructor with default values for title, author, and year. If no arguments are provided during object creation, the default values are used.

This allows for more flexibility when creating objects, as you can choose to specify some, all, or none of the constructor arguments.

Constructor overloading is a useful feature in C++ that enables you to create objects with different initializations based on the specific requirements of your program. It enhances code reusability and flexibility when working with classes and objects.

Member initialization in constructors

In C++, you can use member initialization lists in constructors to initialize class data members before the body of the constructor is executed. Member initialization lists are a more efficient way to initialize data members, especially for complex types or objects, and they help avoid unnecessary default constructions and assignment operations. Here are examples of member initialization in constructors:

1. Initializing Primitive Data Members:

```
#include <iostream>

class MyClass {
public:
   // Constructor with member initialization
   MyClass(int x, double y) : num(x), value(y) {
      // No need to initialize 'num' and 'value' inside the constructor body.
   }

   void display() {
      std::cout << "num = " << num << ", value = " << value << std::endl;
   }

private:
   int num;
   double value;
};

int main() {
   MyClass obj(42, 3.14);
   obj.display();
   return 0;
}
```

In this example:

The MyClass constructor takes two parameters, x and y.

In the member initialization list, we initialize the data members num and value with the values of x and y, respectively.

Initializing data members in the member initialization list is more efficient than initializing them in the constructor body.

2. Initializing Object Data Members:

```
#include <iostream>

class Point {
public:
   // Constructor with member initialization for objects
   Point(int x, int y) : xCoord(x), yCoord(y) {
      // No need to initialize 'xCoord' and 'yCoord' inside the constructor body.
   }

   void display() {
      std::cout << "xCoord = " << xCoord << ", yCoord = " << yCoord << std::endl;
   }

private:
   int xCoord;
   int yCoord;
};

class Line {
public:
   // Constructor with member initialization for objects
   Line(Point start, Point end) : startPoint(start), endPoint(end) {
      // No need to initialize 'startPoint' and 'endPoint' inside the constructor body.
   }

   void display() {
      std::cout << "Start Point: ";
      startPoint.display();
      std::cout << "End Point: ";
      endPoint.display();
   }

private:
   Point startPoint;
   Point endPoint;
};

int main() {
   Point p1(1, 2);
```

```
    Point p2(3, 4);

    Line line(p1, p2);
    line.display();

    return 0;
}
```

In this example:

The Point class has a constructor that takes x and y coordinates as parameters and initializes the xCoord and yCoord data members in the member initialization list.

The Line class has a constructor that takes two Point objects and initializes the startPoint and endPoint data members in the member initialization list.

Initializing object data members in the member initialization list is important to avoid invoking default constructors and assignment operators for objects, which can be costly for user-defined types.

Using member initialization lists is a good practice in C++ because it ensures that data members are properly initialized and can lead to more efficient code when working with objects.

Pointers to Classes

In C++, you can use pointers to access and manipulate objects of a class. Pointers to classes are used when you need to dynamically allocate objects on the heap or when you want to pass objects to functions by reference. Here are examples of using pointers to classes in C++:

1. Creating and Using Pointers to Objects:

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

class Student {
public:
   Student(std::string n, int r) : name(n), rollNumber(r) {}

   void displayInfo() {
      std::cout << "Name: " << name << ", Roll Number: " << rollNumber << std::endl;
   }

private:
   std::string name;
   int rollNumber;
};

int main() {
   // Creating a pointer to a Student object
   Student* studentPtr = nullptr;

   // Dynamically allocating a Student object on the heap
   studentPtr = new Student("Alice", 101);

   // Accessing object's member function using the pointer
   studentPtr->displayInfo();

   // Deallocating memory (don't forget to delete dynamically allocated objects)
   delete studentPtr;

   return 0;
}
```

In this example:

We declare a pointer to a Student object using Student* studentPtr.

We dynamically allocate a Student object on the heap using new. The pointer studentPtr now points to the dynamically allocated object.

We access the member function displayInfo() of the Student object using the pointer.

It's important to deallocate the memory using delete to prevent memory leaks.

2. Passing Objects to Functions using Pointers:

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

class Student {
public:
   Student(std::string n, int r) : name(n), rollNumber(r) {}

   void displayInfo() {
      std::cout << "Name: " << name << ", Roll Number: " << rollNumber << std::endl;
   }

private:
   std::string name;
   int rollNumber;
};

// Function that takes a pointer to a Student object
void printStudentInfo(Student* studentPtr) {
   studentPtr->displayInfo();
}

int main() {
   Student alice("Alice", 101);

   // Passing the object 'alice' to the function using a pointer
   printStudentInfo(&alice);

   return 0;
}
```

In this example:

We define a function printStudentInfo that takes a pointer to a Student object as its argument.

We create a Student object named alice in the main function.

We pass the object alice to the printStudentInfo function using a pointer.

Using pointers to classes allows you to work with objects more dynamically and efficiently, especially when dealing with dynamic memory allocation or passing objects to functions. However, you must be careful to manage memory properly to avoid memory leaks or accessing invalid memory locations.

Overloading Operators

In C++, operator overloading allows you to define how operators should behave when applied to objects of user-defined classes. You can create custom operator functions for your classes to provide meaningful operations. Here are examples of operator overloading in C++:

1. Overloading the + Operator:

```
#include <iostream>

class Complex {
public:
   Complex(double real, double imag) : real(real), imag(imag) {}

   // Overloading the + operator
   Complex operator+(const Complex& other) const {
      return Complex(real + other.real, imag + other.imag);
   }

   void display() const {
      std::cout << real << " + " << imag << "i" << std::endl;
   }

private:
   double real;
   double imag;
};

int main() {
   Complex a(1.0, 2.0);
   Complex b(3.0, 4.0);

   Complex result = a + b; // Operator+ overload
   result.display();

   return 0;
}
```

In this example:

We define a Complex class to represent complex numbers.

We overload the + operator by creating a member function named operator+. This function takes a const Complex& parameter, which represents the right-hand operand, and returns a new Complex object as the result of the addition.

We can then use the + operator to add two Complex objects, as shown in the main function.

2. Overloading the << Operator (for Stream Insertion):

```
#include <iostream>

class Point {
public:
   Point(int x, int y) : x(x), y(y) {}

   // Overloading the << operator for stream insertion
   friend std::ostream& operator<<(std::ostream& out, const Point& p) {
      out << "(" << p.x << ", " << p.y << ")";
      return out;
   }

private:
   int x;
   int y;
};

int main() {
   Point p(3, 4);

   // Using the << operator to display a Point object
   std::cout << "Point p: " << p << std::endl;

   return 0;
}
```

In this example:

We define a Point class to represent 2D points with x and y coordinates.

We overload the << operator as a friend function, allowing us to stream the Point object's data members into an output stream (std::ostream).

We can then use the << operator to display a Point object directly.

Operator overloading allows you to make your classes more intuitive and user-friendly by defining how operators should work with objects of your class. It is a powerful feature for creating expressive and natural interfaces for user-defined types.

Keyword ‘this

In C++, the this keyword is a pointer that represents the current object. It is a special pointer that is automatically available in member functions of a class. this is used to refer to the object on which the member function is called. It allows you to access the object's data members and call other member functions within the class. Here are examples of using the this keyword in C++:

1. Accessing Data Members with this:

```
#include <iostream>

class MyClass {
public:
   void setData(int x, int y) {
      // Use 'this' to access data members
      this->x = x;
      this->y = y;
   }

   void displayData() {
      std::cout << "x = " << this->x << ", y = " << this->y << std::endl;
   }

private:
   int x;
   int y;
};

int main() {
   MyClass obj;
   obj.setData(10, 20);
   obj.displayData();
   return 0;
}
```

In this example:

The setData member function sets the x and y data members of the object using this.

this->x and this->y explicitly refer to the data members of the current object.

2. Avoiding Name Conflicts:

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

class Person {
public:
    Person(std::string name, int age) : name(name), age(age) {}

    // Compare ages of two Person objects
    bool isOlderThan(Person other) {
        // Avoid name conflict by using 'this'
        return this->age > other.age;
    }

private:
    std::string name;
    int age;
};

int main() {
    Person alice("Alice", 25);
    Person bob("Bob", 30);

    if (alice.isOlderThan(bob)) {
        std::cout << alice.isOlderThan(bob) << std::endl;
        std::cout << alice.isOlderThan(bob) << std::endl;
        std::cout << "Alice is older than Bob." << std::endl;
    } else {
        std::cout << "Alice is not older than Bob." << std::endl;
    }

    return 0;
}
```

In this example:

The isOlderThan member function takes another Person object as a parameter and compares their ages.

Inside the function, we use this->age to refer to the age of the current object and other.age to refer to the age of the parameter object.

Using this helps avoid a naming conflict between the parameter age and the data member age.

The this keyword is especially useful when there is a need to disambiguate between data members and function parameters that share the same name. It is automatically available in all member functions and provides a way to access the current object's data members and call other member functions within the class.

Static Members

In C++, static members are members of a class that are shared among all objects of the class rather than being specific to each object. These members are associated with the class itself, not with individual instances of the class. Static members are often used to store class-level data and perform class-level operations. Here are examples of static members in C++:

1. Static Data Members:

```
#include <iostream>

class MyClass {
public:
   // Static data member
   static int count;

   MyClass() {
      count++; // Increment the static count for each object created
   }

   static int getCount() {
      return count; // Access the static count
   }
};

// Initialize the static data member
int MyClass::count = 0;

int main() {
   MyClass obj1;
   MyClass obj2;
   MyClass obj3;

   std::cout << "Total objects created: " << MyClass::getCount() << std::endl;

   return 0;
}
```

In this example:

The MyClass class has a static data member named count, which is shared among all instances of the class.

Each time an object of MyClass is created, the constructor increments the static count.

The getCount static member function allows us to access the value of the static data member.

We initialize the static data member count outside the class definition.

2. Static Member Functions:

```
#include <iostream>

class MathUtility {
public:
   // Static member function to calculate the square of a number
   static double square(double x) {
      return x * x;
   }
};

int main() {
   double number = 5.0;
   double squared = MathUtility::square(number); // Calling a static member function

   std::cout << "Square of " << number << " is " << squared << std::endl;

   return 0;
}
```

In this example:

The MathUtility class has a static member function named square that calculates the square of a number.

We can call the static member function using the class name (MathUtility::square(number)).

Static members are useful when you want to store class-level data or provide class-level functionality that doesn't depend on individual object instances. They are shared among all objects of the class and can be accessed without creating an instance of the class.

Const Member Functions

In C++, a const member function is a member function of a class that promises not to modify the state of the object on which it is called. When a member function is declared as const, it is indicating to the compiler that it will not modify any non-static data members of the class. This allows you to call the member function on const objects and objects that are non-const. Here are examples of const member functions in C++:

1. Const Member Function Example:

```
#include <iostream>

class MyClass {
public:
   MyClass(int val) : value(val) {}

   // A const member function that does not modify 'value'
   int getValue() const {
      return value;
   }

   // A non-const member function that can modify 'value'
   void setValue(int val) {
      value = val;
   }

private:
   int value;
};

int main() {
   const MyClass obj1(42); // Const object
   MyClass obj2(10);

   // Calling const member function on const and non-const objects
   std::cout << "obj1's value: " << obj1.getValue() << std::endl;
   std::cout << "obj2's value: " << obj2.getValue() << std::endl;

   // obj2.setValue(20); // This would result in a compilation error

   return 0;
}
```

In this example:

We define a MyClass class with a const member function getValue() that returns the value of the value data member without modifying it.

We also have a non-const member function setValue() that can modify the value data member.

We create both const and non-const objects of MyClass and call the getValue() function on both of them. The const object obj1 can only call const member functions.

2. Const Member Function for Immutable Objects:

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

class Person {
public:
   Person(const std::string& n, int a) : name(n), age(a) {}

   // Const member function to display person's information
   void displayInfo() const {
      std::cout << "Name: " << name << ", Age: " << age << std::endl;
   }

private:
   std::string name;
   int age;
};

int main() {
   const Person alice("Alice", 25); // Const object
   Person bob("Bob", 30);

   alice.displayInfo(); // Calling a const member function

   // bob.displayInfo(); // This would also work for non-const objects

   return 0;
}
```

In this example:

We have a Person class with a const member function displayInfo() that displays the person's information without modifying it.

We create both const and non-const objects of Person and call the displayInfo() function on them. The const object alice can only call const member functions, but the non-const object bob can call both const and non-const member functions.

const member functions are essential for creating "read-only" interfaces to objects. They allow you to work with const objects and ensure that the state of the object is not modified when calling certain member functions.

Class Templates

In C++, class templates allow you to define generic classes that can work with different data types. Class templates are similar to function templates, but they define classes instead. You can create class templates to define a blueprint for a class that can work with various data types without having to rewrite the code for each data type. Here are examples of class templates in C++:

1. Simple Class Template:

```
#include <iostream>

// Class template for a generic Pair
template <typename T>
class Pair {
public:
   Pair(T first, T second) : first(first), second(second) {}

   void display() {
      std::cout << "(" << first << ", " << second << ")" << std::endl;
   }

private:
   T first;
   T second;
};

int main() {
   Pair<int> intPair(1, 2);
   Pair<double> doublePair(3.14, 2.71);

   intPair.display();
   doublePair.display();

   return 0;
}
```

In this example:

We define a class template Pair that takes a single template type parameter T.

The class template Pair has two data members of type T and a constructor to initialize those members.

We create instances of Pair for both int and double data types.

2. Class Template with Multiple Type Parameters:

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

// Class template for a generic KeyValue pair
template <typename KeyType, typename ValueType>
class KeyValue {
public:
   KeyValue(KeyType key, ValueType value) : key(key), value(value) {}

   void display() {
      std::cout << "Key: " << key << ", Value: " << value << std::endl;
   }

private:
   KeyType key;
   ValueType value;
};

int main() {
   KeyValue<int, std::string> person(101, "Alice");
   KeyValue<std::string, double> temperature("Today", 28.5);

   person.display();
   temperature.display();

   return 0;
}
```

In this example:

We define a class template KeyValue that takes two template type parameters, KeyType and ValueType.

The class template KeyValue has data members key and value, each of a different template type.

We create instances of KeyValue for different data types, such as int with std::string and std::string with double.

Class templates are powerful tools for creating generic and reusable code. They allow you to write code that can work with various data types while maintaining type safety. Template classes are commonly used in C++ to create container classes like vectors, stacks, and queues that can hold different data types.

Template Specialization

Template specialization in C++ allows you to provide custom implementations for specific data types while still using a generic template for other data types. It's a way to customize the behavior of a template for certain cases. Here are examples of template specialization in C++:

1. Function Template Specialization:

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

// Generic template
template <typename T>
void printType(T value) {
   std::cout << "Generic: " << value << std::endl;
}

// Template specialization for strings
template <>
void printType(std::string value) {
   std::cout << "String: " << value << std::endl;
}

int main() {
   int num = 42;
   double pi = 3.14159;
   std::string text = "Hello, World!";

   printType(num);   // Calls the generic version
   printType(pi);    // Calls the generic version
   printType(text);  // Calls the specialized version

   return 0;
}
```

In this example:

We define a generic function template printType that can print values of any data type.

We then provide a template specialization for the std::string type, which customizes the behavior for strings.

When we call printType, the generic version is used for int and double, but the specialized version is used for std::string.

2. Class Template Specialization:

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

// Generic class template
template <typename T>
class Printer {
public:
   void print(T value) {
      std::cout << "Generic: " << value << std::endl;
   }
};

// Class template specialization for strings
template <>
class Printer<std::string> {
public:
   void print(std::string value) {
      std::cout << "String: " << value << std::endl;
   }
};

int main() {
   Printer<int> intPrinter;
   Printer<double> doublePrinter;
   Printer<std::string> stringPrinter;

   intPrinter.print(42);        // Calls the generic version
   doublePrinter.print(3.14159);  // Calls the generic version
   stringPrinter.print("Hello");  // Calls the specialized version

   return 0;
}
```

In this example:

We define a generic class template Printer with a member function print that can print values of any data type.

We provide a class template specialization for the std::string type, which customizes the behavior for strings.

When we create instances of Printer for different data types, the generic version of the class is used for int and double, but the specialized version is used for std::string.

Template specialization is a powerful feature in C++ that allows you to tailor the behavior of templates for specific data types while still benefiting from the generic template for other cases. It is commonly used in libraries and frameworks to provide efficient and specialized implementations for certain types.

Namespace

In C++, a namespace is a way to group related code together and prevent naming conflicts. It allows you to create a separate scope for your code, so you can define variables, functions, or classes with the same names in different namespaces without causing conflicts. Here are examples of using namespaces in C++:

1. Creating a Namespace:

```
#include <iostream>

// Define a namespace named 'MyNamespace'
namespace MyNamespace {
   int value = 42;

   void display() {
      std::cout << "Value from MyNamespace: " << value << std::endl;
   }
}

int main() {
   // Access 'value' and 'display' from 'MyNamespace'
   MyNamespace::display();
   std::cout << "Value from main: " << MyNamespace::value << std::endl;

   return 0;
}
```

In this example:

We define a namespace named MyNamespace and place a variable value and a function display inside it.

Inside the main function, we access the value and display from the MyNamespace using the MyNamespace:: prefix.

2. Nested Namespaces:

```
#include <iostream>

namespace OuterNamespace {
```

```
    int value = 10;

    namespace InnerNamespace {
        int value = 20;
    }
}

int main() {
    std::cout << "Value from OuterNamespace: " << OuterNamespace::value << std::endl;
    std::cout << "Value from InnerNamespace: " << OuterNamespace::InnerNamespace::value
<< std::endl;

    return 0;
}
```

In this example:

We define an outer namespace OuterNamespace and an inner namespace InnerNamespace inside it.

We place different variables named value in both namespaces.

In the main function, we access the value in each namespace using the :: operator to specify the namespace scope.

3. Using Directive:

```
#include <iostream>

namespace A {
    int value = 10;
}

namespace B {
    int value = 20;
}

int main() {
    using namespace A; // Using directive for namespace A

    std::cout << "Value from A: " << value << std::endl;
    // std::cout << "Value from B: " << B::value << std::endl; // Error: 'value' is ambiguous
```

```
    return 0;
}
```

In this example:

We use the using namespace A; directive to bring the symbols from namespace A into the current scope.

Without the using directive, we would have to use A::value to access the variable. However, using the directive may lead to naming conflicts if there are variables with the same name in multiple namespaces.

Namespaces are particularly useful in large projects and when incorporating external libraries, as they help prevent naming clashes and organize code into logical units. They are an essential feature for writing modular and maintainable C++ code.

Friendship (Friend Functions & Friend Classes)

In C++, friendship is a mechanism that allows certain functions or classes to access the private and protected members of a class, even though they are not part of the class itself. Friend functions and friend classes are declared with the friend keyword inside the class definition. Here are examples of friend functions and friend classes in C++:

1. Friend Function:

```
#include <iostream>

// Forward declaration of the class
class MyClass;

// Friend function declaration
void printValue(const MyClass& obj);

class MyClass {
public:
   MyClass(int val) : value(val) {}

   // Declaring the friend function
   friend void printValue(const MyClass& obj);

private:
   int value;
};

// Friend function definition
void printValue(const MyClass& obj) {
   std::cout << "Value from MyClass: " << obj.value << std::endl;
}

int main() {
   MyClass obj(42);
   printValue(obj); // Call the friend function

   return 0;
}
```

In this example:

We declare a friend function printValue that can access the private value member of the MyClass class.

Inside the main function, we create an instance of MyClass and call the friend function to print the value member.

2. Friend Class:

```
#include <iostream>

class FriendClass;

class MyClass {
public:
   MyClass(int val) : value(val) {}

   // Declaring FriendClass as a friend class
   friend class FriendClass;

private:
   int value;
};

class FriendClass {
public:
   void printValue(const MyClass& obj) {
      std::cout << "Value from MyClass: " << obj.value << std::endl;
   }
};

int main() {
   MyClass obj(42);
   FriendClass friendObj;
   friendObj.printValue(obj); // Call a member function of FriendClass

   return 0;
}
```

In this example:

We declare a friend class FriendClass inside the MyClass class.

The FriendClass class has a member function printValue that can access the private value member of the MyClass class.

Inside the main function, we create an instance of MyClass and an instance of FriendClass, and then we call the member function of FriendClass to print the value member of MyClass.

Friendship allows you to grant specific functions or classes access to the private and protected members of a class, which can be useful in scenarios where you want to provide controlled access to certain parts of your class's implementation without making those members public. However, it should be used with caution, as it can break encapsulation and increase code coupling.

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create a new class based on an existing class. The new class (called the derived class or subclass) inherits the attributes and behaviors (data members and member functions) of the existing class (called the base class or superclass). In C++, inheritance is implemented using the class or struct keyword, followed by a colon (:) and the access specifier (public, protected, or private). Here are examples of inheritance in C++:

1. Single Inheritance:

```
#include <iostream>

// Base class (superclass)
class Shape {
public:
   void setDimensions(float length, float width) {
      this->length = length;
      this->width = width;
   }

   void displayDimensions() {
      std::cout << "Length: " << length << ", Width: " << width << std::endl;
   }

private:
   float length;
   float width;
};

// Derived class (subclass) inheriting from Shape
class Rectangle : public Shape {
public:
   float calculateArea() {
      return length * width; // Access base class members
   }
};

int main() {
   Rectangle rect;
   rect.setDimensions(5.0, 3.0);
   rect.displayDimensions();
```

```
    std::cout << "Area: " << rect.calculateArea() << std::endl;

    return 0;
}
```

In this example:

Shape is the base class, and Rectangle is the derived class.

The Rectangle class inherits the setDimensions() and displayDimensions() member functions from the Shape class.

The calculateArea() function in Rectangle uses the length and width data members inherited from Shape to calculate the area.

2. Multiple Inheritance:

```
#include <iostream>

// Base class 1
class Animal {
public:
    void speak() {
        std::cout << "Animal speaks" << std::endl;
    }
};

// Base class 2
class Mammal {
public:
    void breathe() {
        std::cout << "Mammal breathes" << std::endl;
    }
};

// Derived class inheriting from both Animal and Mammal
class Dog : public Animal, public Mammal {
public:
    void bark() {
        std::cout << "Dog barks" << std::endl;
    }
};
```

```
int main() {
   Dog myDog;
   myDog.speak();  // Access function from Animal
   myDog.breathe();  // Access function from Mammal
   myDog.bark();  // Access function from Dog

   return 0;
}
```

In this example:

Animal and Mammal are base classes, and Dog is the derived class that inherits from both base classes.

The Dog class inherits the speak() function from Animal and the breathe() function from Mammal.

3. Hierarchical Inheritance:

```
#include <iostream>

// Base class
class Vehicle {
public:
   void start() {
      std::cout << "Vehicle started" << std::endl;
   }
};

// Derived class 1 inheriting from Vehicle
class Car : public Vehicle {
public:
   void drive() {
      std::cout << "Car is being driven" << std::endl;
   }
};

// Derived class 2 inheriting from Vehicle
class Bicycle : public Vehicle {
public:
   void pedal() {
      std::cout << "Bicycle is being pedaled" << std::endl;
   }
```

```
};

int main() {
    Car myCar;
    Bicycle myBike;

    myCar.start();
    myCar.drive();

    myBike.start();
    myBike.pedal();

    return 0;
}
```

In this example:

Vehicle is the base class, and both Car and Bicycle are derived classes that inherit from Vehicle.

Both Car and Bicycle have their own member functions (drive() and pedal()), and they inherit the start() function from Vehicle.

Inheritance allows you to create a hierarchy of classes, reuse code, and model real-world relationships between objects. It promotes code reuse and supports the principles of OOP like encapsulation and polymorphism.

Polymorphism

Polymorphism is one of the fundamental principles of object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common base class. Polymorphism enables you to write code that can work with objects of different derived classes through a common interface. In C++, polymorphism is often achieved through virtual functions and inheritance. Here are examples of polymorphism in C++:

1. Polymorphism with Virtual Functions:

```
#include <iostream>

// Base class
class Animal {
public:
   // Declare a virtual function
   virtual void makeSound() {
      std::cout << "Animal makes a sound" << std::endl;
   }
};

// Derived class 1
class Dog : public Animal {
public:
   // Override the virtual function
   void makeSound() override {
      std::cout << "Dog barks" << std::endl;
   }
};

// Derived class 2
class Cat : public Animal {
public:
   // Override the virtual function
   void makeSound() override {
      std::cout << "Cat meows" << std::endl;
   }
};

int main() {
   Animal* animalPtr;
```

```
    Dog myDog;
    Cat myCat;

    animalPtr = &myDog;
    animalPtr->makeSound(); // Calls Dog's makeSound()

    animalPtr = &myCat;
    animalPtr->makeSound(); // Calls Cat's makeSound()

    return 0;
}
```

In this example:

We have a base class Animal with a virtual function makeSound().

Two derived classes, Dog and Cat, override the makeSound() function.

In the main function, we use a pointer of type Animal* to point to objects of both Dog and Cat. When we call makeSound() through the pointer, it invokes the appropriate function based on the actual object type.

2. Polymorphism with Abstract Base Classes:

```
#include <iostream>

// Abstract base class
class Shape {
public:
    // Pure virtual function makes the class abstract
    virtual double calculateArea() const = 0;
};

// Derived class 1
class Rectangle : public Shape {
public:
    Rectangle(double l, double w) : length(l), width(w) {}

    double calculateArea() const override {
        return length * width;
    }

private:
```

```
    double length;
    double width;
};

// Derived class 2
class Circle : public Shape {
public:
    Circle(double r) : radius(r) {}

    double calculateArea() const override {
        return 3.14159 * radius * radius;
    }

private:
    double radius;
};

int main() {
    Rectangle rect(5.0, 3.0);
    Circle circle(2.5);

    Shape* shapes[2] = {&rect, &circle};

    for (const Shape* shapePtr : shapes) {
        std::cout << "Area: " << shapePtr->calculateArea() << std::endl;
    }

    return 0;
}
```

In this example:

We have an abstract base class Shape with a pure virtual function calculateArea(). An abstract class cannot be instantiated; it provides a common interface for derived classes.

Two derived classes, Rectangle and Circle, inherit from Shape and provide concrete implementations for the calculateArea() function.

In the main function, we create objects of both Rectangle and Circle and store them in an array of Shape*. We can then loop through the array and calculate the areas using polymorphism.

Polymorphism allows you to write flexible and extensible code that can work with objects of different classes through a common interface. It simplifies code maintenance and promotes code reuse in object-oriented programming.

Virtual Members

In C++, a virtual member is a member function declared in a base class with the virtual keyword. Virtual members enable dynamic dispatch, which means that the appropriate derived class's version of the function is called at runtime when invoked through a pointer or reference to a base class object. This feature is essential for achieving polymorphism in C++. Here are examples of virtual functions and virtual destructors in C++:

1. Virtual Functions:

```
#include <iostream>

// Base class with a virtual function
class Shape {
public:
   virtual void display() {
      std::cout << "This is a Shape." << std::endl;
   }
};

// Derived class
class Circle : public Shape {
public:
   void display() override {
      std::cout << "This is a Circle." << std::endl;
   }
};

// Derived class
class Rectangle : public Shape {
public:
   void display() override {
      std::cout << "This is a Rectangle." << std::endl;
   }
};

int main() {
   Shape* shapes[2];
   shapes[0] = new Circle();
   shapes[1] = new Rectangle();

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

```
        shapes[i]->display(); // Calls the appropriate derived class's display()
        delete shapes[i];
    }

    return 0;
}
```

In this example:

Shape is the base class with a virtual function display(), which is overridden in the derived classes Circle and Rectangle.

In the main function, we create an array of pointers to Shape objects and assign instances of Circle and Rectangle to them.

When we call the display() function on each object, the appropriate derived class's version of the function is called, demonstrating dynamic dispatch.

2. Virtual Destructors:

```
#include <iostream>

// Base class with a virtual destructor
class Base {
public:
    Base() {
        std::cout << "Base constructor" << std::endl;
    }

    virtual ~Base() {
        std::cout << "Base destructor" << std::endl;
    }
};

// Derived class
class Derived : public Base {
public:
    Derived() {
        std::cout << "Derived constructor" << std::endl;
    }

    ~Derived() override {
        std::cout << "Derived destructor" << std::endl;
```

```
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Calls the Derived destructor through the virtual destructor

    return 0;
}
```

In this example:

Base is the base class with a virtual destructor. The destructor is declared as virtual to ensure that the correct destructor is called when an object of the derived class is deleted through a pointer to the base class.

Derived is the derived class that inherits from Base and defines its destructor.

In the main function, we create a pointer to a Base object and assign it to an instance of Derived. When we delete the pointer, the virtual destructor ensures that both the base and derived class destructors are called in the correct order.

Using virtual members, especially virtual functions and virtual destructors, is crucial for achieving proper polymorphism and avoiding resource leaks when working with base and derived classes in C++.

Abstract base class

In C++, an abstract base class is a class that cannot be instantiated and is used as a blueprint for other classes. Abstract base classes define a common interface that must be implemented by their derived classes. Abstract classes often contain pure virtual functions, making them incomplete and requiring derived classes to provide concrete implementations. Here's an example of an abstract base class in C++:

```
#include <iostream>

// Abstract base class (Shape)
class Shape {
public:
   // Pure virtual function for calculating area
   virtual double calculateArea() const = 0;

   // Concrete member function
   void printArea() const {
      std::cout << "Area: " << calculateArea() << std::endl;
   }
};

// Derived class (Rectangle)
class Rectangle : public Shape {
public:
   Rectangle(double l, double w) : length(l), width(w) {}

   double calculateArea() const override {
      return length * width;
   }

private:
   double length;
   double width;
};

// Derived class (Circle)
class Circle : public Shape {
public:
   Circle(double r) : radius(r) {}

   double calculateArea() const override {
```

```
        return 3.14159 * radius * radius;
    }

private:
    double radius;
};

int main() {
    Rectangle rect(5.0, 3.0);
    Circle circle(2.5);

    rect.printArea();   // Calls Rectangle's calculateArea()
    circle.printArea(); // Calls Circle's calculateArea()

    return 0;
}
```

In this example:

Shape is an abstract base class that contains a pure virtual function calculateArea(). An abstract class cannot be instantiated, and any class that derives from it must provide an implementation for calculateArea().

Rectangle and Circle are derived classes that inherit from Shape. They provide concrete implementations for the pure virtual function calculateArea().

The printArea() member function of the Shape class is not pure virtual and has a default implementation that calls calculateArea(). Derived classes inherit this function and can override it if needed.

In the main function, we create objects of Rectangle and Circle and call the printArea() function on them, which in turn calls the appropriate implementation of calculateArea() based on the object's actual type. This demonstrates polymorphism and the use of an abstract base class.