C++ Programming Guide

Introduction to C++

What is C++ and its features

C++ is a powerful, general-purpose programming language created by Bjarne Stroustrup as an extension of the C language. It is a multi-paradigm language, supporting procedural, object-oriented, and generic programming. C++ is widely used in system programming, game development, high-performance computing, and embedded systems due to its efficiency, control over hardware, and extensive libraries.

Key features of C++ include:

- **Object-Oriented Programming (OOP):** C++ supports classes, objects, inheritance, polymorphism, abstraction, and encapsulation, which are fundamental concepts of OOP. This allows for modular, reusable, and maintainable code.
- **Performance:** C++ provides low-level memory management capabilities, allowing developers to optimize code for maximum performance. It compiles directly to machine code, resulting in fast execution speeds.
- **Portability:** C++ programs can be compiled and run on various platforms (Windows, Linux, macOS) with minimal changes.
- **Compatibility with C:** C++ is almost a superset of C, meaning most valid C programs are also valid C++ programs. This allows for seamless integration with existing C code and libraries.
- **Standard Library:** C++ comes with a rich standard library, including the Standard Template Library (STL), which provides a collection of powerful and efficient algorithms, containers, and iterators.
- **Memory Management:** C++ offers both automatic and manual memory management. Developers can use `new` and `delete` operators for dynamic

memory allocation and deallocation, or rely on smart pointers for automatic memory management.

C++ vs C vs other languages

C++ builds upon the foundations of C, adding object-oriented features and other enhancements. While C is a procedural language primarily focused on low-level memory access and system programming, C++ introduces classes and objects, enabling a more structured and modular approach to software development. Compared to higher-level languages like Python or Java, C++ offers greater control over system resources and often achieves superior performance, albeit with a steeper learning curve and more manual memory management.

Compilation process

The compilation process of a C++ program typically involves several stages:

1. **Preprocessing:** The preprocessor handles directives like `#include` and `#define`. It expands macros, includes header files, and removes comments, generating an expanded source file.
2. **Compilation:** The compiler translates the preprocessed code into assembly language, which is a low-level language specific to the target machine's architecture.
3. **Assembly:** The assembler converts the assembly code into machine code (object code), which is a binary representation of the program. Object files usually have a `.o` or `.obj` extension.
4. **Linking:** The linker combines the object files with necessary libraries (e.g., standard library functions) to produce a single executable file. This executable file can then be run on the target system.

```cpp
// Example: Basic compilation process
// Save this as main.cpp
#include <iostream>

int main() {
    std::cout << "Hello, C++!" << std::endl;
    return 0;
}
```

To compile this program using a common C++ compiler like g++:

```
g++ main.cpp -o my_program
./my_program
```

Basic program structure

A typical C++ program consists of several fundamental components:

- **Header files:** These files contain declarations of functions, classes, and variables that are used in the program. They are included using the `#include` directive.
- **`main` function:** This is the entry point of every C++ program. Execution begins from `main`.
- **Statements:** Instructions that perform actions, terminated by a semicolon.
- **Comments:** Used to explain code and improve readability.

```cpp
// Example: Basic program structure

#include <iostream> // Include the iostream header for input/output operations

// The main function where program execution begins
int main() {
    // This is a single-line comment
    /*
     * This is a multi-line comment.
     * It can span multiple lines.
     */

    std::cout << "Hello, World!" << std::endl; // Output "Hello, World!" to the
console

    return 0; // Indicate successful program execution
}
```

Comments and documentation

Comments are essential for making code understandable. C++ supports two types of comments:

- **Single-line comments:** Start with `//` and continue to the end of the line.
- **Multi-line comments:** Start with `/*` and end with `*/`. They can span multiple lines.

Good documentation practices involve using comments to explain complex logic, function purposes, and variable usage.

Variables and data types

Variables are named storage locations that hold data. Each variable has a specific data type, which determines the kind of data it can store and the operations that can be performed on it.

```cpp
// Example: Variables and basic data types

#include <iostream>
#include <string>

int main() {
    // Integer type
    int age = 30;
    std::cout << "Age: " << age << std::endl;

    // Floating-point type
    double price = 19.99;
    std::cout << "Price: " << price << std::endl;

    // Character type
    char initial = 'J';
    std::cout << "Initial: " << initial << std::endl;

    // Boolean type
    bool is_student = true;
    std::cout << "Is student: " << is_student << std::endl;

    // String type (from C++ Standard Library)
    std::string name = "Alice";
    std::cout << "Name: " << name << std::endl;

    return 0;
}
```

Input/output operations

C++ uses streams for input and output. The `iostream` library provides `std::cin` for input and `std::cout` for output.

```cpp
// Example: Input/output operations

#include <iostream>
#include <string>

int main() {
    std::string user_name;
    int user_age;

    // Output a prompt to the user
    std::cout << "Enter your name: ";
    // Read input from the user
    std::cin >> user_name;

    std::cout << "Enter your age: ";
    std::cin >> user_age;

    // Output the collected information
    std::cout << "Hello, " << user_name << "! You are " << user_age << " years
old." << std::endl;

    return 0;
}
```

Variables and Data Types

Variables are fundamental to programming, serving as named storage locations for data. Each variable is associated with a data type, which dictates the kind of values it can hold and the operations that can be performed on it.

Primitive data types (int, float, double, char, bool)

C++ provides a set of built-in, fundamental data types to represent various kinds of information:

- **`int`**: Used to store whole numbers (integers) without decimal points. Its size typically depends on the system architecture (e.g., 2 or 4 bytes).
- **`float`**: Used to store single-precision floating-point numbers (numbers with decimal points). It typically occupies 4 bytes.
- **`double`**: Used to store double-precision floating-point numbers, offering greater precision and range than `float`. It typically occupies 8 bytes.
- **`char`**: Used to store single characters (e.g., 'A', 'b', '5'). It typically occupies 1 byte.
- **`bool`**: Used to store boolean values, which can only be `true` or `false`. It typically occupies 1 byte.

```cpp
// Example: Primitive data types

#include <iostream>

int main() {
    int integer_var = 100;
    float float_var = 10.5f; // 'f' suffix denotes a float literal
    double double_var = 20.75;
    char char_var = 'Z';
    bool bool_var = true;

    std::cout << "Integer: " << integer_var << std::endl;
    std::cout << "Float: " << float_var << std::endl;
    std::cout << "Double: " << double_var << std::endl;
    std::cout << "Char: " << char_var << std::endl;
    std::cout << "Bool: " << bool_var << std::endl;

    return 0;
}
```

Type modifiers (signed, unsigned, long, short)

Type modifiers can be applied to primitive data types to alter their storage size or range of values:

- **`signed`**: (Default for integer types) Indicates that the variable can hold both positive and negative values.
- **`unsigned`**: Indicates that the variable can only hold non-negative (zero or positive) values, effectively doubling the positive range.
- **`long`**: Increases the size of integer or double types. Can be `long int` or `long long int` (for even larger integers), and `long double`.
- **`short`**: Decreases the size of an integer type. Can be `short int`.

```cpp
// Example: Type modifiers

#include <iostream>

int main() {
    unsigned int positive_number = 4000000000; // Can hold larger positive
values
    long long big_number = 123456789012345LL; // 'LL' suffix for long long
literal
    short int small_number = -30000;

    std::cout << "Unsigned int: " << positive_number << std::endl;
    std::cout << "Long long: " << big_number << std::endl;
    std::cout << "Short int: " << small_number << std::endl;

    // Size of data types (can vary by system)
    std::cout << "Size of int: " << sizeof(int) << " bytes" << std::endl;
    std::cout << "Size of unsigned int: " << sizeof(unsigned int) << " bytes"
<< std::endl;
    std::cout << "Size of long long: " << sizeof(long long) << " bytes" <<
std::endl;
    std::cout << "Size of short int: " << sizeof(short int) << " bytes" <<
std::endl;

    return 0;
}
```

Variable declaration and initialization

Declaration introduces a variable's name and type to the compiler. **Initialization** assigns an initial value to the variable.

```cpp
// Example: Variable declaration and initialization

#include <iostream>

int main() {
    // Declaration
    int declared_variable;

    // Initialization (assignment after declaration)
    declared_variable = 50;
    std::cout << "Declared and assigned: " << declared_variable << std::endl;

    // Declaration and initialization in one step
    double initialized_variable = 99.99;
    std::cout << "Initialized at declaration: " << initialized_variable <<
std::endl;

    // Uniform initialization (C++11 and later)
    int uniform_init {200};
    std::cout << "Uniform initialized: " << uniform_init << std::endl;

    return 0;
}
```

Constants (const, #define)

Constants are values that cannot be changed during program execution.

- **`const` keyword**: A type-safe way to define constants. The compiler enforces that `const` variables are not modified.
- **`#define` preprocessor directive**: An older, less type-safe method to define symbolic constants. The preprocessor replaces all occurrences of the macro name with its value before compilation.

```
// Example: Constants

#include <iostream>

// Using #define (preprocessor constant)
#define PI 3.14159

int main() {
    // Using const (compile-time constant)
    const int MAX_ATTEMPTS = 3;
    const double GRAVITY = 9.81;

    std::cout << "Max Attempts: " << MAX_ATTEMPTS << std::endl;
    std::cout << "Gravity: " << GRAVITY << std::endl;
    std::cout << "PI (from #define): " << PI << std::endl;

    // MAX_ATTEMPTS = 4; // This would cause a compile-time error

    return 0;
}
```

Type conversion (implicit and explicit)

Type conversion, or type casting, is the process of converting a value from one data type to another.

- **Implicit conversion (coercion)**: Automatically performed by the compiler when types are compatible (e.g., `int` to `double`). This is generally safe.
- **Explicit conversion (type casting)**: Performed by the programmer using cast operators. This is necessary when implicit conversion is not possible or to avoid data loss.

```cpp
// Example: Type conversion

#include <iostream>

int main() {
    int int_val = 10;
    double double_val = 3.14;

    // Implicit conversion: int to double
    double result_implicit = int_val + double_val;
    std::cout << "Implicit conversion (int to double): " << result_implicit <<
std::endl; // 10.0 + 3.14 = 13.14

    // Explicit conversion: double to int (C-style cast)
    int result_explicit_c = (int)double_val;
    std::cout << "Explicit conversion (double to int, C-style): " <<
result_explicit_c << std::endl; // 3 (data loss)

    // Explicit conversion: double to int (C++ static_cast)
    int result_explicit_cpp = static_cast<int>(double_val);
    std::cout << "Explicit conversion (double to int, static_cast): " <<
result_explicit_cpp << std::endl; // 3

    // Implicit conversion: char to int (ASCII value)
    char char_grade = 'A';
    int int_grade = char_grade;
    std::cout << "Implicit conversion (char to int): " << int_grade <<
std::endl; // ASCII value of 'A' (e.g., 65)

    return 0;
}
```

Scope and lifetime of variables

Scope refers to the region of a program where a variable can be accessed. **Lifetime** refers to the period during which a variable exists in memory.

- **Local scope (Block scope)**: Variables declared inside a block (e.g., within a function or a loop) are local to that block and cannot be accessed outside it. Their lifetime begins when the block is entered and ends when the block is exited.
- **Global scope (File scope)**: Variables declared outside any function or block have global scope and can be accessed from anywhere in the file after their declaration. Their lifetime is the entire duration of the program.

```cpp
// Example: Scope and lifetime of variables

#include <iostream>

int global_var = 100; // Global variable

void myFunction() {
    int function_local_var = 20; // Local to myFunction
    std::cout << "Inside myFunction: global_var = " << global_var << ",
function_local_var = " << function_local_var << std::endl;
}

int main() {
    int main_local_var = 5; // Local to main

    std::cout << "Inside main: global_var = " << global_var << ",
main_local_var = " << main_local_var << std::endl;

    myFunction();

    // std::cout << function_local_var; // This would cause a compile-time
error (out of scope)

    {
        // This is a new block scope
        int block_local_var = 30;
        std::cout << "Inside block: block_local_var = " << block_local_var <<
std::endl;
    } // block_local_var is destroyed here

    // std::cout << block_local_var; // This would cause a compile-time error
(out of scope)

    return 0;
}
```

Operators

Operators are special symbols that perform operations on one or more operands (values or variables). C++ provides a rich set of operators for various purposes.

Arithmetic operators (+, -, *, /, %)

These operators perform basic mathematical calculations:

- `+` (Addition)
- `-` (Subtraction)
- `*` (Multiplication)

- `/` (Division): Integer division truncates the decimal part if both operands are integers.
- `%` (Modulo): Returns the remainder of an integer division.

```cpp
// Example: Arithmetic operators

#include <iostream>

int main() {
    int a = 10, b = 3;

    std::cout << "a + b = " << (a + b) << std::endl; // 13
    std::cout << "a - b = " << (a - b) << std::endl; // 7
    std::cout << "a * b = " << (a * b) << std::endl; // 30
    std::cout << "a / b = " << (a / b) << std::endl; // 3 (integer division)
    std::cout << "a % b = " << (a % b) << std::endl; // 1

    double x = 10.0, y = 3.0;
    std::cout << "x / y = " << (x / y) << std::endl; // 3.333...

    return 0;
}
```

Relational operators (==, !=, <, >, <=, >=)

These operators compare two operands and return a boolean (`true` or `false`) result:

- `==` (Equal to)
- `!=` (Not equal to)
- `<` (Less than)
- `>` (Greater than)
- `<=` (Less than or equal to)
- `>=` (Greater than or equal to)

```cpp
// Example: Relational operators

#include <iostream>

int main() {
    int p = 5, q = 10;

    std::cout << "p == q: " << (p == q) << std::endl; // 0 (false)
    std::cout << "p != q: " << (p != q) << std::endl; // 1 (true)
    std::cout << "p < q: " << (p < q) << std::endl;   // 1 (true)
    std::cout << "p > q: " << (p > q) << std::endl;   // 0 (false)
    std::cout << "p <= q: " << (p <= q) << std::endl; // 1 (true)
    std::cout << "p >= q: " << (p >= q) << std::endl; // 0 (false)

    return 0;
}
```

Logical operators (&&, ||, !)

These operators combine or modify boolean expressions:

- `&&` (Logical AND): Returns `true` if both operands are `true`.
- `||` (Logical OR): Returns `true` if at least one operand is `true`.
- `!` (Logical NOT): Reverses the boolean value of its operand.

```cpp
// Example: Logical operators

#include <iostream>

int main() {
    bool is_sunny = true;
    bool is_warm = false;

    std::cout << "is_sunny && is_warm: " << (is_sunny && is_warm) << std::endl;
// 0 (false)
    std::cout << "is_sunny || is_warm: " << (is_sunny || is_warm) << std::endl;
// 1 (true)
    std::cout << "!is_sunny: " << (!is_sunny) << std::endl;
// 0 (false)

    return 0;
}
```

Bitwise operators (&, |, ^, ~)

These operators perform operations on individual bits of integer operands:

- `&` (Bitwise AND)

- `|` (Bitwise OR)
- `^` (Bitwise XOR)
- `~` (Bitwise NOT/Complement)
- `<<` (Left Shift)
- `>>` (Right Shift)

```
// Example: Bitwise operators

#include <iostream>

int main() {
    int num1 = 5;  // Binary: 0101
    int num2 = 12; // Binary: 1100

    std::cout << "num1 & num2: " << (num1 & num2) << std::endl; // 0100 (4)
    std::cout << "num1 | num2: " << (num1 | num2) << std::endl; // 1101 (13)
    std::cout << "num1 ^ num2: " << (num1 ^ num2) << '
'; // 1101 (13) -> 1001 (9)
    std::cout << "~num1: " << (~num1) << std::endl;         // ...11111010 (-6
for 2's complement)
    std::cout << "num1 << 1: " << (num1 << 1) << std::endl; // 1010 (10)
    std::cout << "num2 >> 2: " << (num2 >> 2) << std::endl; // 0011 (3)

    return 0;
}
```

Assignment operators (=, +=, -=, *=, /=, %=)

These operators assign a value to a variable. Compound assignment operators perform an operation and then assign the result:

- `=` (Simple assignment)
- `+=` (Add and assign)
- `-=` (Subtract and assign)
- `*=` (Multiply and assign)
- `/=` (Divide and assign)
- `%=` (Modulo and assign)

```cpp
// Example: Assignment operators

#include <iostream>

int main() {
    int x = 10;

    x += 5; // x = x + 5; (x is now 15)
    std::cout << "x after += 5: " << x << std::endl;

    x -= 3; // x = x - 3; (x is now 12)
    std::cout << "x after -= 3: " << x << std::endl;

    x *= 2; // x = x * 2; (x is now 24)
    std::cout << "x after *= 2: " << x << std::endl;

    x /= 4; // x = x / 4; (x is now 6)
    std::cout << "x after /= 4: " << x << std::endl;

    x %= 5; // x = x % 5; (x is now 1)
    std::cout << "x after %= 5: " << x << std_endl;

    return 0;
}
```

Ternary operator (?:)

The ternary operator is a shorthand for an `if-else` statement. It takes three operands:

`condition ? expression_if_true : expression_if_false;`

```cpp
// Example: Ternary operator

#include <iostream>

int main() {
    int age = 20;
    std::string status = (age >= 18) ? "Adult" : "Minor";
    std::cout << "Status: " << status << std::endl;

    int num = 7;
    std::string parity = (num % 2 == 0) ? "Even" : "Odd";
    std::cout << "Parity of " << num << ": " << parity << std::endl;

    return 0;
}
```

Operator precedence

Operator precedence determines the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before operators with

lower precedence. Parentheses `()` can be used to override precedence.

For example, multiplication and division have higher precedence than addition and subtraction.

```cpp
// Example: Operator precedence

#include <iostream>

int main() {
    int result1 = 5 + 3 * 2;     // Evaluates as 5 + (3 * 2) = 5 + 6 = 11
    int result2 = (5 + 3) * 2;   // Evaluates as (8) * 2 = 16

    std::cout << "Result 1: " << result1 << std::endl;
    std::cout << "Result 2: " << result2 << std::endl;

    return 0;
}
```

Functions

Functions are blocks of code that perform a specific task. They help in organizing code, making it more modular, reusable, and easier to understand.

Function declaration and definition

- **Declaration (Prototype):** Informs the compiler about the function's name, return type, and parameters. It's typically placed in a header file or before `main()`.
- **Definition:** Contains the actual code (body) of the function.

```cpp
// Example: Function declaration and definition

#include <iostream>

// Function Declaration (Prototype)
int add(int a, int b);

int main() {
    int sum = add(5, 3);
    std::cout << "Sum: " << sum << std::endl;
    return 0;
}

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

Parameters and arguments

- **Parameters:** Variables listed in the function declaration/definition that receive values from the caller.
- **Arguments:** The actual values passed to the function when it is called.

```cpp
// Example: Parameters and arguments

#include <iostream>

void greet(std::string name, int age) { // name and age are parameters
    std::cout << "Hello, " << name << "! You are " << age << " years old." <<
std::endl;
}

int main() {
    greet("Alice", 30); // "Alice" and 30 are arguments
    return 0;
}
```

Return types and values

- **Return Type:** Specifies the type of value a function will send back to the caller. `void` means the function does not return any value.
- **Return Value:** The actual value sent back by the function using the `return` keyword.

```cpp
// Example: Return types and values

#include <iostream>

int multiply(int x, int y) { // int is the return type
    return x * y; // Returns an int value
}

void displayMessage() { // void return type, no value returned
    std::cout << "This function displays a message." << std::endl;
}

int main() {
    int product = multiply(4, 5);
    std::cout << "Product: " << product << std::endl;
    displayMessage();
    return 0;
}
```

Function overloading

Function overloading allows multiple functions to have the same name but different parameter lists (different number or types of parameters). The compiler determines which function to call based on the arguments provided.

```cpp
// Example: Function overloading

#include <iostream>
#include <string>

int operate(int a, int b) {
    return a + b;
}

double operate(double a, double b) {
    return a * b;
}

std::string operate(std::string s1, std::string s2) {
    return s1 + " " + s2;
}

int main() {
    std::cout << "Sum of ints: " << operate(10, 20) << std::endl; // Calls int
version
    std::cout << "Product of doubles: " << operate(2.5, 3.0) << std::endl; //
Calls double version
    std::cout << "Concatenated strings: " << operate("Hello", "World") <<
std::endl; // Calls string version
    return 0;
}
```

Default arguments

Default arguments allow a function parameter to have a default value if no argument is provided for that parameter during a function call. Default arguments must be specified from right to left in the parameter list.

```cpp
// Example: Default arguments

#include <iostream>
#include <string>

void printInfo(std::string name, int age = 25) {
    std::cout << "Name: " << name << ", Age: " << age << std::endl;
}

int main() {
    printInfo("Bob");         // Uses default age (25)
    printInfo("Charlie", 30); // Overrides default age
    return 0;
}
```

Pass by value vs pass by reference

- **Pass by Value:** A copy of the argument's value is passed to the function. Changes made to the parameter inside the function do not affect the original argument.
- **Pass by Reference:** The memory address of the argument is passed to the function. Changes made to the parameter inside the function *do* affect the original argument.

```cpp
// Example: Pass by value vs pass by reference

#include <iostream>

void passByValue(int x) {
    x = 100; // Changes only the copy
}

void passByReference(int &y) { // y is a reference to the original variable
    y = 200; // Changes the original variable
}

int main() {
    int a = 10;
    int b = 20;

    std::cout << "Before passByValue: a = " << a << std::endl;
    passByValue(a);
    std::cout << "After passByValue: a = " << a << std::endl; // a is still 10

    std::cout << "Before passByReference: b = " << b << std::endl;
    passByReference(b);
    std::cout << "After passByReference: b = " << b << std::endl; // b is now
200

    return 0;
}
```

Inline functions

An `inline` function is a hint to the compiler that it should try to expand the function's code directly at the point of call, rather than performing a regular function call. This can sometimes improve performance by avoiding function call overhead, but it's a suggestion, not a command, and the compiler can ignore it.

```cpp
// Example: Inline functions

#include <iostream>

inline int square(int n) {
    return n * n;
}

int main() {
    int result = square(7);
    std::cout << "Square of 7: " << result << std::endl;
    return 0;
}
```

Recursion

Recursion is a programming technique where a function calls itself to solve a problem. A recursive function must have a base case (a condition that stops the recursion) to prevent infinite loops.

```cpp
// Example: Recursion (Factorial calculation)

#include <iostream>

long long factorial(int n) {
    // Base case
    if (n == 0 || n == 1) {
        return 1;
    }
    // Recursive step
    return n * factorial(n - 1);
}

int main() {
    int num = 5;
    std::cout << "Factorial of " << num << ": " << factorial(num) << std::endl;
// 5! = 120
    return 0;
}
```

Arrays and Strings

Arrays and strings are fundamental data structures used to store collections of data. Arrays store elements of the same data type in contiguous memory locations, while strings are sequences of characters.

Array declaration and initialization

An array is a collection of elements of the same data type, stored in contiguous memory locations. Elements are accessed using an index, starting from 0.

```cpp
// Example: Array declaration and initialization

#include <iostream>

int main() {
    // Declaration without initialization (values are garbage)
    int numbers[5];

    // Declaration and initialization
    int scores[3] = {100, 90, 80};

    // Accessing elements
    std::cout << "First score: " << scores[0] << std::endl; // 100

    // Modifying elements
    scores[1] = 95;
    std::cout << "Modified second score: " << scores[1] << std::endl; // 95

    // Omitting size (compiler calculates it)
    int grades[] = {85, 70, 92, 65};
    std::cout << "Size of grades array: " << sizeof(grades) / sizeof(grades[0])
<< std::endl;

    // Initializing all elements to 0 (or default for other types)
    int zeros[5] = {}; // All elements are 0
    std::cout << "First zero: " << zeros[0] << std::endl;

    return 0;
}
```

Multi-dimensional arrays

Multi-dimensional arrays are arrays of arrays, commonly used to represent tables or matrices. A 2D array is often called a matrix.

```cpp
// Example: Multi-dimensional arrays

#include <iostream>

int main() {
    // Declare and initialize a 2x3 matrix
    int matrix[2][3] = {
        {1, 2, 3},
        {4, 5, 6}
    };

    // Accessing elements
    std::cout << "Element at [0][1]: " << matrix[0][1] << std::endl; // 2

    // Iterating through a 2D array
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << std::endl;
    }

    return 0;
}
```

C-style strings

C-style strings are character arrays terminated by a null character (`\0`). They are inherited from the C language.

```cpp
// Example: C-style strings

#include <iostream>
#include <cstring> // For strlen, strcpy, etc.

int main() {
    char greeting[20] = "Hello";
    char name[] = "World"; // Size is automatically determined

    std::cout << "Greeting: " << greeting << std::endl;
    std::cout << "Name: " << name << std::endl;

    // Concatenate strings
    strcat(greeting, ", ");
    strcat(greeting, name);
    std::cout << "Concatenated: " << greeting << std::endl;

    // Length of string
    std::cout << "Length of greeting: " << strlen(greeting) << std::endl;

    // Copy string
    char copy[20];
    strcpy(copy, greeting);
    std::cout << "Copied string: " << copy << std::endl;

    return 0;
}
```

std::string class

The `std::string` class (part of the C++ Standard Library) provides a more robust, flexible, and safer way to handle strings compared to C-style strings. It automatically manages memory and offers many useful member functions.

```cpp
// Example: std::string class

#include <iostream>
#include <string>

int main() {
    std::string s1 = "Hello";
    std::string s2 = "World";

    // Concatenation
    std::string s3 = s1 + " " + s2;
    std::cout << "Concatenated: " << s3 << std::endl;

    // Length
    std::cout << "Length of s3: " << s3.length() << std::endl;

    // Substring
    std::cout << "Substring (from index 6, length 5): " << s3.substr(6, 5) <<
std::endl; // World

    // Finding characters/substrings
    size_t pos = s3.find("World");
    if (pos != std::string::npos) {
        std::cout << "'World' found at position: " << pos << std::endl;
    }

    // Comparison
    if (s1 == "Hello") {
        std::cout << "s1 is 'Hello'" << std::endl;
    }

    return 0;
}
```

Array manipulation

Manipulating arrays involves operations like adding, removing, or reordering elements. For dynamic manipulation, `std::vector` is often preferred over raw arrays.

```cpp
// Example: Array manipulation (using std::vector for dynamic size)

#include <iostream>
#include <vector>
#include <algorithm> // For std::sort, std::reverse

int main() {
    std::vector<int> vec = {5, 2, 8, 1, 9};

    std::cout << "Original vector: ";
    for (int x : vec) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    // Add element
    vec.push_back(7);
    std::cout << "After push_back(7): ";
    for (int x : vec) {
        std_cout << x <<
```

Pointers and References

Pointers and references are fundamental concepts in C++ that allow for indirect access to memory locations. They are crucial for efficient memory management, working with data structures, and implementing advanced programming techniques.

Pointer declaration and initialization

A **pointer** is a variable that stores the memory address of another variable. It "points" to a location in memory. The `*` symbol is used to declare a pointer, and the `&` operator is used to get the address of a variable.

```cpp
// Example: Pointer declaration and initialization

#include <iostream>

int main() {
    int var = 10;
    int* ptr; // Declare a pointer to an integer

    ptr = &var; // Initialize ptr with the address of var

    std::cout << "Value of var: " << var << std::endl;       // Output: 10
    std::cout << "Address of var: " << &var << std::endl;     // Output: Memory address of var
    std::cout << "Value of ptr (address of var): " << ptr << std::endl; // Output: Same memory address
    std::cout << "Value pointed to by ptr: " << *ptr << std::endl; // Output: 10 (dereferencing ptr)

    *ptr = 20; // Change the value of var through the pointer
    std::cout << "New value of var: " << var << std::endl; // Output: 20

    // Null pointer initialization
    int* null_ptr = nullptr; // Recommended way to initialize null pointers in C++11 and later
    // int* old_null_ptr = 0; // Older C-style null pointer
```

```
    // int* another_null_ptr = NULL; // Also common, but nullptr is preferred

    if (null_ptr == nullptr) {
        std::cout << "null_ptr is a null pointer." << std::endl;
    }

    return 0;
}
```

Pointer arithmetic

Pointers can be manipulated using arithmetic operations (addition, subtraction, increment, decrement). When performing arithmetic on pointers, the operation is scaled by the size of the data type the pointer points to.

```
// Example: Pointer arithmetic

#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    int* ptr = arr; // ptr points to the first element (arr[0])

    std::cout << "Value at ptr: " << *ptr << std::endl; // 10

    ptr++; // Move ptr to the next integer (arr[1])
    std::cout << "Value at ptr after ptr++: " << *ptr << std::endl; // 20

    ptr += 2; // Move ptr two integers forward (arr[3])
    std::cout << "Value at ptr after ptr += 2: " << *ptr << std::endl; // 40

    ptr--; // Move ptr back one integer (arr[2])
    std::cout << "Value at ptr after ptr--: " << *ptr << std::endl; // 30

    // Subtracting pointers gives the number of elements between them
    int* ptr_end = &arr[4]; // ptr_end points to arr[4]
    std::cout << "Elements between ptr and ptr_end: " << (ptr_end - ptr) <<
std::endl; // 2 (arr[3] and arr[4])

    return 0;
}
```

Pointers and arrays

In C++, array names often decay into pointers to their first element. This close relationship allows pointers to be used interchangeably with array indexing in many contexts.

```cpp
// Example: Pointers and arrays

#include <iostream>

int main() {
    int arr[] = {10, 20, 30, 40, 50};

    // Array name as a pointer
    int* p = arr; // p points to arr[0]

    std::cout << "Using pointer p[0]: " << p[0] << std::endl;     // 10
    std::cout << "Using pointer *(p+1): " << *(p + 1) << std::endl; // 20

    // Iterating through array using pointer arithmetic
    for (int i = 0; i < 5; ++i) {
        std::cout << *(arr + i) << " "; // Equivalent to arr[i]
    }
    std::cout << std::endl;

    // Function taking an array (actually takes a pointer)
    void printArray(int* arr_ptr, int size);
    printArray(arr, 5);

    return 0;
}

void printArray(int* arr_ptr, int size) {
    std::cout << "Printing array via pointer in function: ";
    for (int i = 0; i < size; ++i) {
        std::cout << arr_ptr[i] << " ";
    }
    std::cout << std::endl;
}
```

References vs pointers

References are aliases (alternative names) for existing variables. Once initialized, a reference cannot be reseated to refer to another variable. They must be initialized at declaration. **Pointers** can be null, can be reassigned, and can perform arithmetic.

Feature	Pointer	Reference
Initialization	Can be declared without initialization	Must be initialized at declaration
Reassignment	Can be reassigned to point to another variable	Cannot be reseated (always refers to the same object)
Null Value	Can be `nullptr` (or `NULL` / `0`)	Cannot be null (must refer to an object)
Arithmetic	Supports pointer arithmetic	Does not support arithmetic
Dereference	Requires `*` to access value	No dereference operator needed (`.` or `->` for objects)

```cpp
// Example: References vs pointers

#include <iostream>

int main() {
    int value = 10;

    // Pointer
    int* ptr = &value; // ptr stores the address of value
    std::cout << "Pointer value: " << *ptr << std::endl; // Dereference to get value
    *ptr = 20;
    std::cout << "Value after pointer change: " << value << std::endl;

    // Reference
    int& ref = value; // ref is an alias for value
    std::cout << "Reference value: " << ref << std::endl; // Access directly
    ref = 30;
    std::cout << "Value after reference change: " << value << std::endl;

    // ptr = &another_value; // Valid: pointer can be reassigned
    // int another_value = 50;
    // ref = another_value; // Invalid: reference cannot be reseated (this would assign 50 to 'value')

    return 0;
}
```

Dynamic memory allocation (new, delete)

Dynamic memory allocation allows programs to request memory at runtime from the heap (free store). `new` is used to allocate memory, and `delete` is used to deallocate it, preventing memory leaks.

```cpp
// Example: Dynamic memory allocation

#include <iostream>

int main() {
    // Allocate single integer
    int* dynamic_int = new int; // Allocate memory for one integer
    *dynamic_int = 100;
    std::cout << "Dynamically allocated int: " << *dynamic_int << std::endl;
    delete dynamic_int; // Deallocate memory
    dynamic_int = nullptr; // Good practice to set to nullptr after delete

    // Allocate an array of integers
    int* dynamic_array = new int[5]; // Allocate memory for 5 integers
    for (int i = 0; i < 5; ++i) {
        dynamic_array[i] = (i + 1) * 10;
    }

    std::cout << "Dynamically allocated array: ";
    for (int i = 0; i < 5; ++i) {
        std::cout << dynamic_array[i] << " ";
    }
    std::cout << std::endl;

    delete[] dynamic_array; // Deallocate array memory
    dynamic_array = nullptr;

    return 0;
}
```

Smart pointers (unique_ptr, shared_ptr)

Smart pointers are objects that behave like pointers but provide automatic memory management, helping to prevent memory leaks and dangling pointers. They are part of the C++ Standard Library (`<memory>` header).

- **`std::unique_ptr`:** Owns the object it points to exclusively. When the `unique_ptr` goes out of scope, the object it manages is automatically deleted. Cannot be copied, only moved.
- **`std::shared_ptr`:** Allows multiple `shared_ptr` objects to own the same object. It uses a reference count to keep track of how many `shared_ptr`s point to the object. The object is deleted when the last `shared_ptr` pointing to it is destroyed.

```cpp
// Example: Smart pointers

#include <iostream>
#include <memory> // For unique_ptr and shared_ptr

class MyClass {
public:
    MyClass() { std::cout << "MyClass Constructor!" << std::endl; }
    ~MyClass() { std::cout << "MyClass Destructor!" << std::endl; }
    void greet() { std::cout << "Hello from MyClass!" << std::endl; }
};

void uniquePtrExample() {
    std::unique_ptr<MyClass> u_ptr(new MyClass()); // C++11 way
    // Or better (C++14 onwards):
    // std::unique_ptr<MyClass> u_ptr = std::make_unique<MyClass>();
    u_ptr->greet();
    // MyClass object is automatically deleted when u_ptr goes out of scope
}

void sharedPtrExample() {
    std::shared_ptr<MyClass> s_ptr1 = std::make_shared<MyClass>();
    s_ptr1->greet();
    std::cout << "Shared count: " << s_ptr1.use_count() << std::endl; // 1

    std::shared_ptr<MyClass> s_ptr2 = s_ptr1; // s_ptr2 now shares ownership
    std::cout << "Shared count: " << s_ptr1.use_count() << std::endl; // 2

    // MyClass object is deleted when both s_ptr1 and s_ptr2 go out of scope
}

int main() {
    std::cout << "--- Unique Pointer Example ---" << std::endl;
    uniquePtrExample();
    std::cout << "--- Shared Pointer Example ---" << std::endl;
    sharedPtrExample();
    return 0;
}
```

Pointer pitfalls and best practices

Working with pointers requires careful attention to avoid common errors:

- **Dangling Pointers:** Pointers that point to memory that has been deallocated. Accessing such memory leads to undefined behavior.
- **Memory Leaks:** Occur when dynamically allocated memory is no longer accessible (e.g., pointer goes out of scope or is reassigned) but has not been deallocated with `delete`.
- **Wild Pointers:** Uninitialized pointers that contain garbage values. Dereferencing them can lead to crashes.

Best Practices:

- **Initialize Pointers:** Always initialize pointers to `nullptr` (or `NULL` / `0`) if they don't point to valid memory immediately.
- **`delete` What You `new`:** For every `new` allocation, ensure there's a corresponding `delete` (or `delete[]` for arrays).
- **Use Smart Pointers:** Prefer `std::unique_ptr` and `std::shared_ptr` for dynamic memory management. They automate deallocation and significantly reduce the risk of memory leaks and dangling pointers.
- **Avoid `goto` with Pointers:** Using `goto` can make it difficult to track memory allocation and deallocation, increasing the risk of leaks.
- **Check for `nullptr`:** Before dereferencing a pointer, especially one that might be null, check if it's `nullptr` to prevent crashes.

```cpp
// Example: Pointer pitfalls and best practices

#include <iostream>
#include <memory>

void bad_example() {
    int* ptr = new int; // Allocate memory
    // ... some operations ...
    // Forgetting to delete leads to memory leak
    // delete ptr;
}

void dangling_pointer_example() {
    int* ptr = new int;
    delete ptr; // Memory deallocated
    // ptr is now a dangling pointer, still holds the old address
    // *ptr = 10; // Dereferencing now is undefined behavior
}

void good_example_raw_pointer() {
    int* ptr = nullptr; // Initialize to nullptr
    ptr = new int;      // Allocate
    *ptr = 10;
    std::cout << "Good raw pointer example: " << *ptr << std::endl;
    delete ptr;         // Deallocate
    ptr = nullptr;      // Set to nullptr after delete
}

void good_example_smart_pointer() {
    std::unique_ptr<int> u_ptr = std::make_unique<int>(20); // Smart pointer
    std::cout << "Good smart pointer example: " << *u_ptr << std::endl;
    // No need to call delete, memory is managed automatically
}

int main() {
    bad_example(); // Potential memory leak
    dangling_pointer_example(); // Potential undefined behavior

    good_example_raw_pointer();
    good_example_smart_pointer();

    return 0;
}
```

Foundational Object-Oriented Programming Concepts

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (attributes or properties), and code in the form of procedures (methods).

Classes and objects

- **Class:** A blueprint or a template for creating objects. It defines the properties (data members) and behaviors (member functions) that objects of that class will have.
- **Object:** An instance of a class. When a class is defined, no memory is allocated until an object of that class is created.

```
// Example: Classes and objects

#include <iostream>
#include <string>

// Define a class named 'Car'
class Car {
public: // Access specifier
    // Data members (attributes)
    std::string brand;
    std::string model;
    int year;

    // Member function (behavior)
    void displayInfo() {
        std::cout << "Brand: " << brand << ", Model: " << model << ", Year: "
<< year << std::endl;
    }
};

int main() {
    // Create objects (instances) of the Car class
    Car car1; // Object 1
    car1.brand = "Toyota";
    car1.model = "Camry";
    car1.year = 2020;

    Car car2; // Object 2
    car2.brand = "Honda";

    car2.model = "Civic";
    car2.year = 2021;

    // Call member functions
    car1.displayInfo();
    car2.displayInfo();

    return 0;
}
```

Constructors and destructors

- **Constructor:** A special member function that is automatically called when an object of a class is created. It is used to initialize the object's data members.

Constructors have the same name as the class and do not have a return type.

    - **Default Constructor:** A constructor that takes no arguments.
    - **Parameterized Constructor:** A constructor that takes arguments to initialize the object with specific values.
    - **Copy Constructor:** A constructor that creates a new object as a copy of an existing object.

- **Destructor:** A special member function that is automatically called when an object is destroyed (goes out of scope or is explicitly deleted). It is used to release resources (like dynamically allocated memory) acquired by the object during its lifetime. Destructors have the same name as the class, prefixed with a tilde (`~`), and do not take any arguments or have a return type.

```cpp
// Example: Constructors and destructors

#include <iostream>
#include <string>

class Dog {
public:
    std::string name;
    int age;

    // Default Constructor
    Dog() {
        name = "Unknown";
        age = 0;
        std::cout << "Default Dog created!" << std::endl;
    }

    // Parameterized Constructor
    Dog(std::string n, int a) {
        name = n;
        age = a;
        std::cout << "Parameterized Dog " << name << " created!" << std::endl;
    }

    // Copy Constructor
    Dog(const Dog& other) {
        name = "Copy of " + other.name;
        age = other.age;
        std::cout << "Copy Dog " << name << " created!" << std::endl;
    }

    // Destructor
    ~Dog() {
        std::cout << "Dog " << name << " destroyed!" << std_endl;
    }

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

int main() {
    Dog dog1; // Calls default constructor
    dog1.display();

    Dog dog2("Buddy", 3); // Calls parameterized constructor
    dog2.display();

    Dog dog3 = dog2; // Calls copy constructor
    dog3.display();

    // Objects are destroyed when they go out of scope (end of main function)
    return 0;
}
```

Encapsulation (access specifiers)

Encapsulation is one of the fundamental principles of OOP. It involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit (class). It also involves restricting direct access to some of an object's components, which is achieved using **access specifiers**:

- **`public`:** Members declared `public` are accessible from anywhere outside the class. They form the interface of the class.
- **`private`:** Members declared `private` are only accessible from within the same class. They are typically used for data members to protect them from external modification, promoting data hiding.
- **`protected`:** Members declared `protected` are accessible from within the same class and by derived classes (in inheritance). This will be covered in advanced OOP concepts.

Encapsulation helps in achieving data hiding and abstraction, making the code more secure, maintainable, and easier to debug.

```cpp
// Example: Encapsulation (access specifiers)

#include <iostream>
#include <string>

class BankAccount {
private:
    double balance; // Private data member

public:
    std::string accountNumber; // Public data member

    // Constructor
    BankAccount(std::string accNum, double initialBalance) {
        accountNumber = accNum;
        if (initialBalance >= 0) {
            balance = initialBalance;
        } else {
            balance = 0;
            std::cout << "Initial balance cannot be negative. Setting to 0." <<
std::endl;
        }
    }

    // Public member function to deposit money (interface to modify private
data)
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited: " << amount << ", New balance: " <<
balance << std::endl;
        } else {
            std::cout << "Deposit amount must be positive." << std::endl;
        }
    }

    // Public member function to withdraw money
    void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            std::cout << "Withdrew: " << amount << ", New balance: " << balance
<< std::endl;
        } else {
            std::cout << "Invalid withdrawal amount or insufficient balance."
<< std::endl;
        }
    }

    // Public member function to get balance (interface to access private data)
    double getBalance() {
        return balance;
    }
};

int main() {
    BankAccount myAccount("12345", 1000.0);

    // myAccount.balance = 500; // ERROR: 'balance' is private

    myAccount.deposit(200.0);
    myAccount.withdraw(300.0);
```

```
    myAccount.withdraw(1500.0); // Insufficient balance

    std::cout << "Current balance: " << myAccount.getBalance() << std::endl;

    return 0;
}
```

Advanced Object-Oriented Programming Concepts

Advanced OOP concepts in C++ build upon the foundational principles of classes and objects, enabling more complex and flexible software designs. These concepts are crucial for developing robust, scalable, and maintainable applications.

Inheritance (single, multiple, multilevel)

Inheritance is a mechanism in OOP that allows a new class (derived class or subclass) to inherit properties and behaviors from an existing class (base class or superclass). This promotes code reusability and establishes an "is-a" relationship between classes.

- **Single Inheritance:** A derived class inherits from only one base class.
- **Multiple Inheritance:** A derived class inherits from multiple base classes. This can lead to complexities like the "diamond problem" but offers powerful design options.
- **Multilevel Inheritance:** A derived class inherits from a base class, which in turn is inherited from another base class, forming a chain of inheritance.

```cpp
// Example: Inheritance (Single, Multiple, Multilevel)

#include <iostream>
#include <string>

// Base class for Single and Multilevel Inheritance
class Animal {
public:
    void eat() {
        std::cout << "Animal eats." << std::endl;
    }
};

// Single Inheritance: Dog inherits from Animal
class Dog : public Animal {
public:
    void bark() {
        std::cout << "Dog barks." << std::endl;
    }
};

// Base class for Multiple Inheritance
class Swimmer {
public:
    void swim() {
        std::cout << "Swims in water." << std::endl;
    }
};

// Multiple Inheritance: Duck inherits from Animal and Swimmer
class Duck : public Animal, public Swimmer {
public:
    void quack() {
        std.cout << "Duck quacks." << std::endl;
    }
};

// Multilevel Inheritance: Puppy inherits from Dog, which inherits from Animal
class Puppy : public Dog {
public:
    void play() {
        std::cout << "Puppy plays." <<

        std::cout << std::endl;
    }
};

int main() {
    // Single Inheritance
    Dog myDog;
    myDog.eat();  // From Animal
    myDog.bark(); // From Dog

    // Multiple Inheritance
    Duck myDuck;
    myDuck.eat();  // From Animal
    myDuck.swim(); // From Swimmer
    myDuck.quack(); // From Duck

    // Multilevel Inheritance
    Puppy myPuppy;
```

```
    myPuppy.eat();  // From Animal (via Dog)
    myPuppy.bark(); // From Dog
    myPuppy.play(); // From Puppy

    return 0;
}
```

Polymorphism (virtual functions)

Polymorphism means "many forms." In C++, it allows objects of different classes to be treated as objects of a common base class. This is primarily achieved through **virtual functions**.

- **Virtual Function:** A member function declared in a base class that is redefined (overridden) in a derived class. When a virtual function is called through a pointer or reference to the base class, the version of the function executed is determined by the type of the object pointed to or referred to, not by the type of the pointer or reference. This is known as **runtime polymorphism** or **dynamic dispatch**.

```cpp
// Example: Polymorphism (virtual functions)

#include <iostream>
#include <string>

class Shape {
public:
    // Virtual function
    virtual void draw() {
        std::cout << "Drawing a generic shape." << std::endl;
    }

    // Virtual destructor is important for proper cleanup in polymorphic
hierarchies
    virtual ~Shape() {
        std::cout << "Shape destructor called." << std::endl;
    }
};

class Circle : public Shape {
public:
    void draw() override { // override keyword (C++11) is good practice
        std::cout << "Drawing a circle." << std::endl;
    }
    ~Circle() {
        std::cout << "Circle destructor called." << std::endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        std::cout << "Drawing a rectangle." << std::endl;
    }
    ~Rectangle() {
        std::cout << "Rectangle destructor called." << std::endl;
    }
};

int main() {
    Shape* s1 = new Circle();
    Shape* s2 = new Rectangle();
    Shape* s3 = new Shape();

    s1->draw(); // Calls Circle's draw()
    s2->draw(); // Calls Rectangle's draw()
    s3->draw(); // Calls Shape's draw()

    delete s1;
    delete s2;
    delete s3;

    return 0;
}
```

Abstract classes and interfaces

- **Abstract Class:** A class that cannot be instantiated directly. It contains at least one **pure virtual function** (a virtual function declared with `= 0`). Abstract classes serve as base classes for other classes, providing a common interface.
- **Pure Virtual Function:** A virtual function declared with `= 0` in the base class. It has no definition in the base class and must be implemented by any concrete (non-abstract) derived class.
- **Interface (in C++ context):** While C++ doesn't have a specific `interface` keyword like Java, an abstract class with *only* pure virtual functions (and possibly a destructor) effectively acts as an interface. It defines a contract that derived classes must fulfill.

```cpp
// Example: Abstract classes and interfaces

#include <iostream>

// Abstract class
class Vehicle {
public:
    // Pure virtual function
    virtual void start() = 0;
    virtual void stop() = 0;

    // Regular virtual function (can have implementation or be overridden)
    virtual void honk() {
        std::cout << "Vehicle honks." << std::endl;
    }

    // Virtual destructor is crucial for proper cleanup
    virtual ~Vehicle() {
        std::cout << "Vehicle destructor called." << std::endl;
    }
};

// Concrete class implementing Vehicle
class Car : public Vehicle {
public:
    void start() override {
        std::cout << "Car starts with ignition." << std::endl;
    }
    void stop() override {
        std::cout << "Car stops with brakes." << std::endl;
    }
    // honk() is inherited and can be used or overridden
    ~Car() {
        std::cout << "Car destructor called." << std::endl;
    }
};

// Another concrete class implementing Vehicle
class Bicycle : public Vehicle {
public:
    void start() override {
        std::cout << "Bicycle starts pedaling." << std::endl;
    }
    void stop() override {
        std::cout << "Bicycle stops with feet." << std::endl;
    }
    void honk() override { // Overriding honk()
        std::cout << "Bicycle rings bell." << std::endl;
    }
    ~Bicycle() {

        std::cout <<


### Static members

**Static members** (both data members and member functions) belong to the class itself, rather than to any specific object of the class. They are shared by all objects of the class.

*   **Static Data Members:** There is only one copy of a static data member for
```

```
the entire class, regardless of how many objects are created. They must be
defined outside the class definition.
*   **Static Member Functions:** Can only access static data members and other
static member functions. They cannot access non-static (instance-specific) data
members or functions because they are not associated with a particular object.

```cpp
// Example: Static members

#include <iostream>

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

    MyClass() {
        count++; // Increment count every time an object is created
    }

    ~MyClass() {
        count--; // Decrement count when an object is destroyed
    }

    static void displayCount() { // Static member function
        std::cout << "Number of objects: " << count << std::endl;
    }
};

// Definition of static data member (outside the class)
int MyClass::count = 0;

int main() {
    MyClass::displayCount(); // Call static function using class name

    MyClass obj1;
    MyClass obj2;
    MyClass::displayCount();

    { // New scope
        MyClass obj3;
        MyClass::displayCount();
    } // obj3 is destroyed here

    MyClass::displayCount();

    return 0;
}
```

Friend functions and classes

In C++, `friend` functions and `friend` classes are mechanisms that allow non-member functions or other classes to access the `private` and `protected` members of a class. While they break the strict encapsulation principle, they can be useful in specific scenarios, such as operator overloading or when two classes are tightly coupled.
```

- **Friend Function:** A non-member function that is granted special permission to access the `private` and `protected` members of a class. It is declared inside the class with the `friend` keyword.
- **Friend Class:** When an entire class is declared as a `friend` of another class, all member functions of the friend class can access the `private` and `protected` members of the class that declared it as a friend.

```
// Example: Friend functions and classes

#include <iostream>

class MyClass;

// Friend function declaration
void showPrivateData(MyClass& obj);

class MyClass {
private:
    int private_data;

public:
    MyClass() : private_data(100) {}

    // Declare showPrivateData as a friend function
    friend void showPrivateData(MyClass& obj);

    // Declare AnotherClass as a friend class
    friend class AnotherClass;
};

// Friend function definition
void showPrivateData(MyClass& obj) {
    std::cout << "Friend function accessing private_data: " << obj.private_data
<< std::endl;
}

class AnotherClass {
public:
    void accessMyClassPrivate(MyClass& obj) {
        std::cout << "Friend class accessing private_data: " <<
obj.private_data << std::endl;
    }
};

int main() {
    MyClass obj;
    showPrivateData(obj); // Call friend function

    AnotherClass anotherObj;
    anotherObj.accessMyClassPrivate(obj); // Call friend class member function

    return 0;
}
```