# MQF633 C++ For Financial Engineering

# C++ Functions and Modular Programing

## Part I: Introduction Functions

A function is a named, self-contained block of code that performs a specific task. Functions are essential for code organization, reuse, and modularity.

### Function Declaration

Declaration includes the function’s:

1. Return type
2. Name
3. Parameters (if any)

int add(int a, int b);

### Function Definition

Definition provides the actual implementation of the function.

int add(int a, int b) {

return a + b;

}

**Function Call**

Invoking a function involves providing arguments

int result = add(3, 4);

### Function Parameter and Return Types

#### Parameters

Input values passed to a function. Parameters are defined in the function declaration.

void greet(std::string name);

In C++, you can pass parameters to functions either by reference or by value (copy). Let's look at examples for both scenarios:

Pass by value (copy)

#include <iostream>

// Function that takes two integers by value and returns their sum

int addByValue(int a, int b) {

return a + b;

}

int main() {

int x = 5, y = 7;

// Passing values to the function by copy

int result = addByValue(x, y);

std::cout << "Result (Pass by Value): " << result << std::endl;

// The original values of x and y remain unchanged

std::cout << "Original x: " << x << ", y: " << y << std::endl;

return 0;

}

Pass by reference

#include <iostream>

// Function that takes two integers by reference and modifies them

void addByReference(int &a, int &b) {

a += 2;

b += 3;

}

int main() {

int x = 5, y = 7;

// Passing values to the function by reference

addByReference(x, y);

std::cout << "Modified x: " << x << ", y: " << y << std::endl;

return 0;

}

In this example, the addByReference function takes parameters by reference using the & symbol. Modifications made to the parameters inside the function directly affect the original variables.

These examples demonstrate the difference between passing values by copy and by reference. Choosing between them depends on the use case and whether you want the function to modify the original variables or not.

### Using const reference

Using a const reference in C++ is a common practice, especially when you want to pass parameters to a function without allowing the function to modify the original values. Here's an example

#include <iostream>

#include <string>

// Function that takes a constant reference to a string and prints it

void printString(const std::string &str) {

std::cout << "Received string: " << str << std::endl;

// Uncommenting the line below would result in a compilation error

// str += " (modified)"; // Error: assignment of read-only reference 'str'

}

int main() {

std::string message = "Hello, C++!";

// Passing a string by constant reference

printString(message);

// The original string remains unchanged

std::cout << "Original string: " << message << std::endl;

return 0;

}

In this example, the printString function takes a const reference to a std::string. This means that the function cannot modify the content of the string. Attempting to modify the string inside the function would result in a compilation error.

Using const references is beneficial when you want to avoid unnecessary copies of large objects (like strings or complex data structures) and ensure that the function does not accidentally modify the input parameters.

### Return Values

Functions can return a value using the return statement.

int square(int x) {

return x \* x;

}

In C++, you can return a structure from a function. This allows you to encapsulate multiple data members into a single unit and return it as a result. Here's an example

#include <iostream>

// Define a structure named Point

struct Point {

int x;

int y;

};

// Function that returns a Point structure

Point createPoint(int x, int y) {

Point p;

p.x = x;

p.y = y;

return p;

}

int main() {

// Call the function and store the returned Point structure

Point myPoint = createPoint(3, 7);

// Access the members of the returned structure

std::cout << "X-coordinate: " << myPoint.x << std::endl;

std::cout << "Y-coordinate: " << myPoint.y << std::endl;

return 0;

}

In C++, you can return a pointer from a function. This is useful when you want to return dynamically allocated memory or when you want to return the address of an object. Here's an example

#include <iostream>

// Function that returns a pointer to an integer

int\* createInteger() {

// Dynamically allocate memory for an integer

int\* ptr = new int;

// Initialize the value

\*ptr = 42;

// Return the pointer

return ptr;

}

int main() {

// Call the function and get the returned pointer

int\* myIntegerPtr = createInteger();

// Access the value through the pointer

std::cout << "Value: " << \*myIntegerPtr << std::endl;

// Don't forget to free the allocated memory to avoid memory leaks

delete myIntegerPtr;

return 0;

}

In C++, you can return a reference from a function. This is often used when you want to return a reference to an existing object, avoiding unnecessary copies. Here's an example:

In C++, you can return a reference from a function. This is often used when you want to return a reference to an existing object, avoiding unnecessary copies. Here's an example:

#include <iostream>

#include <vector>

// Function that returns a reference to a vector

std::vector<int>& getVector() {

// Declare a static vector (for the sake of example)

static std::vector<int> myVector = {1, 2, 3, 4, 5};

// Return a reference to the vector

return myVector;

}

int main() {

// Call the function and get the returned reference

std::vector<int>& vecRef = getVector();

// Modify the vector through the reference

vecRef.push\_back(6);

// Print the modified vector

for (int value : vecRef) {

std::cout << value << " ";

}

std::cout << std::endl;

return 0;

}

Explanation:

1. The getVector function returns a reference to a static std::vector<int>. The use of static here is just for the sake of example; in real-world scenarios, you might have dynamic objects or obtain references to existing objects.
2. In the main function, the returned reference is assigned to vecRef.
3. The vector is modified through the reference by adding a new element.
4. The modified vector is then printed.

It's crucial to be cautious when returning references, especially to local variables or temporary objects, as they might go out of scope and lead to undefined behaviour. In this example, using a static variable helps avoid such issues, but in practice, you might return references to objects with a longer lifespan.

### Function Overloading

Definition: Multiple functions with the same name but different parameter types or count.

#include <iostream>

// Function to add two integers

int add(int a, int b) {

return a + b;

}

// Function to concatenate two strings

std::string add(std::string str1, std::string str2) {

return str1 + str2;

}

// Function to add three integers

int add(int a, int b, int c) {

return a + b + c;

}

int main() {

// Example 1: Adding two integers

int sum1 = add(3, 5);

std::cout << "Sum of two integers: " << sum1 << std::endl;

// Example 2: Concatenating two strings

std::string resultStr = add("Hello, ", "World!");

std::cout << "Concatenated string: " << resultStr << std::endl;

// Example 3: Adding three integers

int sum2 = add(2, 4, 6);

std::cout << "Sum of three integers: " << sum2 << std::endl;

return 0;

}

## Default Arguments

### Default Arguments

Parameters with default values.

#include <iostream>

// Function with default arguments

int multiply(int a, int b = 2, int c = 1) {

return a \* b \* c;

}

int main() {

// Example 1: Using all parameters

int result1 = multiply(3, 4, 2);

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

// Example 2: Using only the first parameter

int result2 = multiply(5);

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

// Example 3: Using the first and second parameters

int result3 = multiply(2, 3);

std::cout << "Result 3: " << result3 << std::endl;

return 0;

}

## Recursive Functions

### Recursive Functions

Definition: Recursive functions in C++ offer several benefits, but it's essential to use them judiciously to avoid stack overflow and inefficiencies. Here are some advantages of using recursive functions:

1. Elegance and Readability: Recursive solutions can be more elegant and easier to read than their iterative counterparts, especially for problems with inherently recursive structures.
2. Simplifies Complex Problems: Recursive functions can simplify the solution of problems that have a recursive nature or are naturally defined in a recursive manner.
3. Handling Dynamic Data Structures: Recursive functions are well-suited for working with dynamic data structures like trees and graphs, where the recursive structure mirrors the inherent structure of the data.
4. Mathematical Expressions: Recursive functions are often used to represent mathematical expressions and formulas, providing a natural and concise way to express relationships.
5. Tail Recursion Optimization:In some cases, compilers can optimize tail-recursive calls, eliminating the need for additional stack frames and reducing the risk of stack overflow.

* Remember to handle the initial condition

#include <iostream>

// Recursive function to calculate the factorial of a number

unsigned long long factorial(int n) {

// Base case: factorial of 0 is 1

if (n == 0 || n == 1) {

return 1;

} else {

// Recursive case: n! = n \* (n-1)!

return static\_cast<unsigned long long>(n) \* factorial(n - 1);

}

}

int main() {

// Calculate and display the factorial of 5

int number = 5;

unsigned long long result = factorial(number);

std::cout << "Factorial of " << number << " is: " << result << std::endl;

return 0;

}

### Scope and Lifetime of Variables

* Local variables exist only within the function.
* Function parameters are local variables.
* Global variables have a broader scope.

#include <iostream>

// Global variable with file scope and static duration

int globalVariable = 10;

void exampleFunction() {

// Local variable with function scope and automatic duration

int localVar = 5;

// Accessing the global variable

std::cout << "Inside function - Global variable: " << globalVariable << std::endl;

// Accessing the local variable

std::cout << "Inside function - Local variable: " << localVar << std::endl;

}

int main() {

// Accessing the global variable

std::cout << "In main - Global variable: " << globalVariable << std::endl;

// Local variable with block scope and automatic duration

int localVar = 7;

// Accessing the local variable

std::cout << "In main - Local variable: " << localVar << std::endl;

// Call the function

exampleFunction();

// Attempting to access the local variable from the function (will result in an error)

// std::cout << "In main after function - Local variable: " << localVar << std::endl;

return 0;

}

1. globalVariable is a global variable with file scope and static duration. It exists throughout the entire program and can be accessed from any function.
2. localVar is a local variable declared within the main function. It has function scope and automatic duration. It exists only within the block where it is declared and is destroyed when the block is exited.
3. The exampleFunction function has access to the global variable globalVariable. However, it cannot directly access the local variable localVar from the main function because of its local scope.
4. When the main function is called, it first prints the global and local variables within its own scope. It then calls the exampleFunction, which prints the global variable accessible from within the function.
5. Attempting to access localVar from the main function after the exampleFunction call would result in a compilation error because the variable is out of scope.
6. Understanding the scope and lifetime of variables is crucial for writing correct and maintainable C++ code.

### Function Prototypes

Declare function prototype in header file and then implement the function definition (body) later.

// this prototype in header file

int add(int a, int b);

// implement this later with header file #include

int add(int a, int b) {

return a+b;

}

## Inline Functions

Definition: tell the compiler to replace the function call with the actual code

#include <iostream>

// Example of an inline function

inline int square(int x) {

return x \* x;

}

int main() {

int result = square(5);

std::cout << "Square: " << result << std::endl;

return 0;

}

When to use Inline function?

Small Functions:

Inline functions are most effective when they are small in terms of code. If a function's code is larger, the overhead of inlining might outweigh the benefits, and the compiler may choose not to inline the function.

Frequent Function Calls:

Inline functions are useful when a function is called frequently. Inlining avoids the overhead of a function call, resulting in potentially better performance.

Simple Operations:

Functions containing simple operations or calculations are good candidates for inlining. For instance, getter and setter functions, or functions performing basic arithmetic, can benefit from inlining.

Header-Only Libraries:

When designing header-only libraries, inline functions are often used. This allows the compiler to generate code inline at the point of use, reducing the chances of multiple definition errors.

Function Templates:

Functions defined within class templates in headers are implicitly inline. This is common in generic programming and allows the compiler to generate specialized code for different template arguments.

Performance-Critical Code:

Inlining can be beneficial in performance-critical sections of code where the elimination of function call overhead is crucial for achieving optimal performance.

Constexpr Functions:

Constexpr functions, which are evaluated at compile-time, are implicitly inline. Using inline functions for simple compile-time calculations can improve code efficiency.

### Template function

Template functions in C++ provide a powerful mechanism for writing generic code that can work with different data types. There are several reasons to use template functions:

1. Code Reusability: Templates allow you to write a single function or class that can work with different data types. This promotes code reusability, as you don't need to rewrite the same logic for each specific data type.
2. Type Safety: Even though templates provide flexibility for working with various types, the type safety of C++ is maintained. The compiler generates specific instances of the template for each data type, ensuring type correctness at compile-time.
3. Flexibility: Templates provide a way to create flexible and generic algorithms that can operate on various types without sacrificing performance or safety.
4. Standard Template Library (STL): The C++ Standard Template Library heavily utilizes templates. Containers like std::vector, std::list, and algorithms like std::sort are implemented using templates, allowing them to be used with different data types.
5. Performance: Templates are resolved at compile-time, which can result in more efficient code execution compared to runtime polymorphism. The compiler generates specialized code for each data type, avoiding the overhead of runtime type checks.
6. Consistency: Templates allow you to write consistent and uniform code for different data types, promoting a cleaner and more maintainable codebase.

Here's a simple example of a template function that swaps two values:

#include <iostream>

// Template function to swap two values of any data type

template <typename T>

void swapValues(T &a, T &b) {

T temp = a;

a = b;

b = temp;

}

int main() {

// Example with integers

int x = 5, y = 10;

std::cout << "Before swapping: x = " << x << ", y = " << y << std::endl;

// Call the template function to swap integers

swapValues(x, y);

std::cout << "After swapping: x = " << x << ", y = " << y << std::endl;

// Example with double values

double a = 3.14, b = 2.718;

std::cout << "Before swapping: a = " << a << ", b = " << b << std::endl;

// Call the template function to swap doubles

swapValues(a, b);

std::cout << "After swapping: a = " << a << ", b = " << b << std::endl;

return 0;

}

In this example, the swapValues function is a template function that can swap values of any data type (int, double, etc.). The main function demonstrates using the template function with both integer and double values.

When you compile and run this program, you should see the values being swapped correctly for both types. Templates provide a way to write generic and reusable code for different data types.

### Cautions!!

While C++ template functions are powerful and useful in many situations, there are scenarios where their usage might be avoided or should be approached with caution:

* Code Readability: Excessive use of templates, especially in complex scenarios, can lead to code that is difficult to read and understand. If not used judiciously, it may make the codebase more challenging for others (or even yourself) to comprehend.
* Compilation Time: Templates can result in longer compilation times, especially when dealing with large and complex template-based code. This can be a concern in projects where fast compilation is crucial.
* Debugging: Debugging template code can be more challenging compared to non-template code. Error messages from the compiler might be cryptic, making it harder to identify and fix issues.
* Instantiation Overhead: The compiler generates code for each template instantiation. If the template is instantiated with numerous different types or in various places throughout the code, it may lead to increased binary size.
* Complexity: Templates introduce a level of complexity, and using them unnecessarily may lead to overengineering. Simple functions that work with a fixed set of data types may not benefit significantly from being templated.
* Compatibility: In some cases, template code might not be as compatible across different compilers or may have subtle differences in behavior. This can be a concern when writing code intended for use in various environments.
* Learning Curve: Templates, especially advanced template metaprogramming techniques, can have a steep learning curve. If the team is not familiar with template concepts, it might be preferable to use simpler constructs for better maintainability.
* Tool Support: Some tools and IDEs might have limitations in handling template-based code, affecting features like code navigation, auto-completion, or refactoring.

### Function Pointer

Definition: in C++, function pointers allow you to store the address of a function and later invoke it using the pointer. This feature is particularly useful when dealing with callbacks or dynamically choosing functions to execute. Below are key points and examples illustrating the use of function pointers in C++.

Define a function pointer using the function's signature. Syntax: returnType (\*pointerName) (parameterTypes);

// Example of a function pointer declaration

int (\*addPtr) (int, int);

Assign the address of a function to a function pointer.

// Example of assigning function addresses to pointers

int add(int a, int b) {

return a + b;

}

addPtr = &add; // or addPtr = add;

Call a function through its pointer using the dereference operator (\*ptr).

// Example of invoking a function through a pointer

int result = (\*addPtr)(3, 4);

// or

int result = addPtr(3, 4);

Pass function pointers as parameters to other functions.

// Example of using function pointers as parameters

void performOperation(int a, int b, int (\*operation)(int, int)) {

int result = operation(a, b);

std::cout << "Result: " << result << std::endl;

}

performOperation(5, 3, addPtr);

Function Pointers in Callbacks to have more flexiblity

// Example of using function pointers for callbacks

void callbackFunction() {

std::cout << "Callback function called." << std::endl;

}

void performCallback(void (\*callback)()) {

callback();

}

performCallback(callbackFunction);

Use Typedef to improve the code readability.

Use Typedef to improve code readability for function pointer

#include <iostream>

// Function to add two numbers

int add(int a, int b) {

return a + b;

}

// Function to subtract two numbers

int subtract(int a, int b) {

return a - b;

}

// Define a typedef for a function pointer type

typedef int (\*BinaryOperation)(int, int);

// Function that performs an operation using a function pointer

int performOperation(int a, int b, BinaryOperation operation) {

return operation(a, b);

}

int main() {

// Declare function pointers using the typedef

BinaryOperation addPtr = add;

BinaryOperation subtractPtr = subtract;

// Use the function pointers

std::cout << "Addition: " << performOperation(5, 3, addPtr) << std::endl;

std::cout << "Subtraction: " << performOperation(5, 3, subtractPtr) << std::endl;

return 0;

}

## **Lambda Expressions**

Definition: Lambda expressions in C++ provide a concise way to create anonymous functions or function objects. They are particularly useful in situations where a short-lived function is needed, such as in algorithms, event handling, or functional programming constructs.

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

// Example 1: Lambda expression for a simple addition function

auto add = [](int a, int b) { return a + b; };

std::cout << "Sum: " << add(3, 4) << std::endl;

// Example 2: Lambda expression for a predicate in a standard algorithm

std::vector<int> numbers = {5, 2, 8, 1, 7, 3};

// Sorting the vector in descending order using a lambda expression

std::sort(numbers.begin(), numbers.end(), [](int a, int b) { return a > b; });

std::cout << "Sorted numbers in descending order: ";

for (int num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

// Example 3: Lambda expression capturing variables from the surrounding scope

int base = 10;

auto addWithBase = [base](int value) { return base + value; };

std::cout << "Result with captured base: " << addWithBase(5) << std::endl;

return 0;

}

In this example:

* Simple Addition Lambda:

auto add = [](int a, int b) { return a + b; };

The lambda expression takes two integers and returns their sum.

The auto keyword is used to let the compiler deduce the lambda's type.

* Lambda in a Standard Algorithm:

std::sort(numbers.begin(), numbers.end(), [](int a, int b) { return a > b; });

A lambda expression is used as a comparator function to sort a vector in descending order.

* Capturing Variables:

auto addWithBase = [base](int value) { return base + value; };

The lambda captures the variable base from the surrounding scope.

The captured variable can be used inside the lambda function.

Lambda expressions are especially powerful when used in conjunction with algorithms and functions that accept function objects. They provide a concise and readable way to define functionality at the point of use.

### C++ function library

Function Libraries:

* C++ Standard Library:

The C++ Standard Library is a collection of classes and functions that provide common functionality.

It includes components for input and output, strings, algorithms, data structures, and more.

Standard Library headers typically do not have a .h extension (e.g., <iostream> instead of <iostream.h>).

* C Standard Library:

The C Standard Library functions are also available in C++, providing fundamental operations like memory allocation, I/O, and string manipulation.

Examples include functions like printf, scanf, malloc, and free.

Standard Template Library (STL):

Containers:

The STL provides various container classes like vectors, lists, queues, stacks, and maps.

Containers facilitate easy storage and retrieval of data in different ways.

Algorithms:

The STL includes a wide range of algorithms that operate on containers.

Examples include sorting (std::sort), searching (std::find), and manipulating ranges of elements.

Iterators:

Iterators provide a uniform way to access elements in a container.

Algorithms in the STL often work with iterators, allowing for generic and efficient algorithms.

Function Objects (Functors):

Functors are objects that can be called like functions.

They are often used as arguments for algorithms, providing a way to customize their behavior.

Utilities:

The STL includes utility components such as pairs, tuples, and smart pointers.

Pairs and tuples can be used to store heterogeneous data.

IOStream Library:

The IOStream library (<iostream>) provides a flexible and extensible framework for input and output operations.

It includes cin, cout, and related stream classes.

The C++ Standard Template Library (STL) provides a powerful string class in the <string> header, offering a convenient and efficient way to handle strings. Here's a simple example demonstrating the usage of std::string:

#include <iostream>

#include <string>

int main() {

// Example 1: Initializing and Assigning

std::string greeting = "Hello, ";

std::string name = "John";

// Example 2: Concatenation

std::string fullGreeting = greeting + name;

// Example 3: String Length

std::cout << "Length of the greeting: " << greeting.length() << std::endl;

// Example 4: Accessing Characters

char firstChar = name[0];

std::cout << "First character of the name: " << firstChar << std::endl;

// Example 5: Iterating through characters

std::cout << "Characters in the name: ";

for (char ch : name) {

std::cout << ch << " ";

}

std::cout << std::endl;

// Example 6: Substring

std::string subString = fullGreeting.substr(0, 5);

// Example 7: Find

size\_t position = fullGreeting.find("John");

if (position != std::string::npos) {

std::cout << "Substring 'John' found at position: " << position << std::endl;

} else {

std::cout << "Substring 'John' not found." << std::endl;

}

return 0;

}

This example covers various operations with std::string:

1. Initializing and Assigning: std::string greeting = "Hello, ";
2. Concatenation: std::string fullGreeting = greeting + name;
3. String Length: greeting.length()
4. Accessing Characters: char firstChar = name[0];
5. Iterating through Characters: for (char ch : name)
6. Substring: fullGreeting.substr(0, 5);
7. Find Substring: fullGreeting.find("John");

These operations showcase the versatility and convenience of the std::string class for various string manipulations in C++

The std::vector is a versatile and widely used container in the C++ Standard Template Library (STL) for dynamic arrays. Here's an example demonstrating the usage of std::vector.

#include <iostream>

#include <vector>

int main() {

// Example 1: Initializing and Populating

std::vector<int> numbers = {1, 2, 3, 4, 5};

// Example 2: Accessing Elements

std::cout << "First element: " << numbers[0] << std::endl;

std::cout << "Size of the vector: " << numbers.size() << std::endl;

// Example 3: Modifying Elements

numbers[2] = 10;

// Example 4: Iterating through Elements

std::cout << "Vector elements: ";

for (int num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

// Example 5: Adding Elements (push\_back)

numbers.push\_back(6);

numbers.push\_back(7);

// Example 6: Removing Elements (pop\_back)

numbers.pop\_back();

// Example 7: Finding an Element

int target = 4;

auto it = std::find(numbers.begin(), numbers.end(), target);

return 0;

}

Summary of common method:

* Initializing and Populating:std::vector<int> numbers = {1, 2, 3, 4, 5};
* Accessing Elements: numbers[0]
* Checking size: numbers.size()
* Modifying Elements:numbers[2] = 10;
* Iterating through Elements: for (int num : numbers)
* Adding Elements (push\_back): numbers.push\_back(6);
* Removing Elements (pop\_back): numbers.pop\_back();
* Finding an Element: Using std::find to find the position of a specific element.

These operations demonstrate the dynamic nature of std::vector and its ability to efficiently manage collections of elements in C++

## C++ Function Best Practice

Function best practices are essential for writing clear, maintainable, and efficient C++ code. Here are some recommended best practices when it comes to writing functions in C++:

1. Modularity and Single Responsibility:

Keep functions small and focused on a single responsibility.

Each function should perform a well-defined task.

2. Descriptive Naming:

Use clear and descriptive names for functions that reflect their purpose.

Choose names that convey the function's actions or intentions.

3. Avoid Global Variables:

Minimize the use of global variables inside functions.

Prefer passing parameters or using local variables.

4. Consistent Parameter Naming:

Be consistent with parameter naming across functions.

Use meaningful names that reflect the purpose of each parameter.

5. Return Type Clarity:

Choose meaningful return types that reflect the nature of the data being returned.

Avoid using ambiguous return types.

6. Error Handling:

Use exceptions for exceptional circumstances, not for expected program flow.

Clearly document error-handling strategies in function comments.

7. Const-Correctness:

Use const wherever possible to indicate that a function does not modify its parameters.

Use const member functions for methods that do not modify the object's state.

8. Function Overloading:

Use function overloading to provide multiple versions of a function with different parameter sets.

Avoid overloading with similar signatures to prevent confusion.

9. Default Parameters:

Use default parameters judiciously to simplify function calls.

Ensure that default values make sense and do not lead to unexpected behavior.

## Part II: C++ Modular Programing

Definition: modular programming is a programming paradigm that encourages the separation of a program into independent, interchangeable modules. Each module represents a different aspect of the program's functionality.

Below example shows 3 files in a small project.

**main.cpp (Main Program)**

// main.cpp

#include <iostream>

#include "math\_operations.h"

int main() {

int a = 5, b = 3;

// Calling functions from the math\_operations module

int sum\_result = add(a, b);

int diff\_result = subtract(a, b);

int product\_result = multiply(a, b);

// Displaying results

std::cout << "Sum: " << sum\_result << std::endl;

std::cout << "Difference: " << diff\_result << std::endl;

std::cout << "Product: " << product\_result << std::endl;

return 0;

}

**math\_operations.h (Header File)**

// math\_operations.h

#ifndef MATH\_OPERATIONS\_H

#define MATH\_OPERATIONS\_H

// Function prototypes

int add(int a, int b);

int subtract(int a, int b);

int multiply(int a, int b);

#endif

**math\_operations.cpp (Implementation)**

// math\_operations.cpp

#include "math\_operations.h"

// Function definitions

int add(int a, int b) {

return a + b;

}

int subtract(int a, int b) {

return a - b;

}

int multiply(int a, int b) {

return a \* b;

}

In this example:

* main.cpp is the main program file that includes the declarations for the functions in math\_operations.h and calls those functions.
* math\_operations.h is a header file containing the function prototypes for the mathematical operations.
* math\_operations.cpp contains the implementations of the mathematical operations.

This separation of code into different files promotes modularity, making it easier to understand, maintain, and extend the codebase. Each module focuses on a specific aspect of the program, promoting code organization and readability. This is a fundamental aspect of modular programming, which becomes increasingly important as programs grow in complexity.

## Homework assignment

Create a project of console application, compute the vanilla option present value using Black Sholes model.

Requirements:

* contains multiple files in this project. Black.h for declaration of function, Black.cpp for implementation of pricing function, and a main cpp for the main program
* In the main program:
  + load the set of pre-defined trades in a txt file, and then compute the pv of each trade;
  + Store all trade pv in an vector, and then output the pv into a result.txt file
  + Implement the error handling for the computation to check any possible error

## Appendix

### Quiz

**1. What is the purpose of function prototypes in C++?**

**A. To declare a function's implementation**

**B. To declare a function's signature before its implementation**

**C. To define a function's parameters**

**D. To specify the return type of a function**

**2. Which keyword is used to define a template function in C++?**

**A. template**

**B. typename**

**C. class**

**D. generic**

**3. In C++, what is the significance of the const qualifier in a member function?**

**A. It indicates that the function is a constructor.**

**B. It specifies that the function does not modify the object's state.**

**C. It indicates that the function is a destructor.**

**D. It specifies that the function returns a constant value.**

**4. What is the purpose of function overloading in C++?**

**A. To reduce code duplication**

**B. To allow a function to return multiple values**

**C. To provide multiple implementations for the same function name**

**D. To optimize function performance**

**5. Which STL header should be included to use the std::vector container?**

**A. <list>**

**B. <deque>**

**C. <vector>**

**D. <array>**

**6. How do you pass an array to a function in C++?**

**A. Using a reference**

**B. Using a pointer**

**C. Using sizeof operator**

**D. All of the above**

**7. What does the auto keyword do in a lambda expression?**

**A. Specifies the lambda's return type**

**B. Indicates that the lambda is a template**

**C. Automatically deduces the lambda's return type**

**D. Declares a lambda function**

**8. What is the purpose of the const\_iterator in C++ STL containers?**

**A. It allows modifying the elements of a container.**

**B. It specifies that the container is constant.**

**C. It designates an iterator that cannot modify the container's elements.**

**D. It is used to iterate only over constant containers.**

**9. Which algorithm is used to find the minimum element in a range in C++ STL?**

**A. std::min\_element**

**B. std::find**

**C. std::lower\_bound**

**D. std::search**

**10. How can you handle errors in C++ functions?**

**A. Using try, catch, and throw for exceptions**

**B. Returning an error code**

**C. Printing an error message to the console**

**D. All of the above**