ASSIGNMENT – 2

Q.1 What is the purpose of the main function in a C++ program?

The purpose of the main function in a C++ program is to serve as the entry point where the execution of the program begins. When the program is run, the main function is the first function that is called, and it contains the code that will be executed. It is a required function in every C++ program.

Key points about the `main` function:

- It must return an integer (usually `0` to indicate successful execution).
- It can take command-line arguments as input (though this is optional).
- Its execution marks the starting point of the program and controls the flow of execution.

Q.2 Explain the significance of the return type of the main function.

The return type of the main function in C++ is significant because it indicates the outcome of the program's execution to the operating system.
In most cases, the return type of main is int, which stands for an integer. The return value is used to convey whether the program ran successfully or encountered an error. By convention:
Return value of 0: Indicates successful execution of the program.
Return value other than 0: Indicates an error or abnormal termination.

Q.3 What are the two valid signatures of the main function in C++?

In C++, the main function is the entry point of a program, and it has two valid signatures:
1. int main()
This signature declares the main function with no parameters. It returns an integer value to indicate the program's exit status.

2. int main(int argc, char* argv[])
This signature declares the main function with two parameters:

- argc: an integer representing the number of command-line arguments passed to the program.
- argv: an array of character pointers representing the command-line arguments.

This signature is used when the program needs to process command-line arguments.

Notes
- The main function must return an integer value.
- The main function cannot be overloaded or redefined.
- The main function is called automatically by the C++ runtime environment when the program starts.

Q.4 What is function prototyping and why is it necessary in C++?

Function prototyping is a feature in C++ that allows you to declare a function before it is defined. A function prototype is a declaration of a function that includes its return type, name, and parameter list, but does not include its function body.

Function prototyping is necessary in C++ for several reasons:

1. **Forward Declaration**: Function prototyping allows you to declare a function before it is defined. This is useful when you want to use a function before its definition appears in the code.
2. **Function Overloading**: Function prototyping is necessary for function overloading, which is a feature in C++ that allows you to define multiple functions with the same name but different parameter lists.
3. **Error Checking**: Function prototyping helps the compiler to check for errors in the function call, such as mismatched parameter types or incorrect number of parameters.
4. **Compiler Optimization**: Function prototyping provides the compiler with information about the function's return type and parameter list, which can help the compiler to optimize the code.

Benefits of Function Prototyping
1. **Improved Code Organization**: Function prototyping allows you to organize your code in a more logical and readable way.
2. **Reduced Errors**: Function prototyping helps to reduce errors in the code by providing a clear declaration of the function's interface.
3. **Improved Compiler Diagnostics**: Function prototyping provides the compiler with more information about the function, which can help to improve the accuracy of compiler diagnostics.

Example of Function Prototyping

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

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

// Function call
int result = add(2, 3);
```

Q.5 How do you declare a function prototype for a function that returns an integer and takes two integer parameters?

To declare a function prototype for a function that returns an integer and takes two integer parameters, you can use the following syntax:

```
int functionName(int param1, int param2);
```

Here:
- int is the return type of the function, indicating that it returns an integer value.
- functionName is the name of the function.
- param1 and param2 are the names of the two integer parameters that the function takes.

For example:
```
int add(int x, int y);
```

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

Q.6 What happens if a function is used before it is prototyped?

If a function is used before it is prototyped, the compiler will typically generate an error message. This is because the compiler has not yet seen the function's prototype, which provides the necessary information about the function's return type, name, and parameter list.

Possible Compiler Errors
The specific error message generated by the compiler may vary depending on the compiler and the programming language being used. Some common error messages include:

- "Implicit declaration of function 'functionName'"
- "Undefined reference to 'functionName'"
- "Function 'functionName' used before declaration"

Why is Prototyping Necessary?
Prototyping is necessary because it provides the compiler with the necessary information about the function's interface. Without a prototype, the compiler has no way of knowing the function's return type, name, or parameter list.

How to Fix the Error
To fix the error, you can simply move the function prototype above the point where the function is first used. Alternatively, you can define the function before it is used, which also provides the necessary information to the compiler.

For example:

```
// Error: Function used before prototype
int main() {
    add(2, 3); // Error: Implicit declaration of function 'add'
    return 0;
}

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

To fix the error, you can add a function prototype above the main function:

```
// Fixed: Function prototype added
int add(int x, int y); // Function prototype

int main() {
    add(2, 3); // Okay: Function prototype seen by compiler
    return 0;
}

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

Q.7 What is the difference between a declaration and a definition of a function?
In programming, a declaration and a definition of a function are two related but distinct concepts:

Declaration
A function declaration, also known as a function prototype, is a statement that tells the compiler about the existence of a function, its name, return type, and parameter list. It does not provide the implementation details of the function.

Example of a Function Declaration

```
int add(int x, int y);
```

Definition
A function definition, on the other hand, provides the actual implementation of the function, including the function body and the code that gets executed when the function is called.

Example of a Function Definition

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

Key Differences
Here are the key differences between a function declaration and a definition:

1. Purpose: A declaration informs the compiler about the function's existence, while a definition provides the implementation details.
2. Syntax: A declaration typically includes the function name, return type, and parameter list, while a definition includes the function body.
3. Implementation: A declaration does not provide any implementation details, while a definition includes the actual code that gets executed.

Importance of Both
Both declarations and definitions are essential in programming:

1. Declarations help with code organization: By declaring functions before they are defined, you can organize your code in a more logical and readable way.
2. Definitions provide the implementation: Without definitions, your functions would not be able to perform any actual work.

In summary, a function declaration tells the compiler about the function's existence, while a function definition provides the implementation details. Both are crucial in programming, and they work together to help you create robust and maintainable code.

Q.8 How do you call a simple function that takes no parameters and returns void?
To call a simple function that takes no parameters and returns void, you can use the following syntax:

Syntax:

```
functionName();
```

Here:
- functionName is the name of the function you want to call.
- The parentheses () are empty, indicating that the function takes no parameters.

Example
Suppose you have a function named greet that takes no parameters and returns void:

```
void greet() {
    printf("Hello, World!\n");
}
```

To call this function, you would use the following statement:

```
greet();
```

This would execute the greet function, printing "Hello, World!" to the console.

Q.9 Explain the concept of "scope" in the context of functions.
In the context of functions, "scope" refers to the region of the code where a variable or function is defined and can be accessed.

Types of Scope
There are two main types of scope:

1. Local Scope: Variables defined within a function have local scope and can only be accessed within that function.
2. Global Scope: Variables defined outside of any function have global scope and can be accessed from any function.

Characteristics of Scope

1. Variable Visibility: Variables are only visible within their scope.
2. Variable Lifetime: Variables are created when their scope is entered and destroyed when their scope is exited.
3. Access Control: Variables can only be accessed within their scope.

Example
Suppose we have the following code:

```
int x = 10; // global variable

void func() {
    int y = 20; // local variable
    printf("%d %d\n", x, y);
}

int main() {
    func();
    printf("%d\n", x);
    // printf("%d\n", y); // error: y is not in scope
```

```
    return 0;
}
```

In this example:
- x has global scope and can be accessed from both func and main.
- y has local scope and can only be accessed within func.

Benefits of Scope

1. Variable Hiding: Variables with the same name can be defined in different scopes without conflicts.
2. Code Organization: Scope helps to organize code into logical units, making it easier to understand and maintain.
3. Memory Management: Scope helps to manage memory by automatically creating and destroying variables when their scope is entered or exited.

Q.10 What is call by reference in C++?
In C++, "call by reference" is a method of passing variables to functions where the actual variable is passed, not a copy of it. This means that any changes made to the variable within the function will affect the original variable.

How Call by Reference Works
When a variable is passed by reference to a function:

1. Address is passed: The memory address of the variable is passed to the function, not a copy of the variable.
2. Function receives a reference: The function receives a reference to the original variable, allowing it to access and modify it directly.

Syntax for Call by Reference
To pass a variable by reference in C++, you can use the following syntax:

```
void functionName(int &variableName) {
    // function code
}
```

In this syntax:
- & is the address-of operator, which indicates that the variable is being passed by reference.
- variableName is the name of the variable being passed.

Example of Call by Reference

```
void swap(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5;
    int b = 10;
```

```
    swap(a, b);
    cout << "a = " << a << endl; // outputs: a = 10
    cout << "b = " << b << endl; // outputs: b = 5
    return 0;
}
```

In this example:
- The swap function takes two integers by reference.
- The function swaps the values of the two integers.
- The changes made to the variables within the function affect the original variables in the main function.

Benefits of Call by Reference
1. Efficient memory use: Only the memory address of the variable is passed, not a copy of the variable.
2. Changes are reflected: Changes made to the variable within the function are reflected in the original variable.
3. Improved performance: Call by reference can be faster than call by value, especially for large variables.

Q.11 How does call by reference differ from call by value?
Call by reference and call by value are two different methods of passing variables to functions in programming.

Call by Value
In call by value:

1. Copy of variable is passed: A copy of the variable is passed to the function.
2. Function receives a copy: The function receives a copy of the variable, not the original variable.
3. Changes are not reflected: Changes made to the variable within the function do not affect the original variable.

Call by Reference
In call by reference:

1. Address of variable is passed: The memory address of the variable is passed to the function.
2. Function receives a reference: The function receives a reference to the original variable.
3. Changes are reflected: Changes made to the variable within the function affect the original variable.

Key Differences
1. Variable passing: Call by value passes a copy of the variable, while call by reference passes the address of the variable.
2. Function receiving: Call by value functions receive a copy of the variable, while call by reference functions receive a reference to the original variable.
3. Changes reflection: Call by value changes are not reflected in the original variable, while call by reference changes are reflected.

Example
Here's an example to illustrate the difference:

```
void swapValue(int x, int y) {
    int temp = x;
    x = y;
    y = temp;
```

```
}

void swapReference(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5;
    int b = 10;

    swapValue(a, b);
    cout << "a = " << a << endl; // outputs: a = 5
    cout << "b = " << b << endl; // outputs: b = 10

    swapReference(a, b);
    cout << "a = " << a << endl; // outputs: a = 10
    cout << "b = " << b << endl; // outputs: b = 5

    return 0;
}
```

In this example:
- swapValue is a call by value function, where changes are not reflected in the original variables.
- swapReference is a call by reference function, where changes are reflected in the original variables.

Q.12 Provide an example of a function that uses call by reference to swap two integers.
Here's an example of a function that uses call by reference to swap two integers:

```
#include <iostream>

// Function to swap two integers using call by reference
void swap(int &x, int &y) {
    int temp = x;
    x = y;
    y = temp;
}

int main() {
    int a = 5;
    int b = 10;

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

    // Call the swap function
    swap(a, b);

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

```
    return 0;
}
```

Output
Before swap: a = 5, b = 10
After swap: a = 10, b = 5

Explanation

1. The swap function takes two integer references x and y as parameters.
2. Inside the swap function, a temporary variable temp is used to hold the value of x.
3. The value of y is assigned to x, and the value of temp (which is the original value of x) is assigned to y. This effectively swaps the values of x and y.
4. In the main function, two integers a and b are declared and initialized with values 5 and 10, respectively.
5. The swap function is called with a and b as arguments, which swaps their values.
6. Finally, the values of a and b are printed before and after the swap operation to demonstrate the effect of the swap function.

Q.13 What is an inline function in C++?
In C++, an inline function is a function that is expanded in-line at the point of invocation, rather than being called as a separate function. This means that the compiler replaces the function call with the actual code of the function.

Benefits of Inline Functions

1. Improved performance: By eliminating the overhead of a function call, inline functions can improve performance.
2. Reduced code size: In some cases, the expanded code of an inline function can be smaller than the code required to make a function call.
3. Better optimization: Because the compiler has access to the expanded code, it can perform better optimization.

Syntax for Inline Functions

```
inline return-type function-name(parameters) {
    // function body
}
```

Example of an Inline Function

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

When to Use Inline Functions

1. Small functions: Functions with a small number of lines of code are good candidates for inlining.
2. Frequently called functions: Functions that are called frequently can benefit from inlining.
3. Performance-critical code: Inlining can be used to optimize performance-critical code.

Limitations of Inline Functions

1. Code bloat: Excessive use of inline functions can lead to code bloat.
2. Debugging difficulties: Because inline functions are expanded in-line, debugging can be more difficult.
3. Link-time errors: Inline functions can cause link-time errors if they are defined multiple times.

Q.14 How do inline functions improve performance?
Inline functions can improve performance in several ways:

1. Reduced Function Call Overhead
When a function is called, the following operations occur:

- The current instruction pointer is saved.
- The function's parameters are pushed onto the stack.
- The function's return address is pushed onto the stack.
- The function is executed.

By inlining a function, these operations are eliminated, resulting in a reduction in overhead.

2. Elimination of Return Instruction
When a function returns, the return instruction is executed, which involves:

- Restoring the previous instruction pointer.
- Popping the return address from the stack.

By inlining a function, the return instruction is eliminated.

3. Improved Instruction-Level Parallelism
Modern CPUs can execute multiple instructions in parallel. By inlining a function, the instructions of the inlined function can be interleaved with the instructions of the calling function, improving instruction-level parallelism.

4. Reduced Cache Misses
When a function is called, the CPU may need to fetch the function's instructions from memory, resulting in a cache miss. By inlining a function, the instructions of the inlined function are already in the cache, reducing the likelihood of a cache miss.

5. Better Optimization Opportunities
When a function is inlined, the compiler has more opportunities to optimize the code, such as:

- Dead code elimination.
- Constant folding.
- Register allocation.

Example
Consider the following example:

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

```
int main() {
    int result = add(2, 3);
    return 0;
}
```

If the add function is inlined, the main function becomes:

```
int main() {
    int result = 2 + 3;
    return 0;
}
```

In this example, the function call overhead is eliminated, and the instructions of the add function are interleaved with the instructions of the main function, improving instruction-level parallelism.

Q.15 Explain the syntax for declaring an inline function.
The syntax for declaring an inline function in C++ is:

```
inline return-type function-name(parameters) {
    // function body
}
```

Here:
- inline is the keyword used to declare an inline function.
- return-type is the data type of the value returned by the function.
- function-name is the name of the function.
- parameters is the list of parameters passed to the function.
- function body is the code that is executed when the function is called.

Example
Here is an example of declaring an inline function:

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

In this example:
- inline is the keyword used to declare the max function as an inline function.
- int is the return type of the function.
- max is the name of the function.
- (int a, int b) is the parameter list, which takes two integers a and b.
- The function body returns the maximum of a and b using the ternary operator.

Note that the inline keyword is only a hint to the compiler, and the compiler may choose to ignore it and not inline the function.

Q.16 What are macros in C++ and how are they different from inline functions?
Macros and inline functions are two different ways to reuse code in C++.

Macros
Macros are a preprocessor feature that allows you to define a block of code that can be expanded in-place wherever it's used. Macros are essentially a text replacement mechanism.

Example of a Macro

```
#define SQUARE(x) ((x) * (x))

int main() {
    int result = SQUARE(5);
    return 0;
}
```

In this example:
- The SQUARE macro takes an argument x and expands to the expression ((x) * (x)).
- When the macro is used in the main function, it's expanded in-place to ((5) * (5)).

Inline Functions
Inline functions, on the other hand, are a feature of the C++ language that allows you to define small functions that can be expanded in-line by the compiler.

Example of an Inline Function

```
inline int square(int x) {
    return x * x;
}

int main() {
    int result = square(5);
    return 0;
}
```

In this example:
- The square function is defined with the inline keyword, indicating that it should be expanded in-line by the compiler.
- When the function is called in the main function, the compiler expands it in-line to the equivalent code.

Key Differences
1. Preprocessor vs. Compiler: Macros are expanded by the preprocessor, while inline functions are expanded by the compiler.
2. Text Replacement vs. Code Generation: Macros perform a simple text replacement, while inline functions generate actual code that's optimized by the compiler.
3. Type Safety: Inline functions provide type safety, as the compiler checks the types of the arguments and return value. Macros do not provide type safety, as they're simply a text replacement mechanism.
4. Debugging: Inline functions are generally easier to debug, as the compiler provides more information about the expanded code. Macros can be more difficult to debug, as the preprocessor expands them before the compiler sees the code.

In general, inline functions are a safer and more flexible way to reuse code in C++, while macros should be used with caution and only when necessary.

Q.17 Explain the advantages and disadvantages of using macros over inline functions.
Macros and inline functions are both used to reuse code, but they have different advantages and disadvantages.

Advantages of Macros
1. Flexibility: Macros can perform more complex operations, such as string manipulation and conditional compilation.
2. Generic Programming: Macros can be used to implement generic programming, where the same code can work with different data types.
3. No Function Call Overhead: Macros are expanded in-place, so there is no function call overhead.

Disadvantages of Macros
1. No Type Safety: Macros do not provide type safety, as they are simply a text replacement mechanism.
2. No Debugging Information: Macros can make debugging more difficult, as the preprocessor expands them before the compiler sees the code.
3. Namespace Pollution: Macros can pollute the namespace, as they can define new names that conflict with existing names.
4. Difficulty in Maintenance: Macros can be difficult to maintain, as they can be complex and difficult to understand.

Advantages of Inline Functions
1. Type Safety: Inline functions provide type safety, as the compiler checks the types of the arguments and return value.
2. Debugging Information: Inline functions provide debugging information, as the compiler generates debugging symbols for the expanded code.
3. Namespace Safety: Inline functions do not pollute the namespace, as they are defined within a specific scope.
4. Easy Maintenance: Inline functions are easy to maintain, as they are defined using standard C++ syntax.

Disadvantages of Inline Functions
1. Limited Flexibility: Inline functions are limited in their flexibility, as they can only perform operations that are valid within a function.
2. Function Call Overhead: Inline functions can still incur function call overhead, although this can be mitigated by using the inline keyword.
3. Code Bloat: Inline functions can cause code bloat, as the expanded code can increase the size of the executable.

Conclusion
In general, inline functions are preferred over macros, as they provide type safety, debugging information, and namespace safety. However, macros can still be useful in certain situations, such as when implementing generic programming or performing complex operations. Ultimately, the choice between macros and inline functions depends on the specific requirements of the project.

Q.18 Provide an example to illustrate the differences between macros and inline functions.
Here's an example that illustrates the differences between macros and inline functions:

Example
Suppose we want to implement a simple square function that takes an integer as input and returns its square.

Macro Implementation

```
#define SQUARE(x) ((x) * (x))

int main() {
    int result = SQUARE(5);
    printf("%d\n", result); // outputs: 25

    int result2 = SQUARE(5 + 1);
    printf("%d\n", result2); // outputs: 11 ( incorrect result! )
    return 0;
}
```

Inline Function Implementation

```
inline int square(int x) {
    return x * x;
}

int main() {
    int result = square(5);
    printf("%d\n", result); // outputs: 25

    int result2 = square(5 + 1);
    printf("%d\n", result2); // outputs: 36 ( correct result! )
    return 0;
}
```

Differences Illustrated
This example illustrates the following differences between macros and inline functions:

1. Order of Operations: In the macro implementation, the expression 5 + 1 is not evaluated before being passed to the SQUARE macro. This results in an incorrect result. In contrast, the inline function implementation evaluates the expression 5 + 1 before passing it to the square function, resulting in a correct result.
2. Type Safety: The inline function implementation provides type safety, as the square function takes an int parameter and returns an int value. In contrast, the macro implementation does not provide type safety, as it simply expands to a block of code that can be used with any type.
3. Debugging Information: The inline function implementation provides debugging information, as the square function is a separate entity that can be debugged independently. In contrast, the macro implementation does not provide debugging information, as the SQUARE macro is simply a block of code that is expanded in-place.

In summary, this example illustrates the benefits of using inline functions over macros, including order of operations, type safety, and debugging information.

Q.19 What is function overloading in C++?

Function overloading is a feature in C++ that allows multiple functions with the same name to be defined, as long as they have different parameter lists. This means that a function can have multiple definitions, each with a unique set of parameters.

Example of Function Overloading

```
#include <iostream>

void print(int x) {
    std::cout << "Integer: " << x << std::endl;
}

void print(double x) {
    std::cout << "Double: " << x << std::endl;
}

void print(char x) {
    std::cout << "Character: " << x << std::endl;
}

int main() {
    print(10);      // Outputs: Integer: 10
    print(3.14);   // Outputs: Double: 3.14
    print('A');    // Outputs: Character: A
    return 0;
}
```

In this example, the print function is overloaded to accept different types of parameters: int, double, and char. The correct function to call is determined by the type of argument passed to it.

Benefits of Function Overloading

1. Improved Code Readability: Function overloading allows multiple functions with the same name to be defined, making the code more readable and easier to understand.
2. Increased Flexibility: Function overloading enables functions to handle different types of data, making them more flexible and reusable.
3. Reduced Code Duplication: By allowing multiple functions with the same name, function overloading reduces code duplication and makes maintenance easier.

Rules for Function Overloading

1. Function Name: The functions must have the same name.
2. Parameter List: The functions must have different parameter lists, which can differ in the number or types of parameters.
3. Return Type: The return type of the functions can be different, but it is not a requirement for overloading.

By following these rules and using function overloading effectively, you can write more flexible, readable, and maintainable code in C++.

Q.20 How does the compiler differentiate between overloaded functions?
The compiler differentiates between overloaded functions by examining the number and types of arguments passed to the function. This process is called function signature matching or overload resolution.

Steps Involved in Overload Resolution

1. Candidate Functions: The compiler creates a list of candidate functions that match the function name.
2. Parameter Matching: The compiler checks the parameter list of each candidate function to see if it matches the arguments passed to the function.
3. Conversion Sequences: If there are no exact matches, the compiler checks for possible conversion sequences, such as promotions or conversions, that can be applied to the arguments to match the parameter list.
4. Best Match: The compiler selects the best match based on the parameter matching and conversion sequences.
5. Error Handling: If there are multiple best matches or no matches at all, the compiler reports an error.

Example of Overload Resolution

```
void print(int x) {
    std::cout << "Integer: " << x << std::endl;
}

void print(double x) {
    std::cout << "Double: " << x << std::endl;
}

void print(char x) {
    std::cout << "Character: " << x << std::endl;
}

int main() {
    print(10);      // Matches print(int x)
    print(3.14);    // Matches print(double x)
    print('A');     // Matches print(char x)
    print(10.5);   // Matches print(double x) after conversion
    return 0;
}
```

In this example:
- print(10) matches print(int x) exactly.
- print(3.14) matches print(double x) exactly.
- print('A') matches print(char x) exactly.
- print(10.5) matches print(double x) after converting the double literal to a double parameter.

The compiler differentiates between these overloaded functions by examining the parameter lists and applying conversion sequences as needed.

Q.21 Provide an example of overloaded functions in C++.
Here's an example of overloaded functions in C++:

```
#include <iostream>

// Overloaded function to calculate the area of a rectangle
int calculateArea(int length, int width) {
    return length * width;
}

// Overloaded function to calculate the area of a circle
double calculateArea(double radius) {
    const double pi = 3.14159;
    return pi * radius * radius;
}

// Overloaded function to calculate the area of a triangle
double calculateArea(double base, double height) {
    return 0.5 * base * height;
}

int main() {
    // Calculate the area of a rectangle
    int rectangleLength = 5;
    int rectangleWidth = 3;
    int rectangleArea = calculateArea(rectangleLength, rectangleWidth);
    std::cout << "Rectangle Area: " << rectangleArea << std::endl;

    // Calculate the area of a circle
    double circleRadius = 4.0;
    double circleArea = calculateArea(circleRadius);
    std::cout << "Circle Area: " << circleArea << std::endl;

    // Calculate the area of a triangle
    double triangleBase = 6.0;
    double triangleHeight = 8.0;
    double triangleArea = calculateArea(triangleBase, triangleHeight);
    std::cout << "Triangle Area: " << triangleArea << std::endl;

    return 0;
}
```

In this example:

- We define three overloaded functions called calculateArea, each with a different parameter list:
    - One function takes two int parameters (length and width) to calculate the area of a rectangle.
    - Another function takes a single double parameter (radius) to calculate the area of a circle.
    - The third function takes two double parameters (base and height) to calculate the area of a triangle.
- In the main function, we call each of the overloaded calculateArea functions with the corresponding parameters and print the results.

This example demonstrates how function overloading allows us to define multiple functions with the same name but different parameter lists, making the code more flexible and easier to use.

Q.22 What are default arguments in C++?
In C++, default arguments are values that are assigned to function parameters when the function is called without providing explicit values for those parameters.
Syntax

```
return-type function-name(parameter1, parameter2 = default-value2, ...) {
   // function body
}
```

Example

```
#include <iostream>

void printMessage(const std::string& message, bool uppercase = false) {
   if (uppercase) {
      for (char& c : message) {
         c = std::toupper(c);
      }
   }
   std::cout << message << std::endl;
}

int main() {
   printMessage("Hello, World!");  // Output: Hello, World!
   printMessage("Hello, World!", true);  // Output: HELLO, WORLD!
   return 0;
}
```

In this example:
- The printMessage function takes two parameters: message and uppercase.
- The uppercase parameter has a default value of false.
- When printMessage is called without providing a value for uppercase, it defaults to false.
- When printMessage is called with an explicit value for uppercase, that value is used instead of the default.

Benefits of Default Arguments

1. Increased flexibility: Default arguments allow functions to be used in different contexts without requiring multiple function definitions.
2. Improved code readability: By providing default values for parameters, functions can be called with fewer arguments, making the code easier to read.
3. Reduced code duplication: Default arguments can help reduce code duplication by allowing a single function to handle different scenarios.

Best Practices for Using Default Arguments
1. Use default arguments sparingly: Default arguments should be used only when it makes sense for the function to have a default behavior.
2. Choose default values carefully: Default values should be chosen based on the most common use case for the function.
3. Document default arguments: Clearly document the default arguments and their values to avoid confusion.

Q.23 How do you specify default arguments in a function declaration?
In C++, you can specify default arguments in a function declaration by using the assignment operator (=) followed by the default value.

Syntax

```
return-type function-name(parameter1, parameter2 = default-value2, ...) {
    // function body
}
```

Example

```
#include <iostream>

void printMessage(const std::string& message, bool uppercase = false, int repetitions = 1) {
    for (int i = 0; i < repetitions; ++i) {
        if (uppercase) {
            for (char& c : message) {
                c = std::toupper(c);
            }
        }
        std::cout << message << std::endl;
    }
}

int main() {
    printMessage("Hello, World!");  // Output: Hello, World!
    printMessage("Hello, World!", true);  // Output: HELLO, WORLD!
    printMessage("Hello, World!", false, 3);  // Output: Hello, World! (3 times)
    return 0;
}
```

In this example:

- The printMessage function takes three parameters: message, uppercase, and repetitions.
- The uppercase parameter has a default value of false.
- The repetitions parameter has a default value of 1.
- When printMessage is called without providing values for uppercase and repetitions, they default to false and 1, respectively.

Rules for Default Arguments

1. Default arguments must be specified from right to left: In a function declaration, default arguments must be specified from right to left. Once a default argument is specified, all subsequent arguments to the right must also have default arguments.
2. Default arguments can be specified only once: A function can have multiple declarations, but default arguments can be specified only once.
3. Default arguments are not stored: Default arguments are not stored in memory; instead, they are substituted into the function call at compile-time.

Q.24 What are the rules for using default arguments in functions?
When using default arguments in functions, follow these rules:

Rule 1: Default Arguments Must Be Specified from Right to Left
In a function declaration, default arguments must be specified from right to left. Once a default argument is specified, all subsequent arguments to the right must also have default arguments.

Example

```
void printMessage(const std::string& message, bool uppercase = false, int repetitions = 1) {
   // function body
}
```

In this example, uppercase and repetitions have default arguments, and they are specified from right to left.

Rule 2: Default Arguments Can Be Specified Only Once
A function can have multiple declarations, but default arguments can be specified only once.

Example

```
void printMessage(const std::string& message, bool uppercase = false);  // default argument specified here
void printMessage(const std::string& message, bool uppercase) {  // error: default argument specified
again
   // function body
}
```

In this example, the default argument for uppercase is specified only once in the first declaration.

Rule 3: Default Arguments Are Not Stored
Default arguments are not stored in memory; instead, they are substituted into the function call at compile-time.

Example

```
void printMessage(const std::string& message, bool uppercase = false) {
   if (uppercase) {
      // modify message
   }
   std::cout << message << std::endl;
}
```

In this example, the default argument for uppercase is substituted into the function call at compile-time, and the function body is executed accordingly.

Rule 4: Avoid Using Default Arguments for Overloaded Functions
When using default arguments with overloaded functions, the default arguments can lead to ambiguity.

Example

```
void printMessage(const std::string& message, bool uppercase = false);
void printMessage(const std::string& message, int repetitions) {
    // function body
}
```

In this example, the default argument for uppercase can lead to ambiguity when calling the printMessage function.

By following these rules, you can effectively use default arguments in your functions and avoid potential pitfalls.

Q.25 Provide an example of a function with default arguments.
Here's an example of a function with default arguments:

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

// Function to print a message with optional uppercase and repetition
void printMessage(const std::string& message, bool uppercase = false, int repetitions = 1) {
    std::string modifiedMessage = message;

    // Convert message to uppercase if required
    if (uppercase) {
        for (char& c : modifiedMessage) {
            c = std::toupper(c);
        }
    }

    // Print the message for the specified number of repetitions
    for (int i = 0; i < repetitions; ++i) {
        std::cout << modifiedMessage << std::endl;
    }
}

int main() {
    // Call the function with default arguments
    printMessage("Hello, World!");

    // Call the function with uppercase argument
    printMessage("Hello, World!", true);

    // Call the function with repetition argument
    printMessage("Hello, World!", false, 3);

    // Call the function with all arguments
    printMessage("Hello, World!", true, 2);

    return 0;
}
```

In this example:

- The printMessage function takes three parameters: message, uppercase, and repetitions.
- The uppercase parameter has a default value of false.
- The repetitions parameter has a default value of 1.
- The function modifies the message to uppercase if required and prints it for the specified number of repetitions.
- In the main function, we demonstrate different ways to call the printMessage function using its default arguments.