**Assignment 2 (Theory)**

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

Ans The main function is the entry point of a C++ program. When a program is executed, execution starts from main(). It controls the flow of the program and usually returns an integer value to the operating system.

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

Ans The return type of main is typically int, which indicates that the program should return a status code to the operating system. A return value of 0 means successful execution, while a nonzero value indicates an error.

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

Ans The two valid signatures of main() in C++ are:

* int main()
* int main(int argc, char \*argv[])

The second version allows command-line arguments to be passed to the program.

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

Ans Function prototyping declares a function before its definition to tell the compiler about its return type and parameters.

It is necessary because it prevents errors when calling functions before their definitions.

Example:

int add(int, int); // Function prototype

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

Ans int add(int, int);

This tells the compiler that add takes two integers as parameters and returns an integer.

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

Ans If a function is used before being declared, the compiler may generate an error. In older versions of C++, the compiler assumed a default return type of int, but this is no longer allowed in modern C++.

7. What is the difference between a declaration and a definition of a function?

Ans difference between a declaration and a definition of a function:

| **Feature** | **Function Declaration** | **Function Definition** |
| --- | --- | --- |
| **Purpose** | Tells the compiler about the function | Provides the actual implementation |
| **Includes Body** | No  int add(int, int); | Yes  int add(int a, int b) { return a + b; } |
| **Ends with ;** | Yes | No |
| **Mandatory** | Optional if function is defined before use | Yes, must be defined exactly once |

Top of Form

Bottom of Form

8. How do you call a simple function that takes no parameters and returns void?

Ans void greet() {

cout << "Hello!" << endl;

}

int main() {

greet(); // Function call

}

9. Explain the concept of "scope" in the context of functions.

Ans Scope determines where a variable can be accessed

* Local scope: Inside a function/block.
* Global scope: Outside all functions, accessible anywhere in the program.

Example:

int x = 10; // Global variable

void func() {

int y = 5; // Local variable

}

10. What is call by reference in C++?

Ans Call by reference means passing arguments to a function by reference using &. This allows the function to modify the original variables.

Example

void modify(int &x) {

x = x + 10;

}

11. How does call by reference differ from call by value?

Ans Heres the difference:

| **Feature** | **Call by Value** | **Call by Reference** |
| --- | --- | --- |
| **What is passed** | A **copy** of the actual argument is passed to the function. | The **original variable’s reference** is passed. |
| **Effect on original value** | Changes made inside the function **do not affect** the original value. | Changes made inside the function **affect** the original variable. |
| **Memory usage** | More memory (due to creating copies). | Less memory (direct access to original variable). |
| **Syntax** | void modify(int x);  modify(a); | void modify(int &x);  modify(a); |

12. Provide an example of a function that uses call by reference to swap two integers.

Ans The complete code is as follows:

#include <iostream>

using namespace std;

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

int temp = a;

a = b;

b = temp;

}

int main() {

int x = 5, y = 10;

cout << "Before swap: x = " << x << ", y = " << y << endl;

swap(x, y); // Call by reference

cout << "After swap: x = " << x << ", y = " << y << endl;

return 0;

}

Output:

Before swap: x = 5, y = 10

After swap: x = 10, y = 5

13. What is an inline function in C++?

Ans when a function is declared as inline or in Inline function, The compiler replaces the function call with the actual code of the function during compilation. This eliminates the overhead of a function call (such as saving registers, pushing to the stack, jumping to function code, and returning).

14. How do inline functions improve performance?

Ans Inline functions can improve performance by avoiding the overhead of function calls. However, overuse may lead to code bloat and longer compile times.

15. Explain the syntax for declaring an inline function.

Ans An inline function is declared by using the inline keyword before the function definition.

**Syntax:**

inline return\_type function\_name(parameter\_list) {

// function body

}

16. What are macros in C++ and how are they different from inline functions?

Ans A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. Macros are defined using the #define directive.

Syntax:

#define macro\_name replacement\_text

Difference between them:

| **Feature** | **Macros** | **Inline Functions** |
| --- | --- | --- |
| **Defined Using** | #define preprocessor directive | inline keyword before function definition |
| **Type Safety** | No type checking | Type checking is performed |
| **Syntax** | #define SQUARE(x) ((x) \* (x)) | inline int square(int x) { return x \* x; } |
| **Handled By** | Preprocessor (before compilation) | Compiler (during compilation) |
| **Debugging Support** | Hard to debug | Easier to debug |
| **Code Replacement** | Simple text replacement | Replaces with actual function code (if compiler allows) |
| **Scope** | Global (no namespace or scope) | Follows C++ scoping rules |

17. Explain the advantages and disadvantages of using macros over inline functions.

Ans **Advantages of Macros over Inline Functions:**

1. **Simplicity**: Macros are simple to define and use, especially for short expressions or constants.
2. **Preprocessor-level substitution**: Since macros are replaced before compilation, they might seem faster in very basic scenarios.
3. **No function overhead**: There’s no actual function call for a macro, which saves the overhead of stack operations (like inline functions).

**Disadvantages of Macros over Inline Functions:**

1. **No Type Checking**: Macros are just text substitutions. The compiler doesn’t check types, which can lead to **unexpected behavior or hard-to-find bugs**.
2. **No Scope Control**: Macros are globally substituted; they don’t follow the C++ scope or namespace rules.
3. **Multiple Evaluations**: Macros may evaluate arguments multiple times. For example:  
   #define SQUARE(x) ((x) \* (x)) → SQUARE(a++) becomes ((a++) \* (a++)), which is incorrect.
4. **Hard to Debug**: Errors involving macros can be hard to trace, since the compiler only sees the expanded code.
5. **No Overloading**: Macros don’t support overloading, while inline functions do.

18. Provide an example to illustrate the differences between macros and inline functions.

Ans Let’s look at a realistic example showing how macros and inline functions behave differently:

**Macro Example:**

#include <iostream>

Using namespace std;

#define SQUARE(x) ((x) \* (x))

int main() {

int a = 5;

cout << "Macro SQUARE: " << SQUARE(a++) <<endl;

cout << "Final a: " << a <<endl;

return 0;

}

Output:

Macro SQUARE: 36

Final a: 7

Problem:

SQUARE(a++) becomes ((a++) \* (a++))

So a is incremented twice, which causes unexpected behavior

**Inline Function Example:**

#include <iostream>

Using namespace std;

inline int square(int x) {

return x \* x;

}

int main() {

int a = 5;

cout << "Inline square: " << square(a++) <<endl;

cout << "Final a: " << a <<endl;

return 0;

}

Output:

Inline square: 25

Final a: 6

Why it works:

* The inline function evaluates a++ only once
* It is type-safe and avoids side effects
* Also easier to debug and maintain

Conclusion:

This example clearly shows the danger of macros (especially with expressions as arguments) and the safety and reliability of inline functions.

19. What is function overloading in C++?

Ans Function overloading allows multiple functions with the same name but different parameter lists (type, number, or order) in the same scope.

Example:

int square(int x) { return x \* x; }

double square(double x) { return x \* x; }

* square(4) calls the int version
* square(4.5) calls the double version

20. How does the compiler differentiate between overloaded functions?

Ans The compiler uses name mangling to give each overloaded function a unique internal name based on its parameter types and order.

For example:

sum(int, int) becomes \_Z3sumii internally.

This helps the compiler identify functions even if they have the same name. Return type is not considered for overloading.

21. Provide an example of overloaded functions in C++.

Ans Example:

#include <iostream>

using namespace std;

void show() {

cout << "No parameters" << endl;

}

void show(int a) {

cout << "Integer: " << a << endl;

}

void show(float b) {

cout << "Float: " << b << endl;

}

int main() {

show(); // Calls show()

show(5); // Calls show(int)

show(3.14); // Calls show(float)

return 0;

}

**Output:**

No parameters

Integer: 5

Float: 3.14

Here,all functions have the same name: show But the parameters are different: (), (int), (float) and the compiler chooses the correct function based on arguments

22. What are default arguments in C++?

Ans Default arguments are values given to function parameters in advance, so if the caller doesn’t pass a value, the default is used.

Example:

void greet(string name = "User");

greet(); // Output: Hello, User!

greet("Raj"); // Output: Hello, Raj!

23. How do you specify default arguments in a function declaration?

Ans You assign values to parameters in the function declaration.

Example:

void show(int x, int y = 5);

Now y is optional — if not passed, it uses 5 by default.

24. What are the rules for using default arguments in functions?

Ans Rules for Using Default Arguments in C++

* Default arguments must be given from right to left (i.e., trailing parameters).

void func(int x, int y = 10); // Valid

void func(int x = 10, int y); // Invalid

* Default values are usually specified in the declaration (prototype) — not required again in the definition.

void show(int x = 5); // Declaration

void show(int x) { // Definition

* You can omit arguments from the right only.

void test(int a = 1, int b = 2, int c = 3);

test(10); // a = 10, b = 2, c = 3

test(10, 20); // a = 10, b = 20, c = 3

25. Provide an example of a function with default arguments.

Ans #include <iostream>

using namespace std;

// Function with default arguments

void display(string name = "Guest", int age = 18) {

cout << "Name: " << name << ", Age: " << age << endl;

}

int main() {

display(); // Uses both default values

display("Anjali"); // Uses default age

display("Rahul", 21); // Uses both given values

return 0;

}

Output:

Name: Guest, Age: 18

Name: Anjali, Age: 18

Name: Rahul, Age: 21