ASSIGNMENT – 02

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

Ans. In C++ the main function serves as the entry point of any C++ program. It is the point at which execution of program is started, and the operating system calls this function to start running a C++ code. Every C++ program must have the main function; otherwise the compiler will raise an error.

~Syntax:

void main( ){

……………….

………………..

}

In above syntax:

* Void: Void is a keyword in C++ language, void means nothing, whenever we use void as a function return type then that function nothing return, here main() function no return any value.
* In place of void we can also use int return type of main() function, at that time main() return integer type value.
* Main: Is a name of function which is predefined function in C++ library.

~Example:

#include<iostream>

void main( ) {

cout<< “This is main function.”

}

~Output:

This is main function.

Que.2: Explain the significance of the return type of the main function?

Ans. In C++, the int return type of the main function signifies the program’s exit status, indicating success or failure, is returned to the operating system, with 0 typically representing success.

In C++, if you don’t explicitly return a value from main, the compiler will automatically insert return0.

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

Ans. The two valid signatures for the main function in C++ are int main( ) and int main(int argc, char\*argv[ ]).

* Int main( ): This signature indicates that the main function takes no arguments and returns an integer value.
* int main(int argc, char\*argv[ ]):

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

Ans. **Function prototyping:** is the declaration of a function that tells the compiler about the function's name, return type, and parameters before its actual definition appears in the code.

 **Early Declaration**: It allows the compiler to check for correct usage of the function before its definition is encountered.

 **Type Checking**: Ensures the correct number and type of arguments are passed to the function.

 **Helps with Organization**: Makes programs easier to structure, especially when defining functions after the main() function.

Syntax:- return\_type function\_name(parameter\_list);

Example:- #include <iostream>

using namespace std;

// Function prototype

int add(int, int);

int main() {

int result = add(5, 3); // Now the compiler knows about 'add'

cout << "Result: " << result;

return 0;

}

// Function definition

int add(int a, int b) {

return a + b;

}

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

Ans. To declare a **function prototype** for a function that:

* **returns an int**
* **takes two int parameters**
* **Syntax:** int functionName(int, int);

Que.6: What happens if a function is used before it is prototype?

Ans. If a function is **used before its prototype or definition** in C++, the compiler doesn't know anything about the function — its name, return type, or parameters — which can lead to a **compiler error or incorrect behavior**.

Que.7: What is the Difference between function declaration and function definition:

| **Aspect** | **Function Declaration** | **Function Definition** |
| --- | --- | --- |
| **What it does** | Tells the compiler about the function's name, return type, and parameters | Provides the actual body (code) of the function |
| **Also called** | Function **prototype** | Function **implementation** |
| **Contains body?** | No | Yes |
| **Ends with** | A semicolon (;) | A block of code { ... } |
| **Purpose** | To declare a function before it's used (especially if it's defined later) | To actually perform the task the function is supposed to do |

#include <iostream>

using namespace std;

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

int main() {

cout << add(3, 4); // Function is called here

return 0;

}

int add(int a, int b) { // Definition

return a + b;

}

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

Ans. #include <iostream>

using namespace std;

// Function declaration (optional if defined before main)

void greet();

int main() {

greet(); // ← Function call

return 0;

}

// Function definition

void greet() {

cout << "Hello, world!" << endl;

}

Que.9: Explain the concept of “Scope” in the context of function?

Ans. **Scope** refers to the **region of a program** where a **variable or function is accessible** or "visible".

| **Type of Scope** | **Description** |
| --- | --- |
| **Local Scope** | Variables declared **inside a function** are only accessible within that function. |
| **Global Scope** | Variables declared **outside all functions** are accessible throughout the file (unless shadowed). |
| **Function Scope** | The name of the function itself is available only in its containing scope (usually global). |
| **Block Scope** | Variables declared within {} blocks (like loops or if statements inside functions) are accessible only within those blocks. |

**Example:-** #include <iostream>

using namespace std;

int globalVar = 10; // Global scope

void myFunction() {

int localVar = 5; // Local to myFunction

cout << "Inside function: " <<localVar << ", " << globalVar << endl;

}

int main() {

myFunction();

// cout << localVar; // Error: localVar is not in scope here

cout << "Inside main: " << globalVar << endl;

return 0;

**}**

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

Ans. **Call by reference** means passing the **actual memory address** of a variable to a function, so the function can **modify the original variable** directly.

**Key Features:-**

| **Feature** | **Description** |
| --- | --- |
| **Access original data** | **Changes made inside the function affect the original variable.** |
| **Uses references** | **Parameters are passed using & (reference symbol).** |
| **More efficient** | **No copies are made, saving memory and time.** |

**Syntax:** void functionName(int &x);

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

Ans.

| **Feature** | **Call by Value** | **Call by Reference** |
| --- | --- | --- |
| **What is passed** | A **copy** of the actual variable | The **actual variable (via reference)** |
| **Function modifies original?** | No — original remains unchanged | Yes — changes affect the original variable |
| **Memory usage** | More (copies data) | Less (passes address/reference) |
| **Safety** | Safer (original data is protected) | Riskier (original data can be altered) |
| **Syntax** | void func(int x) | void func(int &x) |
| **Use case** | When you don’t want to change original data | When you need to modify original data |

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

Ans: #include <iostream>

using namespace std;

// Function to swap two integers using references

void swap(int &x, int &y) {

int temp = x;

x = y;

y = temp;

}

int main() {

int a = 5, b = 10;

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

swap(a, b); // Call by reference

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

return 0;

}

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

Ans. An **inline function** is a function where the **function code is inserted directly into the calling location** by the compiler, instead of making a traditional function call.

**Purpose:-** To reduce the **overhead of function calls**, especially for **small, frequently called functions**.

**Syntax:-** inline return\_type function\_name(parameters) {

// function body

}

**Advantages:**

* **Faster execution** (no function call overhead)
* Good for **small, frequently used** functions

**Disadvantages:**

* **Increased binary size** (if overused)
* **Not guaranteed**: The compiler may **ignore** inline if it decides inlining isn't beneficial

#include <iostream>

using namespace std;

inline int square(int x) {

return x \* x;

}

int main() {

cout << "Square of 5: " << square(5) << endl; // Output: 25

return 0;

}

Que.14: How do inline function improve performance?

Ans. Inline functions can improve performance by **eliminating the overhead of a function call**.

**Performance Benefits:**

| **Aspect** | **Normal Function Call** | **Inline Function** |
| --- | --- | --- |
| **Function call overhead** | Yes — needs stack setup, jump, return | No — code is directly inserted |
| **Execution speed** | Slightly slower | Faster, especially in small functions |
| **Context switching** | Requires moving control to another location | All code remains in one flow |

Que.15: Explain the syntax for declaring an inline function?

Ans. **Syntax:-** inline return\_type function\_name(parameters) {

// function body

}

Que.16: What are macros in C++ and how are they different from inline function?

Ans. **Macros** in C++ are **preprocessor directives** that define **shortcuts or symbolic names** for code, using #define.

**Syntax:-** #define NAME replacement\_text

#include <iostream>

#define PI 3.14159

int main() {

double r = 5;

double area = PI \* r \* r;

std::cout << "Area: " << area << std::endl;

return 0;

}

* Macros are **text substitutions** done before compilation.
* They're a **C-style feature**, and in modern C++ it's **better to use const, inline, or constexpr**.
* Use macros **only when necessary**, like for **header guards** or conditional compilation.

Que.17: Explain the advantages and disadvantages of using macros over inline function?

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

| **Advantage** | **Explanation** |
| --- | --- |
| **No type checking needed** | Macros are simple text substitutions — they work with any type of argument. |
| **Faster preprocessing** | The preprocessor just replaces the text, no need for compilation logic. |
| **Can define complex expressions** | Useful for conditional compilation (#ifdef, etc.) and header guards. |
| **Work even before compiler runs** | Macros are handled by the preprocessor, so they're flexible for low-level code tricks. |

**Disadvantages of Macros over Inline Functions:-**

| **Disadvantage** | **Explanation** |
| --- | --- |
| **No type safety** | The compiler doesn’t check data types in macros. This can lead to subtle bugs. |
| **Hard to debug** | Errors may point to the expanded code, not the macro line. |
| **No scope control** | Macros are globally visible; you can't restrict their scope like inline functions. |
| **Side effects** | Macros can cause unexpected behavior if arguments have expressions with side effects.  Example: #define SQUARE(x) x\*x → SQUARE(1+2) becomes 1+2\*1+2 = 5, not 9 |
| **Not optimized like functions** | Compilers optimize inline functions much better than raw macro expansions. |

Que.18: Provide an example to illustrate the difference between macros and inline functions?

Ans.

| **Feature** | **Macro (#define)** | **Inline Function (inline)** |
| --- | --- | --- |
| **Definition** | Preprocessor directive for **text substitution** | Function that the compiler **expands inline** |
| **Handled by** | **Preprocessor** (before compilation) | **Compiler** (during compilation) |
| **Type safety** | No — No type checking | Yes — Enforces type rules |
| **Debugging** | Difficult — error messages refer to expanded code | Easier — part of compiled code |
| **Scope control** | No — global only | Yes — respects C++ scoping rules |
| **Side effects** | Risky — expressions can be evaluated multiple times | Safe — arguments are evaluated once |
| **Performance** | Fast (no call overhead) | Fast (also avoids call overhead) |
| **Code readability** | Hard to read or trace | Easier to understand |
| **Can be overloaded?** | No | Yes — supports function overloading |
|  |  |  |

Que.19: What is function overloading in C++?

Ans. **Function Overloading** in C++ means defining **multiple functions** with the **same name** but **different parameter lists** (number or types of parameters).  
The compiler chooses the correct version based on the arguments used when the function is called.

* Same function name
* Different number or **types** of parameters
* Return type **alone** is **not enough** to overload

#include <iostream>

using namespace std;

void print(int x) {

cout << "Integer: " << x << endl;

}

void print(double x) {

cout << "Double: " << x << endl;

}

void print(string x) {

cout << "String: " << x << endl;

}

int main() {

print(10); // Calls print(int)

print(3.14); // Calls print(double)

print("Hello"); // Calls print(string)

return 0;

}

Que.20: How does the compiler differentiate between overloaded functions?

Ans.

| **Feature** | **Compiler** | **Overloaded Function** |
| --- | --- | --- |
| **Definition** | A software tool that translates source code (C++) into machine code | A function that has the same name but different parameters |
| **Purpose** | Converts and optimizes code for execution | Increases code flexibility and readability |
| **Responsibility** | Syntax checking, type checking, code generation, linking | Chooses the correct function version based on arguments |
| **Part of program?** | No — it's an external tool | Yes — part of your actual C++ program |
| **Example** | GCC, Clang, MSVC | void print(int);, void print(double); |
| **Used when?** | During the build/compile process | During function calls at **compile time** |

Que.21: Provide an example of overloaded function in C++?

Ans. #include <iostream>

using namespace std;

// Function to add two integers

int add(int a, int b) {

return a + b;

}

// Function to add two doubles

double add(double a, double b) {

return a + b;

}

// Function to add three integers

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

return a + b + c;

}

int main() {

cout << "add(3, 4) = " << add(3, 4) << endl;

cout << "add(2.5, 3.5) = " << add(2.5, 3.5) << endl;

cout << "add(1, 2, 3) = " << add(1, 2, 3) << endl;

* All functions are named add, but they have **different parameter lists**.
* The compiler determines which version to call based on the **number and type of arguments**.

Que.22: What are default arguments in C++?

Ans. **Default arguments** in C++ are values **assigned to function parameters** so that **you can call the function without providing all arguments**.

**Syntax:-** return\_type function\_name(type1 param1 = default1, type2 param2 = default2);

#include <iostream>

using namespace std;

void greet(string name = "User") {

cout << "Hello, " << name << "!" << endl;

}

int main() {

greet(); // Uses default: "User"

greet("Alice"); // Uses provided value

return 0;

}

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

Ans. In C++, you can specify default arguments by assigning a value to the parameter in the function declaration (or definition). The default values are provided from right to left, so once you provide a default for a parameter, all subsequent parameters must also have default values.

**Syntax:-** return\_type function\_name(type1 param1, type2 param2 = default\_value);

**Example:-** #include <iostream>

using namespace std;

void greet(string name, string greeting = "Hello") {

cout << greeting << ", " << name << "!" << endl;

}

int main() {

greet("Alice"); // Uses default value "Hello"

greet("Bob", "Hi"); // Uses "Hi" instead of the default "Hello"

return 0;

}

Que.24: What are the rules for using default arguments in function?

Ans: **1.Default Arguments Must Be Provided in Function Declaration or Definition:**

* Default arguments should be specified in the function declaration (prototype) or in the function definition.
* It is not necessary to specify default values in both places. If you provide default arguments in the declaration, you don't need to repeat them in the definition.

**2.Defaults Must Be Given from Right to Left**

* If you use default arguments, they must be provided starting from the rightmost parameter.
* This means once you specify a default argument for a parameter, all parameters to the right of it must also have default arguments.

**3.Cannot Provide Default Arguments for Already Defined Functions**

* If you declare a function with default arguments in one place (e.g., header file), you cannot provide default arguments in the function definition (e.g., source file).
* You should either specify default arguments in both the declaration and definition **or** just in the declaration.

**4.Default Arguments Are Not Considered in Function Overloading**

* You can overload a function that has default arguments, but the default argument will not be considered when the function is called with matching parameters.

**5.Use of Default Arguments in Function Overloading**

* When you overload functions that have default arguments, the default values only apply when the call matches the function that uses those defaults.

Que.25: Provide an example of a function with default arguments?

Ans. #include <iostream>

using namespace std;

// Function with default arguments

void displayInfo(string name, int age = 25, string city = "New York") {

cout << "Name: " << name << endl;

cout << "Age: " << age << endl;

cout << "City: " << city << endl;

}

int main() {

// Calling the function with all arguments

displayInfo("Alice", 30, "Los Angeles");

// Calling the function with the default values for age and city

displayInfo("Bob");

// Calling the function with the default value for city

displayInfo("Charlie", 35);

return 0;

}