Assignment 2 (Theory)

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

The `main` function is the entry point of a C++ program, where execution begins. It is mandatory in every C++ program, as the operating system calls it to start the program. The `main` function contains the primary logic or orchestrates calls to other functions. It typically returns an integer to indicate the program’s exit status (0 for success, non-zero for errors). Without `main`, the program cannot execute, as the compiler expects this function to initiate the workflow. It can also accept command-line arguments for additional functionality.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

int main() {

cout << "Program starts here\n"; // Main logic

return 0; // Indicates successful execution

}

```

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

The `main` function’s return type is typically `int`, signifying the program’s exit status to the operating system. Returning `0` indicates successful execution, while non-zero values signal errors or specific conditions, allowing scripts or systems to handle outcomes (e.g., in batch processing). The `void` return type is non-standard and not portable, though some compilers allow it. The return value is crucial for error handling in larger systems, enabling communication between the program and its environment. If no return is specified, C++ implicitly returns `0` in `main`.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

int main() {

cout << "Running program\n";

if (true) {

return 0; // Success

}

return 1; // Error (unreachable in this case)

}

```

---

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

The C++ standard defines two valid signatures for the `main` function. The first is `int main()`, which takes no arguments and returns an `int`. This is used for simple programs without command-line input. The second is `int main(int argc, char\* argv[])`, which accepts command-line arguments: `argc` (argument count) and `argv` (array of argument strings). Both return an `int` to indicate exit status. These signatures ensure portability and compliance with the C++ standard. Non-standard signatures (e.g., `void main()`) are not guaranteed to work across compilers.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

int main(int argc, char\* argv[]) { // Second signature

cout << "Arguments: " << argc << "\n";

return 0;

}

// Alternative:

// int main() { cout << "No args\n"; return 0; }

```

---

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

Function prototyping declares a function’s signature (return type, name, and parameters) before its use, without providing the implementation. It informs the compiler about the function’s existence, enabling calls before the function is defined. In C++, prototyping is necessary because the compiler processes code sequentially and needs to know a function’s signature to validate calls. Without a prototype, a function call before its definition causes a compilation error. Prototypes are typically placed in header files or at the top of a source file for organization.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void sayHello(); // Function prototype

int main() {

sayHello(); // Valid call due to prototype

return 0;

}

void sayHello() { // Function definition

cout << "Hello!\n";

}

```

---

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

A function prototype declares the function’s return type, name, and parameter list without the body. For a function returning an `int` and taking two `int` parameters, the prototype is written as: `int functionName(int, int);`. The parameter names are optional in prototypes, but including them improves readability. The prototype must match the function’s definition. It is typically placed before `main` or in a header file to ensure the compiler recognizes the function.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

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

int main() {

cout << "Sum: " << add(3, 4) << "\n";

return 0;

}

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

return a + b;

}

```

---

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

If a function is called before it is prototyped or defined in C++, the compiler generates an error, typically “undeclared identifier” or “function not declared.” This occurs because C++ processes code sequentially and requires prior knowledge of a function’s signature to validate the call. Without a prototype or definition, the compiler cannot verify the function’s return type or parameters. To avoid this, a prototype must be declared before the call, or the function must be defined earlier in the code.

\*\*Example (Error Case)\*\*:

```cpp

#include <iostream>

using namespace std;

int main() {

sayHello(); // Error: sayHello not declared

return 0;

}

void sayHello() {

cout << "Hello!\n";

}

// Fix: Add prototype

// void sayHello();

// int main() { sayHello(); return 0; }

```

---

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

A \*\*declaration\*\* introduces a function’s signature (return type, name, parameters) to the compiler without its implementation, e.g., a prototype like `void func(int);`. It allows the function to be called before its definition. A \*\*definition\*\* provides the function’s implementation, including the body, e.g., `void func(int x) { cout << x; }`. Declarations can appear multiple times (e.g., in header files), but definitions can only appear once to avoid redefinition errors. Declarations are necessary for compilation, while definitions provide the executable code.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void print(int); // Declaration (prototype)

int main() {

print(5); // Valid due to declaration

return 0;

}

void print(int x) { // Definition

cout << "Value: " << x << "\n";

}

```

---

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

A function that takes no parameters and returns `void` is called by using its name followed by empty parentheses, e.g., `functionName();`. The `void` return type means it does not return a value, so it cannot be used in assignments. Such functions typically perform actions like printing or modifying global state. The call must occur after the function’s prototype or definition to avoid compilation errors. The function is invoked directly within a scope, such as `main`.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void sayHello(); // Prototype

int main() {

sayHello(); // Function call

return 0;

}

void sayHello() { // Definition

cout << "Hello, World!\n";

}

```

---

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

Scope defines the region in a C++ program where a variable or function is accessible. In the context of functions, \*\*function scope\*\* refers to variables declared within a function, which are only accessible inside that function and destroyed when it exits. \*\*Global scope\*\* includes functions or variables declared outside all functions, accessible program-wide. \*\*Block scope\*\* limits variables within curly braces `{}` inside a function. Function names themselves are typically in global scope unless defined within a namespace. Scope prevents naming conflicts and manages variable lifetime.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void func() {

int x = 10; // Function scope

cout << "In func: " << x << "\n";

}

int main() {

// cout << x; // Error: x not in scope

func(); // Function call

return 0;

}

```

---

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

Call by reference in C++ passes a function argument by reference, meaning the function receives a reference to the original variable, not a copy. Changes made to the parameter inside the function affect the original variable. This is achieved using the `&` operator in the parameter declaration (e.g., `void func(int& x)`). Call by reference is useful for modifying multiple variables or avoiding the overhead of copying large objects. It ensures direct access to the caller’s data.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void increment(int& x); // Reference parameter

int main() {

int num = 5;

increment(num);

cout << "Num: " << num << "\n"; // Outputs 6

return 0;

}

void increment(int& x) {

x++; // Modifies original

}

```

---

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

Call by value passes a copy of the argument to a function, so changes to the parameter do not affect the original variable. Call by reference passes a reference to the original variable, allowing the function to modify it directly. Call by value is default in C++ (e.g., `void func(int x)`), while call by reference uses `&` (e.g., `void func(int& x)`). Call by value is safer for immutable data but less efficient for large objects. Call by reference is efficient but requires caution to avoid unintended changes.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void byValue(int x) { x++; }

void byReference(int& x) { x++; }

int main() {

int a = 5, b = 5;

byValue(a);

byReference(b);

cout << "By value: " << a << "\n"; // Still 5

cout << "By reference: " << b << "\n"; // Now 6

return 0;

}

```

---

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

A function to swap two integers using call by reference declares parameters with `&` to access the original variables. Inside the function, a temporary variable stores one value while the swap is performed. This modifies the caller’s variables directly. The function is called with the variables to swap, and no return value is needed since the references handle the changes.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

void swap(int& a, int& b) { // Reference parameters

int temp = a;

a = b;

b = temp;

}

int main() {

int x = 10, y = 20;

cout << "Before: x=" << x << ", y=" << y << "\n";

swap(x, y); // Swap values

cout << "After: x=" << x << ", y=" << y << "\n";

return 0;

}

```

---

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

An inline function in C++ is a function marked with the `inline` keyword, suggesting to the compiler that its code should be inserted directly at the call site instead of making a function call. This reduces the overhead of function calls (e.g., stack setup) for small, frequently used functions. Inline functions are defined in header files or before their use, as their code must be available to the compiler. They are a hint, not a guarantee, as the compiler may ignore the suggestion for large functions.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

inline int square(int x) { // Inline function

return x \* x;

}

int main() {

cout << "Square: " << square(5) << "\n"; // Code inserted here

return 0;

}

```

---

\*\*14. How do inline functions improve performance?\*\*

Inline functions improve performance by eliminating the overhead of function calls, such as pushing arguments onto the stack, jumping to the function, and returning. The compiler replaces the function call with Principledly inserts the function’s code at the call site, reducing execution time for small, frequently called functions. This is particularly effective for short functions, as it avoids context-switching costs. However, excessive inlining can increase code size, potentially slowing execution due to cache misses. The compiler optimizes inlining decisions.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

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

int main() {

cout << "Sum: " << add(3, 4) << "\n"; // Faster execution

return 0;

}

```

---

\*\*15. Explain the syntax for declaring an inline function.\*\*

An inline function is declared using the `inline` keyword before the function’s return type, followed by the function name, parameters, and body. The syntax is: `inline returnType functionName(parameters) { body; }`. The function must be defined in the same translation unit (e.g., header file or before use), as the compiler needs the code to insert it at call sites. Parameter names and types are specified as usual. The `inline` keyword is a suggestion to the compiler.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

inline void print(int x) { // Inline declaration

cout << "Value: " << x << "\n";

}

int main() {

print(10); // Code inserted here

return 0;

}

```

---

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

Macros in C++ are preprocessor directives defined with `#define`, performing text substitution before compilation (e.g., `#define SQUARE(x) ((x)\*(x))`). They are not type-safe, lack scope, and can cause unexpected behavior due to blind substitution. Inline functions, marked with `inline`, are actual functions with type checking, scope, and proper parameter passing, compiled into the program. Macros are processed by the preprocessor, while inline functions are handled by the compiler, offering better safety and debugging. Macros are less preferred in modern C++.

\*\*Example\*\*:

```cpp

#include <iostream>

using namespace std;

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

inline int square(int x) { return x \* x; } // Inline

int main() {

cout << SQUARE(3) << "\n"; // Macro: text replacement

cout << square(3) << "\n"; // Inline: type-safe

return 0;

}

```

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

Macros expand code at compile-time, avoiding function call overhead. They’re type-agnostic, working with any data type. Macros enable complex text substitutions for flexible code generation. They’re concise for simple operations. However, macros lack type safety, risking errors from blind text replacement. Debugging is tough as they don’t appear in symbol tables. They can bloat code with repeated expansions. Macros ignore scope and access rules, causing naming conflicts. Side effects in arguments (e.g., `x++`) may evaluate multiple times, leading to bugs. Inline functions are type-safe and debuggable. They respect scope and access control. Inline functions evaluate arguments once, avoiding side-effect issues. Macros suit simple, type-agnostic tasks. Inline functions are safer for modern C++.

---

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

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

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

`int main() {`

` int a = 5;`

` std::cout << SQUARE(a++) << '\n';`

` std::cout << square(a) << '\n';`

`}`

Macro expands to `((a++) \* (a++))`, incrementing `a` twice (e.g., outputs 30, `a=7`). Inline function evaluates `a` once, outputting 25 (`a=6`). Macros risk side effects with multiple evaluations. Inline functions are safer, evaluating arguments once. Macros lack type checking; inline functions enforce types. Macros don’t respect scope; inline functions do. This shows inline functions’ reliability.

---

\*\*19. What is function overloading in C++?\*\*

Function overloading lets multiple functions share a name but differ in parameter lists. Parameters vary by number, type, or order. It’s resolved at compile-time based on arguments. Overloading improves code readability and reusability. Users call one name for related tasks. For example, `print(int)` and `print(double)` handle different types. Return type alone can’t differentiate functions. It’s common in libraries like `std::cout`. Overloading requires distinct signatures. Ambiguity arises if arguments match multiple functions. It supports polymorphism, unlike virtual functions. Overloading provides type-specific behavior. It’s a key C++ feature for intuitive interfaces.

---

\*\*20. How does the compiler differentiate between overloaded functions?\*\*

The compiler resolves overloaded functions at compile-time via overload resolution. It examines the function’s parameter list (number, types, order). Candidate functions share the same name. The compiler seeks an exact match for argument types. Implicit conversions (e.g., `int` to `double`) are considered if needed. Template deduction applies for template functions. Exact matches rank higher than conversions. Conversion hierarchy (e.g., `int` to `long` over `int` to `double`) guides selection. Ambiguity occurs if multiple functions match equally. Default arguments can complicate resolution. Const-qualifiers on parameters matter. Return type is ignored. This ensures the best function is chosen.

---

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

`#include <iostream>`

`void print(int x) { std::cout << "Int: " << x << '\n'; }`

`void print(double x) { std::cout << "Double: " << x << '\n'; }`

`void print(const std::string& s) { std::cout << "String: " << s << '\n'; }`

`int main() {`

` print(42);`

` print(3.14);`

` print("Hello");`

`}`

\*\*Output\*\*: `Int: 42`, `Double: 3.14`, `String: Hello`. Each `print` handles a unique type. Compiler picks based on argument type. Overloading unifies the interface. It’s resolved at compile-time.

---

\*\*22. What are default arguments in C++?\*\*

Default arguments assign predefined values to function parameters. They’re used if arguments are omitted in calls. They simplify calls for optional parameters. For example, `void func(int x, int y = 1)` defaults `y` to 1. Specified in function declarations. They’re resolved at compile-time. Trailing parameters must have defaults. They enhance flexibility for common values. Used in libraries for simpler interfaces. Overuse can reduce code clarity. Callers may miss default behavior. Defaults differ from overloading. They modify one function’s behavior. They’re a C++ convenience feature.

---

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

Default arguments are set in the function declaration. Use `type param = value` syntax. Example: `void draw(int w, int h, int c = 0);`. Definition omits defaults: `void draw(int w, int h, int c) { ... }`. Defaults go in headers for visibility. Trailing parameters get defaults. Multiple defaults are allowed (e.g., `int x = 1, int y = 2`). Invalid: `int x = 1, int y`. Declaration informs compiler of defaults. Callers can omit trailing arguments. Defaults ensure unambiguous calls. Example call: `draw(10, 20)` uses `c=0`. This simplifies function interfaces.

---

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

1. Defaults apply to trailing parameters only.

2. Specify in declaration, not definition.

3. Contiguous defaults from right (e.g., `int x = 1, int y = 2`).

4. No redefinition of defaults in same scope.

5. Calls must resolve defaults clearly.

6. Avoid ambiguity with overloaded functions.

7. Resolved at compile-time.

8. Trailing defaults ensure valid calls.

9. Defaults simplify interfaces.

10. Overuse can obscure behavior.

11. Must follow parameter order.

12. No gaps in default sequence.

13. Const parameters can have defaults.

14. Templates support defaults.

15. Enhance readability if used wisely.

---

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

`#include <iostream>`

`void printMsg(const std::string& msg, int n = 1, char c = '\n');`

`void printMsg(const std::string& msg, int n, char c) {`

` for (int i = 0; i < n; ++i) std::cout << msg << c;`

`}`

`int main() {`

` printMsg("Hi");`

` printMsg("Hey", 2);`

` printMsg("Wow", 3, '!');`

`}`

\*\*Output\*\*: `Hi\n`, `Hey\nHey\n`, `Wow!Wow!Wow!\n`. Defaults: `n=1`, `c='\n'`. `printMsg("Hi")` prints once. `printMsg("Hey", 2)` prints twice. Trailing defaults simplify calls.