بسم الله الرحمن الرحيم و الصلاة والسلام على أشرف المرسلين

What is C++?

C++ is a general-purpose programming language created by Bjarne Stroustrup in the early 1980s as an extension of the C language. It is known for its support of object-oriented, procedural, and generic programming, making it versatile for system and application development. C++ is widely used in software engineering for its high performance, extensive libraries, and low-level memory manipulation capabilities.

Basic Syntax and Structure

C++ programs typically start with preprocessor directives, followed by the ‘main‘ function, which serves as the entry point.
Below is a simple ”Hello, World!” example:

```
#include <iostream>
int main() {
    std::cout << "Hello,␣World!" << std::endl;
    return 0;
}
```

Data Types

C++ supports several basic data types:

- `int`: Integer (e.g., `int age = 30;`)
- `double`: Floating point (e.g., `double pi = 3.14;`)
- `char`: Character (e.g., `char initial = 'A';`)
- `bool`: Boolean (e.g., `bool isTrue = true;`)
- `std::string`: String (e.g., `std::string name = "Ahmad";`)

Variables

Variables are used to store data in C++. You must declare a variable with a specific data type before using it.

- Declaration: `int age;`
- Initialization: `age = 30;`
- Declaration and Initialization: `int age = 30;`

Operators

C++ supports various operators for performing operations on variables:

- **Arithmetic Operators**: `+, -, *, /, %` (addition, subtraction, multiplication, division, modulus)
- **Relational Operators**: `==, !=, <, >, <=, >=` (comparison operations)
- **Logical Operators**: `, ||, !` (logical AND, OR, NOT)
- **Assignment Operators**: `=, +=, -=, *=, /=` (assign values and perform operations)
- **Increment/Decrement Operators**: `++, --` (increase or decrease by one)

Control Structures

Control structures in C++ dictate the flow of program execution. Key types include:

- **Conditional Statements**:
  - `if` - Executes a block if the condition is true.
  - `else` - Executes a block if the preceding `if` is false.
  - `else if` - Checks another condition if the previous `if` is false.
  - `switch` - Selects a block to execute based on the value of a variable.
- **Loops**:
  - `for` - Repeats a block a specific number of times.
  - `while` - Repeats a block while a condition is true.
  - `do while` - Repeats a block at least once and continues while a condition is true.

```
#include <iostream>

int main() {
    int number = 5;

    // Conditional statement
    if (number > 0) {
        std::cout << "Positive␣number"
            << std::endl;
    } else if (number < 0) {
        std::cout << "Negative␣number"
            << std::endl;
    } else {
        std::cout << "Zero"
            << std::endl;
    }

    // Loop
    for (int i = 0; i < 5; i++) {
        std::cout << "Iteration:␣"
            << i << std::endl;
    }

    return 0;
}
```

Functions

Functions are reusable blocks of code that perform a specific task. They improve code organization and modularity.

- **Function Declaration**: Specifies the function’s name, return type, and parameters.
- **Function Definition**: Contains the code that executes when the function is called.
- **Function Call**: Invokes the function to execute its code.

Example:

```
#include <iostream>

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

int main() {
    int result = add(5, 3); // Function call
    std::cout << "Sum:␣" << result << std::endl;
    return 0;
}

// Function definition
int add(int a, int b) {
    return a + b; // Returns the sum of a and b
}
```

Arrays and Strings

Arrays and strings are used to store collections of data in C++.

- **Arrays**: A collection of elements of the same type stored in contiguous memory locations.
  - Syntax: `type arrayName[size];`
- **Strings**: A sequence of characters represented as an object of the `std::string` class from the `<string>` library.
  - Syntax: `std::string str = "Hello";`

Example:

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

int main() {
    // Array of integers
    int numbers[5] = {1, 2, 3, 4, 5};

    // Accessing array elements
    std::cout << "First␣number:␣" << numbers[0] << std::

    // String
    std::string greeting = "Hello,␣World!";
    std::cout << greeting << std::endl;

    return 0;
}
```

Pointers and References

Pointers and references are used to manage memory and reference variables in C++.

- **Pointers**: Variables that store the memory address of another variable.
  - Syntax: `type* pointerName;`
- **References**: An alias for another variable, allowing direct access to that variable's value.
  - Syntax: `type referenceName = variable;`

Example:

```
#include <iostream>

int main() {
    int value = 10;

    // Pointer
    int* ptr = &value; // Stores the address of value
    std::cout << "Value␣via␣pointer:␣" << *ptr
        << std::endl; // Dereference pointer

    // Reference
    int& ref = value; // Reference to value
    std::cout << "Value␣via␣reference:␣" << ref
        << std::endl;

    return 0;
}
```

Classes and Objects

C++ is an object-oriented programming language that uses classes and objects to structure code.

- **Class**: A blueprint for creating objects. It defines attributes and methods.
  - Syntax: `class ClassName { …};`
- **Object**: An instance of a class that contains data and functions defined by the class.

Example:

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

class Car {
public:
    std::string model;
    int year;

    void displayInfo() {
        std::cout << "Model:␣" << model <<
            ",␣Year:␣" << year << std::endl;
    }
};

int main() {
    Car myCar; // Create an object of the Car class
    myCar.model = "Toyota";
    myCar.year = 2020;
    myCar.displayInfo(); // Call the displayInfo me

    return 0;
}
```

Exception Handling

Exception handling in C++ provides a way to respond to runtime errors using try, catch, and throw keywords.

- **try**: Block of code that may throw an exception.
- **catch**: Block of code that handles the exception.
- **throw**: Used to signal the occurrence of an exceptional condition.

Example:

```
#include <iostream>

int main() {
    try {
        int divisor = 0;
        if (divisor == 0) {
         throw std::runtime_error("Division␣by␣zero!");
         // Throw an exception
        }
        int result = 10 / divisor;
        std::cout << "Result:␣" << result << std::endl;
    } catch (const std::runtime_error& e) {
        std::cout << "Error:␣" << e.what() << std::endl;
        // Handle the exception
    }

    return 0;
}
```

Standard Template Library (STL)

The Standard Template Library (STL) is a powerful set of C++ template classes that provide general-purpose classes and functions for data structures and algorithms.

- **Components of STL**:
  - **Containers**: Classes that store collections of objects (e.g., `vector`, `list`, `map`).
  - **Algorithms**: Functions that operate on containers (e.g., `sort`, `find`).
  - **Iterators**: Objects that allow traversal through the elements of a container.

Example:

```
#include <iostream>
#include <vector>
#include <algorithm>

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

    // Sorting the vector
    std::sort(numbers.begin(), numbers.end());

    std::cout << "Sorted␣numbers:␣";
    for (int num : numbers) {
        std::cout << num << "␣";
    }
    std::cout << std::endl;

    return 0;
}
```

File I/O

File Input/Output (I/O) in C++ allows you to read from and write to files using streams.

- **Include the library**: Use `include <fstream>` for file operations.
- **File Streams**:
  - `std::ifstream`: For reading from files.
  - `std::ofstream`: For writing to files.
  - `std::fstream`: For both reading and writing.
- **Common Operations**:
  - Open a file: `file.open("filename.txt");`
  - Check if a file is open: `if (file.is_open())`
  - Close a file: `file.close();`

Example:

```
#include <iostream>
#include <fstream>

int main() {
    // Writing to a file
    std::ofstream outputFile("output.txt");
    if (outputFile.is_open()) {
        outputFile << "Hello,␣File␣I/O!" << std::endl;
        outputFile.close(); // Close the file
    } else {
        std::cout << "Unable␣to␣open␣file␣for␣writing."
            << std::endl;
    }

    // Reading from a file
    std::ifstream inputFile("output.txt");
    std::string line;
    if (inputFile.is_open()) {
        while (getline(inputFile, line)) {
            std::cout << line << std::endl;
        }
        inputFile.close(); // Close the file
    } else {
        std::cout << "Unable␣to␣open␣file␣for␣reading."
            << std::endl;
    }

    return 0;
}
```

Preprocessor Directives

Preprocessor directives are commands that give instructions to the preprocessor before actual compilation starts. They are used to include files, define macros, and control conditional compilation.

- **Common Directives**:
  - `#include`: Includes header files.
  - `#define`: Defines macros or constants.
  - `#ifdef` / `#ifndef`: Checks if a macro is defined or not.
  - `#endif`: Ends a conditional directive.

Example:

```
#include <iostream>
#define PI 3.14159

int main() {
    std::cout << "Value␣of␣PI:␣" << PI
        << std::endl;
    return 0;
}
```

Advanced Topics

Templates

Templates enable generic programming in C++, allowing functions and classes to operate with any data type.

- **Function Template**:

```
template <typename T>
T add(T a, T b) {
    return a + b;
}
```

- **Class Template**:

```
template <typename T>
class Box {
public:
    T value;
    Box(T val) : value(val) {}
};
```

Smart Pointers

Smart pointers manage dynamic memory and ensure proper resource management.

- `std::unique_ptr`: Owns a resource exclusively.
- `std::shared_ptr`: Allows multiple pointers to share ownership.
- `std::weak_ptr`: Provides a non-owning reference to a resource managed by `shared_ptr`.

Example:

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> uniquePtr =
        std::make_unique<int>(5);
    std::cout << "Unique␣Pointer␣Value:␣"
        << *uniquePtr << std::endl;

    std::shared_ptr<int> sharedPtr =
        std::make_shared<int>(10);
    std::cout << "Shared␣Pointer␣Value:␣"
        << *sharedPtr << std::endl;

    return 0;
}
```

Multithreading

C++11 introduced multithreading capabilities to enable concurrent execution.

- `std::thread`: Represents a single thread of execution.
- `std::mutex`: Provides mutual exclusion for shared resources.
- `std::condition_variable`: Allows threads to wait for certain conditions to occur.

Example:

```
#include <iostream>
#include <thread>
#include <mutex>

std::mutex mtx; // Mutex for critical section

void printMessage(const std::string& message) {
    std::lock_guard<std::mutex> lock(mtx);
        // Lock the mutex
    std::cout << message << std::endl;
}

int main() {
    std::thread t1(printMessage,
        "Hello␣from␣Thread␣1");
    std::thread t2(printMessage,
        "Hello␣from␣Thread␣2");

    t1.join();
    t2.join();

    return 0;
}
```

Namespaces

Namespaces help organize code and prevent name collisions. Example:

```
namespace MyNamespace {
    void display() {
        std::cout << "Hello␣from␣MyNamespace!"
            << std::endl;
    }
}

int main() {
    MyNamespace::display();
    // Accessing the function in the namespace
    return 0;
}
```

Move Semantics and Rvalue References

Move semantics allow resources to be moved rather than copied, improving performance.

- `std::move`: Converts an object to an rvalue reference.

Example:

```
#include <iostream>
#include <vector>

class Resource {
public:
    Resource() {
        std::cout << "Resource␣acquired."
            << std::endl;
    }
    Resource(const Resource&) {
        std::cout << "Resource␣copied."
            << std::endl;
    }
    Resource(Resource&&) noexcept {
        std::cout << "Resource␣moved."
            << std::endl;
    }
};

int main() {
    Resource res1;
    Resource res2 = std::move(res1);
        // Move resource

    return 0;
}
```

Best Practices and Common Pitfalls

Best Practices

- **Use Smart Pointers**: Prefer `std::unique_ptr` and `std::shared_ptr` to manage dynamic memory and avoid memory leaks.
- **Follow RAII Principle**: Resource Acquisition Is Initialization; use constructors and destructors to manage resource lifetimes.
- **Use Const Correctness**: Use `const` keyword to indicate that variables should not be modified, improving code safety.
- **Prefer Range-Based Loops**: Use range-based for loops for cleaner and safer iteration over containers.
- **Keep Functions Small and Focused**: Functions should do one thing well. This improves readability and maintainability.

Common Pitfalls

- **Dangling Pointers**: Ensure that pointers do not reference memory that has been deallocated.
- **Memory Leaks**: Always release dynamically allocated memory. Use smart pointers to prevent leaks.
- **Uninitialized Variables**: Always initialize variables before use to avoid undefined behavior.
- **Using Raw Pointers**: Avoid using raw pointers for resource management. Use smart pointers instead.
- **Ignoring Exception Safety**: Always consider exception safety in your code, ensuring proper resource cleanup.

Useful Tools and Libraries

Compilers

- **GCC (GNU Compiler Collection)**: A widely used open-source compiler for C and C++.
- **Clang**: A compiler based on LLVM that offers fast compilation and excellent diagnostics.
- **MSVC (Microsoft Visual C++)**: A powerful IDE and compiler for Windows development.

Build Systems

- **CMake**: A cross-platform build system generator that simplifies project configuration.
- **Make**: A widely used build automation tool that uses Makefiles to manage builds.
- **Ninja**: A small build system focused on speed, often used with CMake.

Libraries

- **Boost**: A collection of peer-reviewed, portable C++ source libraries, enhancing functionality and performance.
- **Qt**: A comprehensive framework for building cross-platform applications with a focus on GUI development.
- **Poco C++ Libraries**: A set of C++ libraries for building networked applications and services.
- **OpenCV**: A library for computer vision and image processing.
- **SFML (Simple and Fast Multimedia Library)**: A library for multimedia applications, including graphics, audio, and network functionalities.

Debugging and Profiling Tools

- **GDB (GNU Debugger)**: A powerful debugger for C/C++ applications, allowing step-by-step execution.
- **Valgrind**: A tool for memory debugging, memory leak detection, and profiling.
- **Visual Studio Debugger**: A comprehensive debugging tool integrated with Visual Studio.

Resources for Further Learning

Recommended Books

- **C++ Primer** by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo: A comprehensive introduction to C++ for beginners and experienced programmers.
- **Effective C++** by Scott Meyers: A guide to writing high-quality C++ code with practical advice and best practices.
- **The C++ Programming Language** by Bjarne Stroustrup: Written by the creator of C++, this book provides an in-depth understanding of the language.
- **Programming: Principles and Practice Using C++** by Bjarne Stroustrup: Aimed at beginners, it teaches programming concepts using C++.
- **Accelerated C++** by Andrew Koenig and Barbara E. Moo: An introduction to C++ that emphasizes a modern approach to programming.

Online Courses

- **C++ for C Programmers** (Coursera): A course that helps C programmers transition to C++.
- **Object-Oriented Data Structures in C++** (Coursera): Focuses on data structures and their implementation using C++.
- **Learn C++** (Codecademy): An interactive course for learning C++ fundamentals.
- **C++: From Beginner to Expert** (Udemy): A comprehensive course covering various C++ topics for all skill levels.
- **C++ Programming for Beginners** (edX): An introductory course aimed at absolute beginners.

Documentation Links

- **C++ Reference**: `https://en.cppreference.com/w/` - A comprehensive online reference for C++ language features and standard libraries.
- **C++ Standard Library Documentation**: `https://en.cppreference.com/w/cpp` - Detailed documentation on the C++ Standard Library.
- **ISO C++ Foundation**: `https://isocpp.org/` - The official site for the ISO C++ standards committee with news and resources.
- **Learn C++**: `https://www.learncpp.com/` - A free online tutorial for learning C++ from basics to advanced topics.
- **C++ Core Guidelines**: `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines` - A set of guidelines for writing high-quality C++ code.