# A Comprehensive Reference for the C and C++ Programming Languages

## Part I: The C Superset - Core Language Features

This part establishes the foundational syntax and semantics inherited from the C language, which form the bedrock of C++. Understanding these concepts is essential for low-level programming and for interacting with a vast ecosystem of C libraries.

### Section 1: Foundational Elements

This section covers the basic anatomy of a C/C++ program, from the entry point to the pre-compilation stage, which is a defining characteristic of the language family.

#### 1.1 Program Structure and the main Function

The main function serves as the mandatory entry point for any C or C++ program; execution begins here when the program is run by the operating system. Its declaration must have an int return type, which signals the program's exit status to the host environment. A return value of 0 or the macro EXIT\_SUCCESS (defined in <cstdlib>) indicates successful execution, while a non-zero value or EXIT\_FAILURE indicates an error.

The two primary signatures for the main function are:

1. int main(): A function that takes no arguments. In C, this is often written as int main(void) to be explicit.
2. int main(int argc, char \*argv): A function that accepts command-line arguments. argc is an integer representing the count of arguments, and argv is an array of C-style strings, where argv is the program name and subsequent elements are the arguments passed to it.

A typical program's structure involves including necessary header files, declaring functions (prototypes), and defining the main function, which contains the program's primary logic.

#### 1.2 The Preprocessor and its Directives

Before the source code is compiled, it is processed by a tool known as the preprocessor. This tool scans the code for lines beginning with a hash symbol (#), known as preprocessor directives, and modifies the source code based on these commands. These directives are not C++ statements and thus do not end with a semicolon.

* **#include for Source File Inclusion:** This directive replaces itself with the entire content of a specified file. It is the primary mechanism for using libraries and organizing code across multiple files.
  + #include <header>: Used for standard library headers (e.g., <iostream>, <vector>). The preprocessor searches for these files in a system-defined list of directories.
  + #include "file": Used for user-defined header files. The search typically begins in the same directory as the source file before checking the system directories.
* **#define for Macro Definitions:** This directive creates macros, which are associations between an identifier and a token string. The preprocessor performs a direct text substitution for every occurrence of the identifier.
  + **Object-like Macros:** Used to define symbolic constants. For example, #define PI 3.14159 replaces all instances of PI with 3.14159.
  + **Function-like Macros:** Accept parameters, mimicking function calls. For example, #define MAX(a,b) ((a) > (b)? (a) : (b)). It is critical to enclose both the entire replacement and each parameter in parentheses to avoid operator precedence errors during substitution. Special operators # (stringizing) and ## (concatenation) can be used within macro definitions.
* **Conditional Compilation:** Directives such as #if, #else, #elif, #endif, #ifdef, and #ifndef allow sections of code to be included or excluded from compilation based on certain conditions. This is essential for writing platform-specific code or creating different build configurations.
* **#undef:** This directive removes a previously defined macro, limiting its scope within a source file.

The preprocessor is more than a simple text-replacement utility; it is the fundamental mechanism that enables C++'s modular, multi-file compilation model. Large software projects are organized into multiple .cpp files, known as translation units, each compiled independently. For a function defined in one translation unit to be callable from another, the compiler requires a *declaration* of that function to verify the call's correctness. The standard practice is to place such declarations in header (.h or .hpp) files and the corresponding *definitions* in source (.cpp) files.

A source file then uses #include to gain access to the necessary declarations. However, in complex projects, a single source file might indirectly include the same header multiple times (e.g., A.h includes B.h, and main.cpp includes both A.h and B.h). Without a protective mechanism, the contents of B.h would be pasted into main.cpp twice, leading to re-declaration errors and violating the **One Definition Rule (ODR)**, which mandates that every function and variable has exactly one definition. The standard solution is the "include guard," a construct built from conditional compilation directives:

#ifndef UNIQUE\_IDENTIFIER\_H  
#define UNIQUE\_IDENTIFIER\_H  
  
// Contents of the header file (declarations, etc.)  
  
#endif // UNIQUE\_IDENTIFIER\_H

The first time the preprocessor encounters this header, UNIQUE\_IDENTIFIER\_H is not defined, so it defines it and includes the file's contents. On any subsequent encounter within the same translation unit, #ifndef evaluates to false, and the preprocessor skips the entire file. This makes conditional compilation an indispensable tool for maintaining modularity and enforcing the ODR in C++.

### Section 2: Data Types, Variables, and Constants

C++ is a statically and strongly-typed language, meaning the type of every variable must be explicitly declared, and this type is checked at compile-time to ensure operations are valid.

#### 2.1 Primitive (Fundamental) Data Types

These are the built-in types that form the basis for all other types in the language.

* **Integral Types:**
  + char: Used to store a single character. It is typically 1 byte in size. Its signedness (whether it can hold negative values) is implementation-defined.
  + short, int, long, long long: These types store integer values. The C++ standard guarantees minimum sizes: short is at least 16 bits, int is at least 16 bits, long is at least 32 bits, and long long is at least 64 bits. The standard also mandates the relationship sizeof(char) \le sizeof(short) \le sizeof(int) \le sizeof(long) \le sizeof(long long).
  + bool: Represents logical values true or false. It is typically 1 byte in size.
* **Floating-Point Types:**
  + float, double, long double: Used for numbers with fractional parts. They typically conform to the IEEE-754 standard for representation. The standard guarantees that a double is at least as large as a float, and a long double is at least as large as a double.
* **Character Types for Unicode (C++11 and later):**
  + wchar\_t: A wide-character type whose size is implementation-defined, used for character sets that do not fit within a single byte.
  + char8\_t (C++20), char16\_t, char32\_t: These types were introduced to provide explicit support for UTF-8, UTF-16, and UTF-32 character encodings, respectively.
* **Void Type:**
  + void: A special type that signifies the absence of a value. It is used as the return type for functions that do not return a value and for creating generic pointers (void\*) that can point to an object of any type.

#### 2.2 Type Modifiers

Modifiers are keywords used to alter the meaning of base types.

* signed and unsigned: These can be applied to integral types. signed types can represent both positive and negative numbers, while unsigned types can only represent non-negative numbers, effectively doubling their maximum positive range.
* short and long: These modify the size of int and double, affecting their range of values.
* const: The const qualifier declares a variable as a constant, meaning its value cannot be modified after initialization. Any attempt to do so will result in a compile-time error.

#### 2.3 Variable Declaration, Definition, and Initialization

In C++, a critical distinction exists between declaring and defining a variable.

* A **declaration** introduces an identifier and its type to the compiler, allowing it to be used in the code. It does not allocate memory. The extern keyword is used to declare a variable without defining it (e.g., extern int score;).
* A **definition** is a declaration that also reserves storage for the variable (e.g., int score;). A variable can be declared many times across different files, but it must be defined exactly once in the entire program to satisfy the One Definition Rule.

Variables can be initialized in several ways:

* **Copy-initialization:** int x = 10;.
* **Direct-initialization:** int x(10);.
* **List-initialization (C++11):** int x{10};. This form, also known as uniform initialization, is generally preferred because it prevents "narrowing conversions" (e.g., initializing an int with a double that would lose precision), which can be a source of bugs.

If no explicit initializer is provided, the variable undergoes default initialization. The outcome depends on its storage duration:

* Variables with **static storage duration** (global variables, variables declared with static) are **zero-initialized**.
* Variables with **automatic storage duration** (non-static local variables) are **uninitialized** and hold an indeterminate, or "garbage," value. Using such a variable before assigning it a value results in undefined behavior and is a common programming error.

#### 2.4 Composite (Derived and User-Defined) Types

Beyond the fundamental types, C++ allows for the creation of more complex types by composing simpler ones. These include arrays, pointers, references, structures (struct), and classes (class), which are foundational to building complex data structures and will be detailed in subsequent sections.

#### Table 2.1: Primitive Data Types in C/C++

The following table summarizes the fundamental data types, their typical sizes on modern 64-bit systems, and their guaranteed value ranges according to the C++ standard. Actual sizes may vary by platform and compiler.

| Type Keyword | Typical Size (bytes) | Typical Value Range |
| --- | --- | --- |
| bool | 1 | true or false |
| char | 1 | -128 to 127 or 0 to 255 (implementation-defined) |
| unsigned char | 1 | 0 to 255 |
| signed char | 1 | -128 to 127 |
| short | 2 | -32,768 to 32,767 |
| unsigned short | 2 | 0 to 65,535 |
| int | 4 | -2,147,483,648 to 2,147,483,647 |
| unsigned int | 4 | 0 to 4,294,967,295 |
| long | 4 or 8 | -2,147,483,648 to 2,147,483,647 (on 32-bit systems) |
| unsigned long | 4 or 8 | 0 to 4,294,967,295 (on 32-bit systems) |
| long long | 8 | -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 |
| unsigned long long | 8 | 0 to 18,446,744,073,709,551,615 |
| float | 4 | Approx. \pm3.4 \times 10^{38} with ~7 decimal digits of precision |
| double | 8 | Approx. \pm1.8 \times 10^{308} with ~15 decimal digits of precision |
| long double | 8, 12, or 16 | Higher precision than double (implementation-defined) |

### Section 3: Operators and Expressions

Operators are symbols that specify computations to be performed on operands (variables or values). The combination of operators and operands forms an expression, which the compiler evaluates to produce a value.

#### 3.1 Classification of Operators

C++ provides a rich set of operators that can be categorized as follows:

* **Arithmetic:** Perform mathematical operations: + (addition), - (subtraction), \* (multiplication), / (division), % (modulus).
* **Relational and Equality:** Compare two operands and return a bool result: == (equal to), != (not equal to), > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to).
* **Logical:** Perform boolean logic, typically on bool operands: ! (logical NOT), && (logical AND), || (logical OR).
* **Bitwise:** Operate on the individual bits of integral operands: & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (left shift), >> (right shift).
* **Assignment:** Assign a value to a variable: = (simple assignment), +=, -=, \*=, /=, %= (compound assignment).
* **Unary:** Operate on a single operand: ++ (increment), -- (decrement), & (address-of), \* (dereference), sizeof (size of type or object).
* **Ternary/Conditional:** A shorthand for an if-else statement: condition? expr\_if\_true : expr\_if\_false.
* **Member Access:** Access members of a struct or class: . (direct member access), -> (indirect/pointer member access).
* **Other:** :: (scope resolution), , (comma), () (type casting), `` (array subscripting).

#### 3.2 Operator Precedence and Associativity

In expressions with multiple operators, **precedence** and **associativity** determine the evaluation order.

* **Precedence** is the priority for grouping different operators. Operators with higher precedence are evaluated before operators with lower precedence. For instance, in a + b \* c, multiplication (\*) has higher precedence than addition (+), so the expression is evaluated as a + (b \* c).
* **Associativity** defines the evaluation order for operators with the same precedence. Most operators are left-to-right associative, meaning they are evaluated from left to right. For example, a - b + c is evaluated as (a - b) + c. A few operators, notably assignment operators and the conditional operator, are right-to-left associative. For example, a = b = c is evaluated as a = (b = c).

Parentheses () can always be used to explicitly control the order of evaluation, overriding the default precedence and associativity rules.

#### Table 3.1: Operator Precedence and Associativity

The following table lists C++ operators in descending order of precedence. Operators on the same row have equal precedence and are evaluated according to their associativity.

| Precedence | Operator | Description | Associativity |
| --- | --- | --- | --- |
| 1 | :: | Scope Resolution | Left-to-right |
| 2 | ++ -- () `` . -> typeid const\_cast dynamic\_cast reinterpret\_cast static\_cast | Postfix Increment/Decrement, Function Call, Subscript, Member Access | Left-to-right |
| 3 | ++ -- + - ! ~ \* & sizeof new delete (type) | Prefix Increment/Decrement, Unary Plus/Minus, Logical/Bitwise NOT, Dereference, Address-of, Size-of, Memory Allocation/Deallocation, C-style Cast | Right-to-left |
| 4 | .\* ->\* | Pointer-to-member | Left-to-right |
| 5 | \* / % | Multiplication, Division, Modulus | Left-to-right |
| 6 | + - | Addition, Subtraction | Left-to-right |
| 7 | << >> | Bitwise Left/Right Shift | Left-to-right |
| 8 | < <= > >= | Relational Operators | Left-to-right |
| 9 | == != | Equality Operators | Left-to-right |
| 10 | & | Bitwise AND | Left-to-right |
| 11 | ^ | Bitwise XOR | Left-to-right |
| 12 | ` | ` | Bitwise OR |
| 13 | && | Logical AND | Left-to-right |
| 14 | ` |  | ` |
| 15 | ?: = += -= \*= /= %= <<= >>= &= ^= ` | = throw` | Conditional, Assignment, Throw |
| 16 | , | Comma | Left-to-right |

A common point of confusion arises from the distinction between how operators are grouped and the order in which their operands are evaluated. Precedence and associativity rules strictly define how operands are grouped with operators. For example, in f() + g() \* h(), precedence dictates that the expression is parsed as f() + (g() \* h()). However, these rules do *not* specify whether the function f() is called before or after the sub-expression g() \* h() is evaluated. The order of evaluation of sub-expressions is, in most cases, deliberately left unspecified by the C++ standard to allow for compiler optimizations.

Relying on a specific evaluation order for expressions with side effects (i.e., expressions that modify state) can lead to non-portable code and undefined behavior. A classic example is i = i++;. The postfix increment i++ has a side effect of incrementing i, and the assignment = has a side effect of modifying i. Because the standard does not guarantee which side effect occurs first, the behavior of this expression is undefined.

The language does, however, define specific points in the evaluation of an expression, called **sequence points**, where all side effects of previous evaluations are guaranteed to be complete. The logical AND (&&), logical OR (||), conditional (?:), and comma (,) operators introduce sequence points. For a && b, a is fully evaluated first. If a is false, b is not evaluated at all (short-circuiting). This guaranteed left-to-right evaluation makes expressions like if (ptr!= nullptr && ptr->member) safe. Understanding this distinction is critical for writing correct and portable C++ code.

## Part II: The C++ Extensions - Modern and Object-Oriented Features

This part builds upon the C foundation, introducing the powerful abstractions and paradigms that define modern C++. The focus shifts from procedural and low-level control to object-oriented design, type safety, and resource management. The features introduced in this part are not merely additions but represent a coherent design philosophy aimed at providing safer, more expressive, and more abstract alternatives to lower-level C idioms. This shift encourages a style of programming that leverages the compiler to prevent common errors, such as null pointer dereferences, buffer overflows, and memory leaks.

### Section 9: C++ Enhancements to Core Features

This section covers C++ features that provide direct, superior alternatives to C idioms.

#### 9.1 References (&)

A reference is an alias for an existing object. It is declared using an ampersand (&) and must be initialized at the time of its declaration. int i = 10; int& ref = i; // ref is now an alias for i

Once initialized, a reference cannot be "reseated" to refer to a different object, nor can it be null. This makes references inherently safer than pointers for many use cases. Their primary application is in function parameters to enable **pass-by-reference**. This allows a function to modify the caller's arguments directly and avoids the performance overhead of copying large objects. Syntactically, a reference is used just like the original object, with dereferencing being implicit, which contrasts with the explicit dereferencing (\*) required for pointers.

#### 9.2 The std::string Class

The C++ standard library provides the std::string class in the <string> header as the modern, preferred method for handling text. It is a significant improvement over C-style null-terminated character arrays (char\*) because it automatically manages its own memory, growing and shrinking as needed, which virtually eliminates the risk of buffer overflows, a common security vulnerability with C-style strings.

Key operations include:

* **Concatenation:** Easily done with the + operator or the .append() method.
* **Size:** The .length() or .size() methods return the string's length in constant time, O(1), unlike C's strlen(), which must scan the string and is O(n).
* **Access:** Characters can be accessed via the subscript operator `` or the .at() method, which provides bounds checking and throws an exception on out-of-range access.
* **Manipulation:** A rich set of member functions is available for searching (.find()), extracting substrings (.substr()), and modification.
* **Comparison:** std::string objects can be compared lexicographically using standard relational operators (==, !=, <, >), which is more intuitive than C's strcmp() function.

#### Table 9.1: C-Style String (char\*) vs. C++ std::string

| Feature | C-Style String (char\*) | std::string |
| --- | --- | --- |
| **Memory Management** | Manual (prone to leaks and overflows) | Automatic (RAII) |
| **Size/Length** | Computed at runtime via strlen() (O(n)) | Stored with the object, retrieved via .size() (O(1)) |
| **Concatenation** | strcat() (unsafe, requires manual buffer checks) | + operator or .append() (safe, memory is managed) |
| **Comparison** | strcmp() function | Relational operators (==, <, etc.) |
| **Assignment** | strcpy() (unsafe) | = operator (safe) |
| **Safety** | Low; high risk of buffer overflows and null-terminator errors | High; bounds checking with .at(), automatic memory management |

#### 9.3 Dynamic Memory with new and delete

While C uses the library functions malloc() and free() for dynamic memory allocation, C++ introduces the new and delete operators. These are deeply integrated with the language's object model and should always be preferred in C++ code.

* **Syntax:** new allocates memory, and delete deallocates it. For arrays, the syntax is new type[size] and delete pointer, respectively. Mismatching delete with new or delete with new results in undefined behavior.
* **Type Safety:** new is a type-safe operator. new MyType() returns a MyType\*, eliminating the need for the explicit and potentially unsafe (MyType\*) cast required with malloc()'s void\* return type.
* **Constructor and Destructor Integration:** This is the most crucial difference. When an object is created with new, its constructor is automatically called after memory is allocated. When it is destroyed with delete, its destructor is called before the memory is deallocated. This ensures objects are properly initialized and cleaned up, a concept central to the RAII (Resource Acquisition Is Initialization) idiom. malloc() and free() are unaware of constructors and destructors.
* **Error Handling:** By default, if new fails to allocate memory, it throws a std::bad\_alloc exception. This allows for robust, centralized error handling using try-catch blocks. An alternative, non-throwing version can be used via the nothrow placement: int\* p = new (std::nothrow) int;, which returns nullptr on failure, similar to malloc().

#### Table 9.2: malloc/free vs. new/delete

| Feature | malloc() / free() (C-style) | new / delete (C++-style) |
| --- | --- | --- |
| **Type** | Library Functions | Language Operators |
| **Type Safety** | Not type-safe; returns void\* requiring a cast | Type-safe; returns a pointer of the correct type |
| **Constructor/Destructor** | Not called | Automatically called |
| **Error Handling** | Returns NULL on failure | Throws std::bad\_alloc exception by default |
| **Array Deallocation** | free(ptr) | delete ptr (distinct syntax for arrays) |
| **Overridability** | Cannot be customized per-class | Can be overridden on a per-class basis |

#### 9.4 Function Overloading

Function overloading allows multiple functions in the same scope to share the same name, as long as their parameter lists are different. The compiler distinguishes them based on the number, type, or order of their parameters. This combination of a function's name and its parameter list is known as the function's **signature**.

For example:

int add(int a, int b);  
double add(double a, double b);

The compiler determines which function to invoke at compile-time by matching the arguments in the function call. This is a form of compile-time polymorphism.

A key restriction is that functions **cannot** be overloaded based on the return type alone. The compiler would be unable to resolve a call where the return value is discarded, such as add(5, 10);, creating ambiguity.

#### 9.5 Namespaces

Namespaces provide a method for preventing name conflicts in large projects. They act as a container for a set of related identifiers (classes, functions, variables), preventing them from polluting the global scope. The entire C++ Standard Library is declared within the std namespace, which is why standard components are accessed with the std:: prefix (e.g., std::cout, std::vector).

The using namespace std; directive imports all names from the std namespace into the current scope. While this can be convenient in small programs, it is strongly discouraged in header files as it can re-introduce the very name collision problems that namespaces were designed to solve.

### Section 10: Object-Oriented Programming (OOP)

Object-Oriented Programming is a paradigm that uses objects—instances of classes—to design applications and computer programs. C++ is a multi-paradigm language with powerful support for OOP.

#### 10.1 The Four Pillars of OOP

* **Encapsulation:** The bundling of data (attributes) and the methods (functions) that operate on that data into a single unit, known as a class. This principle restricts direct access to some of an object's components, which is a key aspect of data hiding.
* **Abstraction:** The concept of hiding complex implementation details from the user and exposing only the essential features. The user interacts with an object through its public interface without needing to know how it works internally.
* **Inheritance:** The mechanism by which one class (the derived or child class) can acquire the properties and behaviors of another class (the base or parent class). This promotes code reuse and establishes an "is-a" relationship between classes (e.g., a Dog is an Animal).
* **Polymorphism:** From Greek for "many shapes," this is the ability of an object to take on many forms. In C++, it primarily manifests as compile-time polymorphism (function and operator overloading) and runtime polymorphism (virtual functions).

#### 10.2 Classes and Objects

* A **class** is a user-defined type or blueprint that defines a set of attributes (data members) and behaviors (member functions). It is a conceptual template.
* An **object** is a concrete instance of a class. When an object is created, memory is allocated for its data members.

A class is defined using the class keyword:

class MyClass {  
 // Access specifier  
 public:  
 // Data member  
 int myNum;  
 // Member function  
 void myMethod();  
};  
  
// Creating an object  
MyClass myObject;

* **Access Specifiers** control the visibility of class members:
  + public: Members are accessible from anywhere outside the class.
  + private: Members can only be accessed by other member functions of the same class. This is the mechanism for data hiding and encapsulation.
  + protected: Members are accessible within the class and by any derived (child) classes.

In C++, struct and class are nearly identical. The only difference is their default access level: members of a struct are public by default, whereas members of a class are private by default.

#### 10.3 Constructors and Destructors

* **Constructors** are special member functions that are automatically called when an object of a class is created. Their purpose is to initialize the object's data members. A constructor has the same name as the class and does not have a return type.
  + **Default Constructor:** A constructor that can be called with no arguments.
  + **Parameterized Constructor:** A constructor that accepts arguments for initialization.
  + **Copy Constructor:** Initializes an object from another object of the same type.
  + **Move Constructor (C++11):** Initializes an object by "stealing" resources from a temporary (rvalue) object, avoiding expensive copies.
* **Destructors** are special member functions automatically called when an object is destroyed (e.g., when it goes out of scope or is explicitly deleted). The destructor's name is the class name preceded by a tilde (~). Its purpose is to release any resources the object may have acquired during its lifetime (such as dynamic memory or file handles).

#### 10.4 Inheritance

Inheritance allows a new class (derived class) to be based on an existing class (base class). The derived class inherits the public and protected members of the base class, enabling code reuse.

* **Syntax:** class Derived : public Base { /\*... \*/ };.
* **Types of Inheritance:** C++ supports several inheritance models, including single, multiple (a class inheriting from more than one base class), multilevel (a chain of inheritance), hierarchical (multiple classes inheriting from a single base), and hybrid (a combination of the others).
* **Access Modes:** The inheritance mode (public, protected, or private) determines the access level of the inherited members within the derived class. Public inheritance is the most common and models a true "is-a" relationship.

#### 10.5 Polymorphism and Virtual Functions

Runtime polymorphism is a powerful feature of OOP that allows objects of different derived classes to be treated as objects of a common base class. This is achieved through the interplay of inheritance, pointers (or references), and virtual functions.

The mechanism works as follows:

1. **Inheritance** establishes an "is-a" relationship, allowing a base class pointer to point to a derived class object (e.g., Animal\* ptr = new Dog();).
2. When a member function is called through this base class pointer (ptr->makeSound();), the compiler's default behavior (static binding) is to call the base class's version of the function (Animal::makeSound()), regardless of the actual object type.
3. To achieve polymorphic behavior, the base class function must be declared with the **virtual** keyword. This instructs the compiler to use dynamic binding (or late binding). At runtime, the program will determine the actual type of the object being pointed to and call the appropriate overridden version of the function (e.g., Dog::makeSound()). This is typically implemented using a virtual function table, or vtable.

* **override Specifier (C++11):** When redefining a virtual function in a derived class, it is best practice to use the override specifier. This allows the compiler to verify that the function is indeed overriding a base class virtual function, catching potential errors like typos or parameter mismatches at compile time.
* **Pure Virtual Functions and Abstract Classes:** A pure virtual function is declared by appending = 0; to its declaration (e.g., virtual void makeSound() = 0;). A class that contains at least one pure virtual function is called an **abstract class** and cannot be instantiated. Abstract classes serve as interfaces, defining a contract that concrete derived classes must fulfill by providing implementations for all inherited pure virtual functions.
* **Virtual Destructors:** If a class is intended to be used as a polymorphic base class, its destructor **must** be declared virtual. If a derived object is deleted through a base class pointer and the destructor is not virtual, only the base class destructor will be called, leading to resource leaks. A virtual destructor ensures that the chain of destructors (from most-derived to base) is called correctly.

### Section 11: The Standard Template Library (STL)

The Standard Template Library (STL) is a powerful set of C++ template classes that provides generic data structures and algorithms. It is a cornerstone of modern C++ programming, promoting code reuse and efficiency.

#### 11.1 Overview: Containers, Iterators, Algorithms

The STL is composed of three main components:

* **Containers:** Data structures that store collections of objects. They are implemented as template classes, allowing them to store objects of any type.
* **Algorithms:** Template functions that perform operations on data, such as sorting, searching, and transforming. They are designed to work with containers.
* **Iterators:** Pointer-like objects that provide a uniform way to access elements within a container. They act as the "glue" that connects algorithms with containers, allowing algorithms to be generic and operate on any container type that provides the required iterator interface.

#### 11.2 Sequence Containers

These containers maintain the order of elements as they are inserted.

* **std::vector:** A dynamic, resizable array that stores elements in a contiguous block of memory. It offers fast random access (O(1)) and efficient insertions/deletions at the end (amortized O(1)). However, insertions or deletions in the middle are slow (O(n)) because subsequent elements must be shifted. It is the most commonly used sequence container.
* **std::list:** A doubly-linked list. It excels at fast insertions and deletions anywhere in the list (O(1)), but it does not support fast random access.
* **std::deque:** A "double-ended queue" that allows for efficient insertions and deletions at both the beginning and the end of the sequence.

#### 11.3 Associative Containers

These containers store elements in a sorted order, which allows for highly efficient searching (logarithmic time, O(log n)). They are typically implemented as self-balancing binary search trees.

* **std::map:** Stores elements as key-value pairs, with unique keys sorted in ascending order. It is ideal for dictionary-like data structures where fast lookup by key is required.
* **std::set:** Stores a collection of unique elements in sorted order. It is optimized for fast membership testing (i.e., checking if an element exists in the set).

#### 11.4 Unordered Associative Containers (C++11)

These containers (std::unordered\_map, std::unordered\_set) provide the same interfaces as their ordered counterparts but are implemented using hash tables. This results in faster average-case performance for search, insertion, and deletion (amortized O(1)), but the elements are not stored in any specific order.

#### 11.5 Container Adapters

These are not full container types but rather provide a specific, restricted interface to an underlying sequence container.

* **std::stack:** Provides a Last-In, First-Out (LIFO) data structure. The main operations are push(), pop(), and top().
* **std::queue:** Provides a First-In, First-Out (FIFO) data structure. The main operations are push(), pop(), front(), and back().

#### Table 11.1: Common std::vector Operations

| Function | Description | Time Complexity |
| --- | --- | --- |
| push\_back(value) | Adds an element to the end. | Amortized O(1) |
| pop\_back() | Removes the last element. | O(1) |
| insert(pos, value) | Inserts an element at iterator position pos. | O(n) |
| erase(pos) | Removes the element at iterator position pos. | O(n) |
| at(i) / [i] | Accesses the element at index i. | O(1) |
| size() | Returns the number of elements. | O(1) |
| empty() | Checks if the vector is empty. | O(1) |
| clear() | Removes all elements. | O(n) |
| begin() / end() | Returns iterators to the beginning/end. | O(1) |

#### Table 11.2: Common std::map Operations

| Function | Description | Time Complexity |
| --- | --- | --- |
| insert({key, val}) | Inserts a key-value pair. | O(log n) |
| erase(key) | Removes the element with the specified key. | O(log n) |
| find(key) | Returns an iterator to the element with the key, or .end() if not found. | O(log n) |
| [key] / at(key) | Accesses the value associated with the key. `` inserts if not found. | O(log n) |
| size() | Returns the number of elements. | O(1) |
| empty() | Checks if the map is empty. | O(1) |

### Section 12: Input/Output Streams in C++ (<iostream>, <fstream>)

C++ provides a type-safe, extensible I/O system through its stream libraries, which offer a unified interface for handling input and output to various devices.

#### 12.1 Standard Streams (<iostream>)

The <iostream> header defines standard stream objects for interacting with the console.

* std::cout: The standard output stream, used for printing to the console. It is used with the stream insertion operator (<<). It automatically handles different data types without requiring format specifiers.
* std::cin: The standard input stream, used for reading from the keyboard. It is used with the stream extraction operator (>>). By default, it reads input until it encounters whitespace.
* std::cerr and std::clog: Standard error streams for outputting error messages. cerr is unbuffered, while clog is buffered.

To read an entire line of input, including spaces, the std::getline() function should be used: std::string line; std::getline(std::cin, line);

#### 12.2 File Streams (<fstream>)

The <fstream> header provides classes for file-based I/O.

* **Classes:**
  + std::ifstream: Input file stream, for reading from files.
  + std::ofstream: Output file stream, for writing to files.
  + std::fstream: A file stream for both reading and writing.
* **File Operations Workflow:**
  1. **Create a stream object** and **open the file**, either in the constructor or with the .open() method. std::ofstream myFile("example.txt");
  2. **Check if the file was opened successfully** using the .is\_open() method. if (myFile.is\_open()) { /\*... \*/ }.
  3. **Perform read/write operations** using the same >> and << operators as with cin and cout. myFile << "Writing to the file.\n";
  4. **Close the file** with the .close() method. Files are also closed automatically when the stream object goes out of scope, thanks to its destructor. This is a key example of the RAII principle.

This stream-based approach provides a powerful abstraction. The << and >> operators offer a single, unified syntax for I/O, whether the target is the console, a file, or even an in-memory string (using <sstream>). This polymorphic behavior allows for the creation of generic code that can be easily redirected to different I/O targets simply by changing the type of stream object it operates on. This stands in contrast to C's distinct families of functions for console, file, and string I/O (e.g., printf, fprintf, sprintf).

#### Table 12.1: fstream Open Modes

The .open() method can take a second argument to specify the file mode. These flags, defined in the ios class, can be combined using the bitwise OR (|) operator.

| Mode Flag (ios::) | Description |
| --- | --- |
| in | Open for reading (default for ifstream). |
| out | Open for writing (default for ofstream). |
| app | Append mode: all output operations are performed at the end of the file. |
| ate | "At end": opens the file and moves the control to the end of the file. |
| trunc | Truncate: if the file already exists, its contents are discarded before opening. |
| binary | Open in binary mode (as opposed to text mode). |

#### Works cited

1. C Reference Card (ANSI), https://www.math.brown.edu/~jhs/ReferenceCards/CRefCard.v2.2.pdf 2. Functions in the C programming Language, https://users.cs.utah.edu/~germain/PPS/Topics/C\_Language/c\_functions.html 3. C Cheat Sheet - GeeksforGeeks, https://www.geeksforgeeks.org/c/c-cheatsheet/ 4. Functions - CPlusPlus.com, https://cplusplus.com/doc/tutorial/functions/ 5. FUNCTIONS IN C. A function is a set of instructions… | by Dev Frank | Medium, https://medium.com/@Dev\_Frank/functions-in-c-837a64673b40 6. Preprocessor directives - CPlusPlus.com, https://cplusplus.com/doc/tutorial/preprocessor/ 7. C++ Preprocessor And Preprocessor Directives - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/cpp-preprocessors-and-directives/ 8. The #include directive - IBM, https://www.ibm.com/docs/en/zos/3.1.0?topic=directives-include-directive 9. #define directive (C/C++) | Microsoft Learn, https://learn.microsoft.com/en-us/cpp/preprocessor/hash-define-directive-c-cpp?view=msvc-170 10. C Language CheatSheet | Blog - CodeWithHarry, https://www.codewithharry.com/blogpost/c-cheatsheet 11. Function declarations and definitions - IBM, https://www.ibm.com/docs/en/xl-c-and-cpp-aix/13.1.3?topic=functions-function-declarations-definitions 12. What is the difference between a definition and a declaration? - Stack Overflow, https://stackoverflow.com/questions/1410563/what-is-the-difference-between-a-definition-and-a-declaration 13. Data Types in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/cpp-data-types/ 14. Built-in types (C++) - Microsoft Learn, https://learn.microsoft.com/en-us/cpp/cpp/fundamental-types-cpp?view=msvc-170 15. Fundamental types - cppreference.com, https://en.cppreference.com/w/cpp/language/types.html 16. Variable declaration, definition and initialization difference? : r/C\_Programming - Reddit, https://www.reddit.com/r/C\_Programming/comments/eu8a7p/variable\_declaration\_definition\_and/ 17. Declaring and Initializing - What are th - C++ Forum, https://cplusplus.com/forum/beginner/80129/ 18. Variables In C++ | Declare, Initialize, Rules & Types (+Examples) - Unstop, https://unstop.com/blog/variables-in-cpp 19. Different Ways to Initialize a Variable in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/computer-science-fundamentals/different-ways-to-initialize-a-variable-in-cpp/ 20. Initializers - Microsoft Learn, https://learn.microsoft.com/en-us/cpp/cpp/initializers?view=msvc-170 21. Initialization - cppreference.com - C++ Reference, https://en.cppreference.com/w/cpp/language/initialization.html 22. C++ Data Types - Tutorials Point, https://www.tutorialspoint.com/cplusplus/cpp\_data\_types.htm 23. Composite data type - Wikipedia, https://en.wikipedia.org/wiki/Composite\_data\_type 24. Operator Precedence and Associativity in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/operator-precedence-and-associativity-in-cpp/ 25. C++ built-in operators, precedence, and associativity | Microsoft Learn, https://learn.microsoft.com/en-us/cpp/cpp/cpp-built-in-operators-precedence-and-associativity?view=msvc-170 26. Operator precedence and associativity - IBM, https://www.ibm.com/docs/en/i/7.5.0?topic=operators-operator-precedence-associativity 27. Operator precedence and associativity - cppreference.com - C++ Reference, https://en.cppreference.com/book/operator\_precedence 28. www.ibm.com, https://www.ibm.com/docs/en/i/7.5.0?topic=operators-operator-precedence-associativity#:~:text=Two%20operator%20characteristics%20determine%20how,that%20have%20the%20same%20precedence. 29. Precedence and order of evaluation | Microsoft Learn, https://learn.microsoft.com/en-us/cpp/c-language/precedence-and-order-of-evaluation?view=msvc-170 30. What is definition of reference type? - c++ - Stack Overflow, https://stackoverflow.com/questions/24827592/what-is-definition-of-reference-type 31. Tutorial: Passing Values to Functions in C++ - CodeHS, https://codehs.com/tutorial/david/passing-values-to-functions-in-c 32. C and C++ Strings — Programming and Data Structures 0.3 documentation, https://eecs280staff.github.io/notes/C\_and\_C%2B%2B\_Strings.html 33. Strings in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/strings-in-cpp/ 34. C++ String – std::string Example in C++ - freeCodeCamp, https://www.freecodecamp.org/news/c-string-std-string-example-in-cpp/ 35. Understanding strings functions on C versus C++ : r/C\_Programming - Reddit, https://www.reddit.com/r/C\_Programming/comments/1imgt6z/understanding\_strings\_functions\_on\_c\_versus\_c/ 36. Tutorial: String Basics in C++ - CodeHS, https://codehs.com/tutorial/12677 37. Dynamic memory - CPlusPlus.com, https://cplusplus.com/doc/tutorial/dynamic/ 38. C++ dynamic memory explained for beginners - YouTube, https://www.youtube.com/watch?v=fc19HhHKtrA 39. In what cases do I use malloc and/or new? - c++ - Stack Overflow, https://stackoverflow.com/questions/184537/in-what-cases-do-i-use-malloc-and-or-new 40. Memory Management, C++ FAQ - Standard C++, https://isocpp.org/wiki/faq/freestore-mgmt 41. new vs malloc and free vs delete explained with simple example - SimpleTechTalks, https://simpletechtalks.com/new-vs-malloc-and-free-vs-delete-explained-with-simple-example/ 42. new vs malloc() and free() vs delete in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/new-vs-malloc-and-free-vs-delete-in-c/ 43. Function Overloading | Microsoft Learn, https://learn.microsoft.com/en-us/cpp/cpp/function-overloading?view=msvc-170 44. Function Overloading in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/function-overloading-c/ 45. Function Overloading in C++ - Tutorials Point, https://www.tutorialspoint.com/cplusplus/cpp\_function\_overloading.htm 46. C++ overloaded functions explained - YouTube, https://www.youtube.com/watch?v=LZd5LhfnYsk 47. Overloading by return type - c++ - Stack Overflow, https://stackoverflow.com/questions/9568852/overloading-by-return-type 48. Function overloading - Wikipedia, https://en.wikipedia.org/wiki/Function\_overloading 49. C++ Programming Language - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/c-plus-plus/ 50. 1.5 — Introduction to iostream: cout, cin, and endl - Learn C++, https://www.learncpp.com/cpp-tutorial/introduction-to-iostream-cout-cin-and-endl/ 51. C++ Basic Input/Output - Programiz, https://www.programiz.com/cpp-programming/input-output 52. Basic Input / Output in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/basic-input-output-c/ 53. Object Oriented Programming in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/object-oriented-programming-in-cpp/ 54. C++ Encapsulation (With Examples) - Programiz, https://www.programiz.com/cpp-programming/encapsulation 55. C++ Object Oriented Programming - Programiz, https://www.programiz.com/cpp-programming/oop 56. C++ Inheritance - Programiz, https://www.programiz.com/cpp-programming/inheritance 57. Polymorphism in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/cpp-polymorphism/ 58. C++ Classes and Objects - Medium, https://medium.com/@pawan2505/c-classes-and-objects-4c60de81aa2d 59. C++ Classes and Objects - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/c-classes-and-objects/ 60. C++ Classes and Objects (With Examples) - Programiz, https://www.programiz.com/cpp-programming/object-class 61. Classes (I) - CPlusPlus.com, https://cplusplus.com/doc/tutorial/classes/ 62. Encapsulation in C++ by Logicmojo, https://logicmojo.com/encapsulation-in-cpp 63. Encapsulation in C++ and C - by Kateryna Bondarenko - Medium, https://medium.com/@kateolenya/encapsulation-in-c-and-c-20dca9097e17 64. struct (C++) | Microsoft Learn, https://learn.microsoft.com/en-us/cpp/cpp/struct-cpp?view=msvc-170 65. Understanding Structs in C++. C++ | Tips | by Madhawa Polkotuwa - Medium, https://madhawapolkotuwa.medium.com/understanding-structs-in-c-3b383faaceb3 66. Overview of constructors and destructors (C++ only) - IBM, https://www.ibm.com/docs/en/xl-c-and-cpp-aix/16.1.0?topic=only-overview-constructors-destructors-c 67. Understanding Constructors and Destructors in C++ | by Tomáš Svojanovský - Medium, https://medium.com/@tomsvoj/understanding-constructors-and-destructors-in-c-83235d5800e2 68. Constructors (C++) | Microsoft Learn, https://learn.microsoft.com/en-us/cpp/cpp/constructors-cpp?view=msvc-170 69. Inheritance in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/inheritance-in-c/ 70. 4.3: Implementing inheritance in C++ - Engineering LibreTexts, https://eng.libretexts.org/Courses/Delta\_College/C\_-\_Data\_Structures/04%3A\_Inheritence/4.03%3A\_Implementing\_inheritance\_in\_C 71. Learn Intermediate C++: Inheritance in C++ Cheatsheet | Codecademy, https://www.codecademy.com/learn/learn-intermediate-c-plus-plus/modules/inheritance-in-c-plus-plus/cheatsheet 72. Inheritance in C++ | for Interview | with Code and Example | by Harsh Chandwani - Medium, https://medium.com/@harshchandwani50028/inheritance-in-c-for-interview-with-code-and-example-7e36ea48499d 73. Inheritance (C++ only) - IBM, https://www.ibm.com/docs/en/zos/2.4.0?topic=reference-inheritance-c-only 74. C++ Virtual Functions and Function Overriding (With Examples) - Programiz, https://www.programiz.com/cpp-programming/virtual-functions 75. Exploring C++ Virtual Functions and Polymorphism - Medium, https://medium.com/@AlexanderObregon/exploring-c-virtual-functions-and-polymorphism-65cd38fbb70b 76. Polymorphism vs virtual function vs pure virtual function : r/cpp\_questions - Reddit, https://www.reddit.com/r/cpp\_questions/comments/rxa8w2/polymorphism\_vs\_virtual\_function\_vs\_pure\_virtual/ 77. C++ STL Tutorial: Most Frequently Used STL Containers & Algorithms | by Utkarshdhande, https://medium.com/@utkarshdhande1432/c-stl-tutorial-most-frequently-used-stl-containers-algorithms-bc1f4e684f34 78. Containers in C++ STL - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/containers-cpp-stl/ 79. Standard Template Library (STL) in C++ - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/the-c-standard-template-library-stl/ 80. C++ STL Tutorial - Tutorials Point, https://www.tutorialspoint.com/cplusplus/cpp\_stl\_tutorial.htm 81. Iterators in C++ STL - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/iterators-c-stl/ 82. C++ Vectors (With Examples) - Programiz, https://www.programiz.com/cpp-programming/vectors 83. Vector in C++ STL - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/vector-in-cpp-stl/ 84. AhmedMohamedAbdelaty/CPP-STL-beginners-guide: A beginner's guide to the C++ Standard Template Library (STL). - GitHub, https://github.com/AhmedMohamedAbdelaty/CPP-STL-beginners-guide 85. Map in C++ STL - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/map-associative-containers-the-c-standard-template-library-stl/ 86. C++ Map Explained with Examples - freeCodeCamp, https://www.freecodecamp.org/news/c-plus-plus-map-explained-with-examples/ 87. C++ Library - <map> - Tutorials Point, https://www.tutorialspoint.com/cpp\_standard\_library/map.htm 88. Set In C++ - A Complete Reference - Great Learning, https://www.mygreatlearning.com/blog/set-in-cpp/ 89. C++ STL: set (Complete Guide) - workat.tech, https://workat.tech/problem-solving/tutorial/cpp-stl-set-complete-guide-ugpnkuyov95m 90. Set in C++ STL - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/set-in-cpp-stl/ 91. C++ Sets and Their Operations | CodeSignal Learn, https://codesignal.com/learn/courses/mastering-complex-data-structures-in-cpp/lessons/cpp-sets-and-their-operations 92. C++ STL Containers - Programiz, https://www.programiz.com/cpp-programming/stl-containers 93. Mastering C++ Input and Output: A Comprehensive Guide to cout and cin - AlgoCademy, https://algocademy.com/blog/mastering-c-input-and-output-a-comprehensive-guide-to-cout-and-cin/ 94. Basic Input/Output - CPlusPlus.com, https://cplusplus.com/doc/tutorial/basic\_io/ 95. Understanding the Cin Object in C++: A Beginner's Guide - Simplilearn.com, https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-cin 96. Input/output with files - CPlusPlus.com, https://cplusplus.com/doc/tutorial/files/ 97. File Handling in C++ Programming - Starter tutorials, https://www.startertutorials.com/blog/file-handling-c.html 98. File Handling through C++ Classes - GeeksforGeeks, https://www.geeksforgeeks.org/cpp/file-handling-c-classes/ 99. C++ Files and Streams - Tutorials Point, https://www.tutorialspoint.com/cplusplus/cpp\_files\_streams.htm 100. Basic Input and Output in C - GeeksforGeeks, https://www.geeksforgeeks.org/c/basic-input-and-output-in-c/ 101. Comprehensive Guide to 'printf' and 'scanf' in C | by Dineshv - Medium, https://medium.com/@dineshvjrd/comprehensive-guide-to-printf-and-scanf-in-c-08af5711d1d6