Comparative Analysis of Programming Languages

C++ and Go

Design and Implementation of Programming Languages
CMSC 124
University of the Philippines Cebu

Instructor:
Asst. Prof. Paula E. Mayol

Submitted by:
Ishah Nicholei L. Bautista
Ceferino S. Jumao-as V
BS Computer Science

Date Submitted:
January 29, 2026

Abstract

This paper presents a comparative analysis of the programming languages C++ and Go. The discussion examines each language's purpose, history, paradigms, features, and evaluates them using established programming language evaluation criteria. The paper further compares language-specific features present in C++ but absent in Go, analyzing their implications and possible workarounds.

1 Description of Languages

2 Language A: C++

2.1 Purpose and Motivations

C++ is a general-purpose programming language designed for both high-level and low-level programming, it is considered an intermediate level language because of this. According to Britannica (2026), C++ is an extension of the programming language C, with it adding object-oriented programming support, while also maintaining the efficiency and low-level nature of its predecessor.

The primary reason for developing C++ was to provide a language that could offer the abstraction capabilities of Simula, without losing the efficiency and flexibility of C. According the its creator Bjarne Stroustrup, the aim was the support efficient execution, strong type checking, data abstraction, and object-oriented programming all in one language (Stroustrup, 2013). Nowadays, C++ is a language that allows programmers to achieve almost anything, allowing expressive and structured code while maintaining strict control over memory and resources.

Because of its characteristics, C++ is widely used in areas where efficiency and resource management are the top priority, such as operating systems, game engines, embedded systems, real-time simulations, and other more intensive operations (Britannica, 2026; Stroustrup, 2013).

2.2 History

C++ was created by Danish computer scientist Bjarne Stroustrup starting in 1979 while working at AT&T Bell Laboratories. The language started as an extension of C and was named ”C with Classes”, which introduced basic object-oriented features such as classes and encapsulation (Britannica, 2026). As mentioned in the previous section, C++ was influenced by Stroustrup’s experience with Simula, deemed as the first high-level object-oriented programming language but it lacked the performance needed for practical systems development.

The language was renamed ”C++” in 1983, a name that referenced the increment operator in C and symbolized the evolution and extension from the original language (Britannica, 2026). The publication of the book *The C++ Programming Language* in 1985 by the creator helped formalize the language and contributed significantly to its early adoption within the programming community (Stroustrup, 1985). Further revisions, including C++ 2.0, introduced features such as multiple inheritence and virtual functions, increasing what the language was capable of.

The first official international standard for the language was published in 1998 as ISO/IEC 14882:1998, commonly referred to as C++98 (Stroustrup, 2013). Since then, the language has continued to improve through a series of standard revisions, including C++03, C++11, C++14, C++17, C++20, and C++23. These updates added modern features that we are all aware of such as lambda expressions, improved type inferences, concurrency, and the enhanced standard libraries (Stroustrup, 2013). Despite the language being around for such a long time, its capabilities, support, and additional features have continued its relevance and widespread adoption in the industry and education.

2.3 Language Features

The C++ language is a direct improvement and extension of C, so much of its base features are derived from and similar to C. However, there are many changes and additions to the language to improve its features and extend its capabilities.

2.3.1 Core Language constructs

The first of the major notable features is the core constructs of the language, with its strongly and statically typed nature, it comes with various primitive types and derived types. Integers (int, float, char), floating (float, double), booleans, pointers, references, and arrays. For control flow, there are expressions and statements such as if/else, switch, and loops (for, while, do), these are what we mentioned before about C++ being imperative in nature due to its control constructs.

```
for (int i=0;i<10;i++) {
    if (i%2==0) continue;
}
```

2.3.2 Memory and Resource Management

The next neat thing about C++ is that it allows low-level programming with direct memory manipulation through raw pointers and manual allocation of memory. This is done through new/delete functions and pointer arithmetic. This makes C++ great for systems programming and performance-critical applications.

```
int *p = new int(42);
delete p;
```

2.3.3 Abstraction Mechanisms

The object-oriented nature of the C++ language allows for various methods for abstraction such as classes/structs with access control (public, private, protected), their own constructors/destructors, and copy/move semantics. Furthermore, the language also features inheritance (single and multiple), as well as polymorphism.

```
class C {
private:
    int x;
public:
    C(int v):x(v){} // constructor
    int get() const {
        return x;
    }
};
```

2.3.4 Templates and Generic Programming

The inclusion of templates enables generic programming as previously mentioned in C++. This makes it so that algorithms and data structures can be written completely independently of the types they operate on.

These are the foundation of the Standard Template Library (STL). GeeksforGeeks (2025) notes that this design allows code reuse without runtime overhead.

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

2.3.5 Standard Template Library (STL)

Directly following up the previous major feature, the STL is a collection of generic containers, algorithms, iterators, and function objects. According to cppreference (n.d.), it is designed around the idea of separating data structures from algorithms, improving modularity and code reuse.

Common STL components include:

- Containers (std::vector, std::map, std::set)
- Algorithms (std::sort, std::find)

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

std::vector<int> nums = {3, 1, 4};
std::sort(nums.begin(), nums.end());
```

2.3.6 Operator Overloading

This feature allows programmers to define or modify the behavior of certain operators for user-defined types. According to Stroustrup (2013), operator overloading is intended to improve expressiveness by allowing manipulation using familiar mathematical and logical notation.

What's interesting about operator overloading as a feature is that when implemented properly, it can enhance readability, especially for domains in mathematics; however, it can just as much be harmful to readability when abused.

```
class Vector2 {
public:
    int x, y;
    Vector2(int x, int y) : x(x), y(y) {}
    Vector2 operator+(const Vector2& other) const {
        return Vector2(x + other.x, y + other.y);
    }
};
```

In the provided example, the + operator is overloaded to allow intuitive addition of two Vector2 objects.

2.3.7 Concurrency and Parallelism

Ever since C++11, the language has added standardized support for concurrency. According to cppreference (n.d.), the ¡thread¿ library allows portable multithreaded programming access platforms

```
#include <thread>

void task() {
    // some concurrent work code
}

int main() {
    std::thread t(task);
    t.join();
}
```

The concurrency model prioritizes performance and control, but it means the responsibility of managing synchronization and other technical aspects is handled by the programmer themselves.

2.4 Paradigms

C++ supports many different programming paradigms, making it a multi-paradigm language. This makes it dynamic and flexible, meaning programmers can choose a single approach or mix and make use of multiple paradigms together. Below are some of the major programming paradigms that C++ supports.

2.4.1 Imperative Programming

Imperative programming is one of the foundational paradigms supported by C++. It involves writing sequences of statements that explicitly change the program state. According to UCL's research computing notes, imperative programs do this through assignments and control flows (loops, conditionals, etc.), and tell the machine how to perform computations step by step. Primarily, C++ is an imperative programming language, similar to its predecessor C.

The reason C++ is imperative is because it exposes memory and state, where there are variables representing memory locations, assignment statements that can change state, and control flow structures. According to Sebasta (2016), these are the characteristics of imperative languages. A sample of this can be seen below:

```cpp
#include <iostream>

int main() {
    int x = 5;
    int y = 3;
    int sum = x + y;
    x = sum; // state change of x
    std::cout << x << sum << std::endl;
    return 0;
}
```

In this sample implementation showcasing the imperative paradigm, x, y, and sum are variables whose state is changed by assignment, which is an imperative computation. The imperative nature is clear here in that each line represents an action or instruction that explicitly tells the computer what to do, which memory locations to manipulate/access, and how to compute them.

2.4.2 Procedural Programming

Procedural programming, a subtype of imperative programming, is based on the concept of breaking up programs into procedure calls. Procedural programming emphasizes modularization, using functions to separate code bound by scope. A procedural program is one that is made up of one or more of these modules (blocks of code). It is a kind of imperative programming because although imperative focuses on state changes, procedural focuses on organizing those changes into named units and groups, manipulated through procedure calls instead of pure inline code.

According to UCL (n.d), C++ is said to be procedural because it can support free functions (functions not bound by a class), does not inherently require the use of object-oriented programming, and parameters can pass state into procedures.

```cpp
int increment(int value) {
    return value + 1;
}
```

```

int main() {
    int x = 5;
    x = increment(x);  // state updated via procedure
}
```

In the above example, increment is a procedure/function, the state (x) is passed and modified through calling the procedure, and the code is modularized.

2.4.3 Object-Oriented Programming

According to Hemmendinger & David (2025), Object-oriented programming is a paradigm in which code is organized around objects, which combine state and behavior. OOP is defined by encapsulation, inheritance, and polymorphism. Objects are instances of classes which is how functions and data are encapsulated, these are helpful in keeping code organized, more intuitive, and easier to understand.

Unlike its predecessor C, which only had structs, C++ supports both structs and classes, with access control (public, private, or protected) for keeping code well maintained. Classes in C++ can feature inheritance, with some classes possibly being a ”subtype” or “child” of another, making it inherit the features of its parent. Another characteristic is polymorphism, according to GeeksforGeeks (2025), this allows a single function name to work differently in different implementations, with the same property applying to operators.

```
class Counter {
private:
    int value;  // encapsulated state

public:
    Counter() : value(0) {}
    void increment() { value++; }
    int get() const { return value; }
}
```

In the example code above, the state (value) is private, meaning only the counter class is able to access it directly. The behavior (increment) function controls the state mutation, allowing it to increase its value. And lastly, encapsulation enforces correctness, the only public elements of the class being the constructor, function for incrementing, and getting the state’s value, without direct access to actual variables, avoiding any mistakes.

2.4.4 Functional Programming

Functional programming is a paradigm where competition is treated as an evaluation of mathematical functions, avoiding mutable states and side effects. UCL (n.d..), explains that while C++ is not a functional language, it does support functional programming techniques such as first-class functions, lambda expressions, and higher-order functions.

A primary characteristic of functional programming is that it does not allow mutability of its variables, meaning they cannot be changed once their value has been assigned. C++ can be functional because although it has mutability, it does not require it, and can be enforced using the const keyword.

```
auto square = [](int x) { return x * x; };
int result = square(5);
```

The above code block is an example of a lambda expression that produces output solely from its input (x), there is no mutability so the value of x does not necessarily change.

2.4.5 Generic Programming

Generic programming emphasizes writing algorithms in terms of types that can be changed or different for the same value. It aims to express algorithms independent of specific representations, in abstract terms, relying solely on required operations.

C++ is a staple example of generic programming because of its features: allowing parametrization over types (using placeholders or templates), compile-time polymorphism, and most importantly, the Standard Template Library (STL), a collection of template classes and functions that allow data management using data structures like vectors, stacks, and maps (GeeksforGeeks, 2025).

```cpp
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}
```

As seen in the code above, the function uses a template, making it type independent, meaning it works with any comparable type.

2.5 Language Evaluation Criteria of C++

2.5.1 Simplicity

According to Sebesta (2016), simplicity refers to the size and conceptual complexity of a programming language; languages with fewer, clearer concepts are generally easier to learn, read, and use correctly.

C++ is not a simple language. It packs a lot, templates, multiple inheritance, operator overloading, move semantics and more. The good thing is that it does provide greater expressiveness and capabilities. But it also makes it harder to understand, features like template metaprogramming and rvalue references introduce rules and idioms that take time and practice to master (Sebesta, 2016; cppreference.com).

This affects:

Readability: Reduced → Basic code can be straightforward and simple, but much of real-world C++ combines many advanced features. Code that mixes templates, overloaded operators, and macros can be hard to understand immediately. Furthermore, even some compiler error messages can be difficult to comprehend.

Writability: Mixed → Experienced C++ programmers can write very compact, efficient code using the full feature set. Those new to coding and C++, however, can be overwhelmed by the language surface and make subtle mistakes.

Reliability: Mixed → Strong static typing improve safety when used well, but low-level features (raw pointers, manual memory management, UB) can hurt reliability if misused.

2.5.2 Orthogonality

Sebesta (2016) defines orthogonality as how consistently language features combine; more orthogonality means fewer special cases and unintended interactions.

C++ shows relatively low orthogonality: many features interact in nontrivial ways. For example, templates interact with overload resolution, multiple inheritance interacts with virtual base classes, and so on (cppreference.com).

This affects:

Readability: Reduced → You often need to understand several interacting features to predict behavior, which makes reading unfamiliar code harder.

Writability: Mixed → Experts can make use of the feature interactions tocreate powerful abstractions, but most programmers will struggle with creating combining features.

Reliability: Lower → Subtle interactions increase the chance of bugs and undefined behavior, especially when not explicitly understood.

2.5.3 Data Types

C++ has a very good data type system: the usual integers,floats, scalars, enums, classes/structs, unions, pointers, references, and even templates. This makes it so that real-world concepts can be modelled in a more precise manner. The biggest problem with this, however, is that C++ also allows many implicit conversions (numeric promotions, user-defined conversions) that can be a problem if not noticed. C++ has modern features like `explicit` and better type inference (`auto`) which aim to reduce these implicit conversions (Stroustrup, 2013; cppreference.com).
This affects:

Readability: Mixed → Strong, well-named types (e.g., `struct Money`) clarify intent, but implicit conversions can obscure what really happens at runtime.

Writability: Increased → The use of templates and the STL let you write reusable, concise code for many problems. Although having to specify data types and ensure the correct ones can be slightly tedious, it isn't too big of a deal.

Reliability: Mixed → Static typing catches many errors early, yet pointer-related types and implicit conversions can be a common sources of bugs.

2.5.4 Syntax Design

Syntax design considers how the language looks and how easy code is to read and write.

C++ has declarations (especially involving pointers, references, function pointers, or templates) can be hard to read. In more recent versions, newer constructs (`auto`, range-based `for`, uniform initialization) have improved the simplicity in implementing in the context of design, but the presence of older-form implementations (like those seen in C) can make it difficult to read and write.
This affects:

Readability: Decent → C++ code is generally not too difficult to read, scoping with curly braces, and arguments encased in parentheses make it easier to parse the syntax. However, template code or complex declarators can reduce this.

Writability: Mixed → Experienced developers can express ideas concisely; beginners may struggle a bit with the syntax, particularly with class declarations, functions, pointers, etc.

Reliability: Mixed → Complex syntax increases the chance of subtle errors. But strong syntax also improves how code is expected to behave, given that implicit actions do not take place.

2.5.5 Support for Abstraction

Support for abstraction is about the language mechanisms that let you hide implementation details and work at higher levels.

C++ is very good in terms of this, having functions, classes with access control, templates, and namespaces giving multiple ways to abstract. However, poorly designed abstractions (excessive operator overloading or poorly structured classes) can be bad.
This affects:

Writability: Increased → You can structure code to be modular and reusable across many domains.

Reliability: Increased → Abstractions such as RAII and smart pointers reduce common resource bugs, though leaky or wrong abstractions still cause problems.

2.5.6 Expressivity

C++ is a very expressive language. It has templates, operator overloading, constexpr, and metaprogramming that allow you to encode complex behavior at compile time, features that we have already mentioned in previous sections. This expressiveness is very beneficial for highly optimized libraries and DSLs embedded in C++, but it also makes it easy to write short code that hides its complexity.
This affects:

Writability: Increased → Experts can encode complex patterns succinctly; beginners can abuse expressiveness.

Reliability: Mixed → Compile-time expressiveness can preclude runtime bugs, but obfuscated complexity can obscure undefined behavior or unexpected interactions.

2.5.7 Type Checking

Type checking is when and how the language enforces type correctness.

C++ supports static type checking, meaning that most errors are caught at compile time. Templates are checked at instantiation, and more modern additions (C++20 concepts) have improved the expressiveness and error messages of template constraints. Nevertheless, the language allows for unsafe casts and low-level programming that can circumvent type safety when necessary (Stroustrup, 2013; cppreference.com).
This affects:

Reliability: Increased → Static typing prevents many runtime errors, but unsafe casts and undefined behavior remain risks.

2.5.8 Exception Handling

Exception handling is how a language indicates and handles errors.

C++ supports `try`/`catch`/`throw` and unwinds the stack during execution of destructors, making cleanup safe during exception handling. The `noexcept` keyword allows programmers to declare functions that do not throw exceptions, allowing certain optimizations and better function contracts. Exceptions is extremely useful for simplifying error handling but also introduce unexpected control flow if used excessively.
This affects:

Reliability: Increased → Exceptions allow clean up of resources safely; misusing exceptions (or relying on them for normal control flow) can lower reliability, but this is often more beneficial than detrimental.

2.5.9 Restricted Aliasing

Restricted aliasing deals with constraints on having multiple names that refer to the same memory location.

C++ allows extensive aliasing via pointers and references, and supports pointer arithmetic and `reinterpret_cast`, which are very powerful but also very hazardous. Violating aliasing assumptions leads to UB and optimization breaks; C++ does not have a standard `restrict` keyword, but compiler extensions and the programmer's choice of higher-level abstractions can be used to sidestep aliasing problems.

Reliability: Reduced → Unsafe aliasing leads to UB; references, smart pointers, but proper programming practices can help to contain the problem.

3 Language B: Go

3.1 Purpose and Motivations

Go was explicitly designed to solve the specific challenges of large-scale programming encountered at Google. The language's creators were driven by frustration with the complexity, slow compilation times, and the increasing difficulty of maintaining massive codebases in traditional languages like C++ and Java (Pike, 2012).

A primary motivation for Go was to bridge the gap between efficiency and ease of use. The goal was to create a language that combined the performance and safety of a compiled, statically typed language like C++ with the readability and developer velocity of an interpreted, dynamic language like Python (Donovan & Kernighan, 2015). Additionally, the designers focused heavily on build speed to improve developer productivity; at the time, C++ builds at Google were taking up to 45 minutes, so Go was engineered to compile almost instantaneously (Pike, 2012).

Finally, Go was designed from the ground up to handle modern multicore networking systems. To address this, features like goroutines and channels were added to the core language to make concurrent programming easy and robust, effectively replacing the complexity of managing heavy operating system threads (Pike, 2012).

3.2 History

The language was conceived in 2007 by three veteran computer scientists at Google: Robert Griesemer, known for his work on the V8 JavaScript engine; Rob Pike, a member of the Unix team and co-creator of UTF-8; and Ken Thompson, the creator of B, C, Unix, and UTF-8 (Pike, 2012).

The design process began in September 2007 as a ”20% project” at Google, leading to the language's open-source release in November 2009 (Pike, 2012). A significant milestone was reached in March 2012 with the release of Go 1.0, which established the ”Go 1 Compatibility Promise,” guaranteeing that code written for Go 1.0 would continue to compile and run for the lifetime of the Go 1.x series (Donovan & Kernighan, 2015). Major revisions followed, including Go 1.5 in 2015, where the compiler was rewritten entirely in Go to remove the last C code from the toolchain, and Go 1.18 in 2022, which introduced Generics (Type Parameters) to address the community's most significant request (The Go Authors, 2022). As of August 2025, Go has reached version 1.25, continuing its focus on supply chain security and cloud-native performance (The Go Authors, 2025).

Go has effectively become the language of the cloud, serving as the core language for major infrastructure projects such as Docker and Kubernetes (Donovan & Kernighan, 2015). It is widely adopted for microservices, API backends, and high-performance distributed systems by major technology companies like Netflix, Uber, and Dropbox, the latter of which migrated critical parts of its architecture from Python to Go to improve efficiency (Donovan & Kernighan, 2015).

3.3 Language Features

Go (Golang) supports features like built-in concurrency (goroutines & channels), compilation, automatic garbage collection, and static typing. It also incorporates built-in tools for formatting and testing.

3.3.1 Concurrency (Goroutines & Channels)

In many languages like Java or C++, creating a thread is heavy, it eats up a lot of memory. In Go, we use Goroutines (Pike, 2012). They are green threads managed by the Go Runtime, not the OS. According to Donovan and Kernighan (2015) They are incredibly cheap, starting at 2KB stack size, so it can spin up thousands of them. To stop these thousands of threads from crashing into each other (race conditions), they use Channels to pass data safely.

```go
// Create a Channel
messageChannel := make(chan string)
// Start a Goroutine
go func() {
    // Simulate some heavy work
    time.Sleep(1 * time.Second)
    // Send data INTO the channel
    messageChannel <- "Processed Data from Worker"
}()
// Receive data from The Channel
msg := <-messageChannel
```

3.3.2 Automatic Garbage Collection

Go has a Garbage Collector that runs in the background. It watches your variables. Pike (2012) stated that when a variable is no longer reachable or no one is using it, the GC sweeps it away and reclaims the RAM. Go's GC is optimized for Low Latency where it tries to pause the program for less than 1 millisecond.

3.3.3 Static Type (with Type Inference)

Go is Statically Typed, meaning the compiler knows the type of every variable before the code runs (Sebesta, 2019). It cannot treat an integer like a string. This catches entire classes of bugs at compile time. However, to avoid being verbose like `int x = 10;`, Go uses Type Inference (`:=`). It writes code that looks dynamic, but is actually strict.

```go
// Explicit typing
func Add(a int, b int) int {
  return a + b
}
```

```go
// Type Inference: Go figures out x is an 'int'
x := 10
```

3.3.4 Built-in Tooling

Go developers don't argue about code style. The language includes a standard formatter (`go fmt`) that rewrites your code to the official style automatically (Pike, 2012). It also includes a testing framework (`go test`) right in the standard library. It doesn't need to install JUnit or Jest, it's just there (The Go Authors, 2025).

```go
package main
// The Code
func Multiply(a, b int) int {
  return a * b
}

// The Test
import "testing"

// Test functions must start with "Test" and take *testing.T
func TestMultiply(t *testing.T) {
  result := Multiply(2, 3)
  expected := 6

  if result != expected {
    // Native error reporting
    t.Errorf("Expected %d, but got %d", expected, result)
  }
}
```

3.3.5 Composition (Struct Embedding)

Go uses Composition (”A Car has an Engine”) via a feature called Struct Embedding. It places one struct inside another. If one embed a struct without giving it a name, Go promotes its fields and methods to the outer struct (The Go Authors, 2025). One can call them as if they belonged to the outer struct directly. This gives it the convenience of inheritance without the complexity.

```go
// The Base or inner part
type Engine struct {
  Horsepower int
  Type       string
}

func (e *Engine) Start() {
  fmt.Println("Vroom!")
}

// The Container or outer part
type Car struct {
  // We list "Engine" without a field name.
  Engine
  Model string
}
```

3.4 Paradigms

Go (Golang) is a multi-paradigm language, primarily a combination of imperative and procedural styles. It has strong support for concurrency, and a unique, composition-based form of object-oriented programming. It also has elements of functional programming with first-class functions and generics.

3.4.1 Imperative & Procedural Programming

Go is fundamentally an imperative language, meaning it focuses on modifying the program state through statements and detailed steps (Sebesta, 2019). It descends directly from the C family, emphasizing procedural programming where code is grouped into procedures (functions) that manipulate data structures (Donovan & Kernighan, 2015).

According to Pike (2012), Go was designed to remove the ”header file” complexity of C++ while retaining the straightforward, step-by-step clarity of C. This makes it highly readable because the control flow is explicit, there are no hidden setters or getters executing behind the scenes.

```go
package main

import "fmt"

```

```go
func main() {
  sum := 0

  // A procedural loop
  for i := 1; i <= 5; i++ {
    sum = sum + i
  }

  fmt.Println("Total:", sum)
}
```

In this snippet, the code explicitly defines how to calculate the result (Donovan & Kernighan, 2015). We initialize a variable sum (state) and then modify that state five times inside a loop. This is the definition of imperative programming: step-by-step state mutation.

3.4.2 Concurrent Programming

While most languages treat concurrency as an operating system detail (threads), Go treats it as a core language paradigm. It is based on Communicating Sequential Processes (CSP), a theory developed by C.A.R. Hoare (Donovan & Kernighan, 2015).

As described by the Go Authors (2025), the language provides goroutines (functions executing independently) and channels (typed conduits for synchronization). This paradigm shifts the focus from managing locks (mutexes) to managing communication, which Pike (2012) argues drastically reduces the complexity of writing parallel software.

```go
package main

import "fmt"

func main() {
  // Create a channel to send text messages
  messages := make(chan string)

  // Start a new concurrent thread
  go func() {
    messages <- "Hello from the background!"
  }()

  // Wait here until a message arrives
  msg := <-messages
  fmt.Println(msg)
}
```

The `go` keyword immediately launches the function into a separate timeline (goroutine) so it runs at the same time as `main`. The `messages` channel acts as a synchronizer. The main function pauses at `<-messages` until the background goroutine finishes sending data, ensuring they coordinate perfectly without using complex locks (Donovan & Kernighan, 2015).

3.4.3 Object-Oriented Programming

Go is object-oriented, but not in the traditional sense of classes and hierarchies. Sebesta (2019) classifies Go as supporting OOP through Data Abstraction (structs), Encapsulation (package-level visibility), and Polymorphism (interfaces).

However, Go explicitly rejects class-based inheritance. Instead, it uses Composition via struct embedding. According to Donovan and Kernighan (2015), this encourages developers to build small, reusable components rather than deep, fragile inheritance trees. Polymorphism is achieved implicitly: a type implements an interface simply by having the matching methods, without declaring `implements` (Pike, 2012).

```go
package main

import "fmt"

type Engine struct {
  Horsepower int
}

type Car struct {
  Engine // Embedding the struct directly
  Model  string
}

func main() {
  c := Car{
    Model:  "Mustang",
    Engine: Engine{Horsepower: 450},
  }

  // We access 'Horsepower' directly
  fmt.Println(c.Horsepower)
}
```

Notice that `Car` does not have a `Horsepower` field declared explicitly. Because we embedded Engine inside `Car` (by listing the type name `Engine` alone), Go allows us to say `c.Horsepower` directly. This mimics inheritance (sharing code) but keeps the architecture simple and modular (Pike, 2012).

3.4.4 Functional Programming

Go is not a pure functional language, but it incorporates functional features to increase expressivity. Functions in Go are First-Class Citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions (The Go Authors, 2025).

With the introduction of Generics in Go 1.18, functional patterns like `Map`, `Filter`, and `Reduce` have become practical. This allows developers to write logic that is declarative and reusable across different data types (Donovan & Kernighan, 2015).

```go
package main

import "fmt"

func main() {
  // Assign a function to a variable
  multiplier := func(x int) int {
    return x * 2
  }

  // Pass that function variable to another function
  result := apply(10, multiplier)

  fmt.Println(result)
}

// A function that accepts ANOTHER function as input
func apply(val int, action func(int) int) int {
  return action(val)
}
```

Here, `multiplier` is not just a block of code; it is a value stored in a variable. We pass this variable into the `apply` function. This demonstrates functional behavior because the logic (the multiplication) is passed around just like a piece of data (Donovan & Kernighan, 2015).

3.5 Language Evaluation Criteria

3.5.1 Simplicity

Go is designed to be explicitly minimalist. It intentionally omits many features found in other modern languages, such as pointer arithmetic, implementation inheritance, method overloading, and complex metaprogramming. The designers prioritized a small feature set to ensure that code remains transparent, meaning a programmer can look at a block of code and understand its execution without worrying about hidden behaviors or side effects (Pike, 2012; Donovan & Kernighan, 2015).

Readability: High → Its design is explicitly minimalist, removing complex features like pointer arithmetic, inheritance, and method overloading. This reduced feature set means a programmer can read almost any Go code and understand exactly what it does without worrying about hidden behaviors.

Writability: Moderate → The simplicity comes at the cost of verbosity. The language lacks implicit behaviors, so one often has to write more lines of code to accomplish simple tasks.

Reliability: High → Simplicity directly aids reliability because there are fewer complex interactions for the programmer to misunderstand, reducing the surface area for bugs.

3.5.2 Orthogonality

Go exhibits moderate orthogonality. Its type system allows methods to be defined on almost any named type, not just classes, which provides a consistent way to add behavior to data. However, Go compromises on pure orthogonality for the sake of practicality; for example, standard built-in functions like `len` and `cap` operate on specific built-in types (slices, maps, channels) in ways that user-defined functions cannot replicate (The Go Authors, 2025; Donovan & Kernighan, 2015).

Readability: Moderate → Features like methods are orthogonal, one can define methods on almost any type, not just classes. This consistency helps reading. However, it has exceptions: standard built-in functions (like len or cap) work on specific types in ways user-defined functions cannot, which breaks pure orthogonality.

Writability: Moderate → The lack of perfect orthogonality historically hindered writability, forcing developers to copy-paste code for different types.

Reliability: High → The orthogonality of the type system improves reliability by decoupling components, allowing easier testing and swapping of implementations.

3.5.3 Data Types

Go utilizes a strong, static typing system that includes primitive types (integers, floats, booleans) and composite types (structs, arrays, slices, maps). While strict about type safety to prevent errors, Go includes features like type inference (e.g., `x := 10`) to reduce verbosity. This balance ensures that the data structure of a variable is always explicit and known at compile time, yet the code remains relatively concise (The Go Authors, 2025).

Readability: High → Go uses strong, static typing, which improves readability by making data structures explicit. One always knows what a variable contains.

Writability: High → The type system is strict but includes type inference (e.g. `x := 10` instead of `var x int = 10`), which improves writability by reducing keystrokes while maintaining safety.

Reliability: High → Go's type safety feature ensures that type mismatches are caught at compile time, preventing runtime crashes.

3.5.4 Syntax Design

Go's syntax is engineered for extreme clarity and consistency. It removes visual noise by eliminating the need for semicolons (which are inserted automatically) and parentheses around control flow conditions. Furthermore, the language enforces a rigid formatting style through the go fmt tool, ensuring that all Go code, regardless of the author, looks visually identical, which streamlines code review and maintenance (Pike, 2012).

Readability: High → Go's syntax is designed for clarity. It removes visual noise like semicolons and parenthesis around if/for conditions. The declaration syntax (var name type) reads left-to-right.

Writability: Moderate → The syntax is rigid which might annoy some writers initially but ensures all code looks the same, speeding up writing in teams.

Reliability: High → The syntax enforces discipline. For example, the compiler throws an error if you import a library or declare a variable but do not use it. This catches potential bugs or dead code immediately.

3.5.5 Support for Abstraction

Go supports process abstraction through functions and data abstraction through structs and interfaces. Crucially, it rejects the traditional Object-Oriented model of implementation inheritance (classes and sub-classes). Instead, it relies on composition (struct embedding) and interfaces to define behavior. This forces developers to design systems based on what objects do (interfaces) rather than what they are (hierarchy), avoiding the complexity of fragile base classes (Donovan & Kernighan, 2015).

Writability: High → Go supports abstraction via Interfaces and Composition (embedding structs), but explicitly rejects implementation inheritance (classes, extends). This forces a different style of writing (composition over inheritance) which proponents argue is more maintainable.

Reliability: High → By avoiding deep inheritance hierarchies, Go avoids the "fragile base class" problem, making long-term maintenance more reliable. Interfaces allow you to define behavior contracts (what an object does) rather than what it is, which simplifies testing with mocks.

3.5.6 Expressivity

Go deliberately opts for low expressivity in many areas, preferring explicit code over concise, implicit one-liners. It lacks many of the functional operators found in other languages (like concise map or filter syntax in early versions), often requiring developers to write out full loops and explicit error handling checks (if err != nil). This design choice prioritizes the visibility of control flow over brevity (Pike, 2012).

Writability: Low → It prefers explicit code over concise, implicit one-liners. One often has to write boilerplate that more expressive languages would hide.

Reliability: High → This trade-off is made for reliability. High expressivity often leads to implicit code that is hard to debug. Go's verbosity ensures that the control flow is always visible, making it easier to verify correctness.

3.5.7 Type Checking

Go enforces strict compile-time type checking. It refuses to compile code that contains mismatched types, such as attempting to add a number to a string, unless an explicit conversion is performed. This rigorous checking at the compilation stage prevents a vast category of runtime errors that are common in dynamically typed languages (The Go Authors, 2025).

Reliability: High → It refuses to compile code with mismatched types, which is more reliable compared to dynamically typed languages.

3.5.8 Exception Handling

Go diverges from the traditional `try-catch` model used in C++ or Java. Instead, it treats errors as standard return values that must be checked and handled explicitly immediately after they occur. For truly unrecoverable errors, Go provides a `panic` and `recover` mechanism, but idiomatic Go heavily favors explicit value-based error handling to ensure that no failure state is ignored (Donovan & Kernighan, 2015).

Reliability: High → Treats errors as values that must be handled explicitly (return result, error). This forces the programmer to consider failure states at every step, increasing the reliability by preventing ignored errors.

3.5.9 Restricted Aliasing

Go allows aliasing through the use of pointers, enabling different parts of a program to reference the same data structures for efficiency. However, unlike C or C++, Go mitigates the dangers of aliasing through memory safety and automatic garbage collection. This ensures that a pointer cannot reference memory that has already been freed, preventing dangling pointer errors (The Go Authors, 2025).

Reliability: High → While unrestricted aliasing is generally considered dangerous for reliability, Go mitigates this with Memory Safety and Garbage Collection. You cannot accidentally free memory that another pointer is using, significantly increasing reliability.

4 Feature Comparison

4.1 Feature 1: Operator Overloading

4.1.1 Description of Feature in C++

According to cppreference (n.d.), operator overloading allows the customization of C++ operators for user-defined types.

The following code uses operator overloading on the "+" operator for C++, basically allowing it to perform addition on two vector objects defined in the struct. The vector is a user-defined type, and the overloading may allow an operation to look like a primitive addition, however, it is actually a function call.

```cpp
#include <iostream>

struct Vector {
    int x;

    // Operator overloading
    Vector operator+(const Vector& other) const {
        return { x + other.x };
    }
};

int main() {
    Vector a{3};
    Vector b{5};

    Vector c = a + b; // uses operator+
    std::cout << c.x << std::endl; // 8
}
```

4.1.2 Advantages of Having the Feature

- Improves readability when the operation is intuitive, such as arithmetic on vectors or complex numbers.
- Allows user-defined types to behave similarly to built-in types.
- Enables concise and expressive syntax (e.g., `a + b` instead of `a.Add(b)`).
- Can reduce boilerplate code in mathematically oriented programs.

4.1.3 Advantages of Not Having the Feature

- Prevents hidden or non-obvious behavior behind familiar operators.
- Makes performance costs and side effects more explicit.
- Improves code readability for developers unfamiliar with a codebase.
- Simplifies the language specification and compiler error messages.

4.1.4 Implementing Similar Functionality in Go

Go explicitly rejects this feature to prioritize readability and transparency. The design philosophy is that an expression like `v1 + v2` is ambiguous: it hides the cost of the operation. A developer reading the code cannot immediately tell if + performs a simple addition, triggers a heavy memory allocation, or executes a complex algorithm. To eliminate this uncertainty, Go forces developers to use Explicit Method Chains (Pike, 2012). Instead of hijacking symbols, we write named methods that clearly describe the action.

```go
type Vector3 struct {
  X, Y, Z float64
}

func (v Vector3) Add(other Vector3) Vector3 {
  return Vector3{
    X: v.X + other.X,
    Y: v.Y + other.Y,
    Z: v.Z + other.Z,
  }
}
```

In C++, this would be an `operator+` overload. In Go, it is a standard function named `Add`. The `(v Vector3)` part before the function name is the receiver, which attaches this function to the `Vector3` struct, effectively making it a method. It takes another vector as input, adds the components explicitly, and returns a new `Vector3` result.

```go
func main() {
  v1 := Vector3{1, 2, 3}
  v2 := Vector3{4, 5, 6}

  // Workaround: Explicit Method Call
  v3 := v1.Add(v2)
}
```

We instantiate two vectors (`v1` and `v2`) using struct literals. Instead of using a + symbol, we call `.Add(v2)` on `v1`. This makes the control flow obvious where the reader knows exactly which function is being executed, ensuring there are no hidden performance costs or side effects.

4.2 Feature 2: Class Inheritance

4.2.1 Description of Feature in C++

Class inheritance in C++ is a feature for object-oriented programming which allows new classes to inherit the properties and characteristics of an existing class. This is done for improved incrementing changes to classes, enforcing an *is-a* relationship from child to parent class (cppreference, n.d.).

The code below demonstrates class inheritance through how the different classes are implemented. The base class animal has a public function: `speak()`, with a generic animal sound output. On the other hand, a class Dog also has a function `speak()`, which prints our dog barks, a function with the same name but different outcome.

The class Dog is a derived class from Animal, which means that whatever functions, properties, variables, and such in the base class are present, the child class also gets. Therefore, if the function class were not to have its own `speak()` function, it would still be possible to call it, but the result would take from its base class.

```cpp
#include <iostream>

// Base class
class Animal {
public:
    void speak() {
        std::cout << "Animal sound\n";
    }
};

// Derived class
class Dog : public Animal {
public:
    void speak() {
        std::cout << "Dog barks\n";
    }
};

int main() {
    Dog d;
    d.speak(); // calls Dog::speak
}
```

4.2.2 Advantages of Having the Feature

- Promotes code reuse by allowing shared behavior in base classes.
- Naturally models *is-a* relationships found in real-world systems.
- Supports polymorphism and method overriding.
- Helps organize large systems using hierarchical class structures.

4.2.3 Advantages of Not Having the Feature

- Reduces tight coupling between types.
- Avoids issues such as the fragile base class problem.
- Encourages composition over inheritance, leading to more flexible designs.
- Makes data flow and behavior easier to trace and reason about.

4.2.4 Implementing Similar Functionality in Go

Go explicitly rejects class-based inheritance to avoid this tight coupling. Instead, it employs a design philosophy based on Composition, or a ”Has-A” relationship. To achieve the convenience of inheritance without its architectural downsides, Go uses a feature called Struct Embedding. By embedding a parent struct inside a child struct, Go provides Promoted Fields and methods. This allows developers to access the inner struct's fields directly (syntactic sugar), making it look like inheritance, while structurally it remains a simple case of one struct holding another (Donovan & Kernighan, 2015).

```go
// The Base or Component
type Animal struct {
  Name string
}

func (a *Animal) Speak() {
  fmt.Printf("%s speaking...", a.Name)
}

func (a *Animal) Walk() {
  fmt.Println("Walking...")
}
```

This defines the base component, `Animal`. Go uses a `struct` to hold data (`Name`) and attaches methods (`Speak`, `Walk`) to it using receivers. These methods represent the reusable behaviors that can be ”mixed in” to other types.

```go
// The Child or Container
type Dog struct {
  // By listing "Animal" without a name, we embed it.
  Animal
  Breed string
}
```

This demonstrates Struct Embedding, Go’s alternative to inheritance. Inside the Dog struct, we list the Animal type but do not give it a field name (like info Animal). This tells the Go compiler to ”embed” Animal. The Dog struct now ”has” an Animal inside it, but due to embedding, the properties of Animal will be promoted to the top level of Dog.

```go
// Overriding or Shadowing Logic
// If Dog defines Speak(), calling dog.Speak() hits this method first
// shadowing the inner Animal.Speak().
func (d *Dog) Speak() {
  fmt.Printf("%s barks loudly...", d.Name)
}
```

This illustrates Method Shadowing. In Go, since we define a Speak method specifically on Dog, it takes precedence over the embedded Animal.Speak(). When Speak is called on a Dog, Go finds this specific method first and executes it, effectively shadowing the generic behavior of the embedded struct.

```go
func main() {
  // Initialization
  myDog := Dog{
    Animal: Animal{Name: "Papi"},
    Breed:  "Golden Retriever",
  }

  // Feature 1: Field Promotion
  fmt.Println(myDog.Name)

  // Feature 2: Method Promotion
  myDog.Walk()

  // Feature 3: Overriding or Shadowing
  myDog.Speak()
}
```

This code demonstrates the three key results of embedding. First is the Field Promotion, where we access myDog.Name directly. The compiler automatically forwards this to myDog.Animal.Name. Next is Method Promotion, where we call myDog.Walk(). Since Dog does not have its own Walk method, the compiler forwards the call to the embedded Animal.Walk(). Lastly Shadowing, where We call myDog.Speak(). Because Dog has its own version, that one is executed (”barks loudly”) instead of the generic Animal version.

References

Amadi, N. (2023, June). *Exploring go's unique approach to object-oriented programming*. https://dev.to/nonsoamadi10/exploring-gos-unique-approach-to-object-oriented-programming-115i

Berger, C. (n.d.). *Go is...? (beyond paradigms)* [n.d.]. Applied Go. https://appliedgo.net/spotlight/beyond-paradigms/

Built In. (2025, September). *What is golang? (definition, features, vs. other languages)*. https://builtin.com/software-engineering-perspectives/golang

cppreference.com. (n.d.). *C++ reference*. https://en.cppreference.com

Donovan, A. A. A., & Kernighan, B. W. (2015). *The go programming language*. Addison-Wesley.

GeeksforGeeks. (2025a, September). *Standard template library (stl) in c++*. https://www.geeksforgeeks.org/cpp/the-c-standard-template-library-stl/

GeeksforGeeks. (2025b, October). *Polymorphism in c++*. https://www.geeksforgeeks.org/cpp/cpp-polymorphism/

Hemmendinger, D. (2025, December). *Object-oriented programming*. https://www.britannica.com/technology/object-oriented-programming

Kuree. (n.d.). *The go programming language report* [n.d.]. GitBook. https://kuree.gitbooks.io/the-go-programming-language-report/content/2/text.html

Osman, A. (2025, July). *Understanding go's type system: A complete guide to interfaces, structs, and composition [2025]*. https://dev.to/arasosman/understanding-gos-type-system-a-complete-guide-to-interfaces-structs-and-composition-2025-3an

Pike, R. (2012). Go at google: Language design in the service of software engineering.

Sebesta, R. W. (2016). *Concepts of programming languages* (11th ed.). Pearson.

Sebesta, R. W. (2019). *Concepts of programming languages* (12th ed.). Pearson.

Singh, A. (n.d.). *Why procedural programming still matters: Understanding go vs java* [n.d.]. Medium. https://medium.com/@singhalok641/why-procedural-programming-still-matters-understanding-go-vs-java-90fcdde666eb

Stroustrup, B. (1985). *The c++ programming language* (1st ed.). Addison-Wesley.

Stroustrup, B. (2013). *The c++ programming language* (4th ed.). Addison-Wesley.

The Go Authors. (n.d.-a). *Frequently asked questions (faq)* [n.d.]. The Go Programming Language. https://go.dev/doc/faq#Is_Go_an_object-oriented_language

The Go Authors. (n.d.-b). *The go programming language* [n.d.]. https://go.dev/

The Go Authors. (2025a). *The go programming language specification*. Google.

The Go Authors. (2025b). *The go programming language specification*. The Go Programming Language. https://go.dev/ref/spec#Statements

This is DASC. (2020, July). *Seven golang features you must know about*. Medium. https://medium.com/@thisisdasc/seven-golang-features-you-must-know-about-944485d413fe

Twilio. (2024, July). *Error handling in go: 6 effective approaches*. https://www.twilio.com/en-us/blog/developers/community/error-handling-go-6-effective-approaches

University College London. (n.d.). *Programming paradigms*. https://github-pages.ucl.ac.uk/research-computing-with-cpp/05libraries/ProgrammingParadigms.html

Amadi, 2023 Berger, n.d. Built In, 2025 Kuree, n.d. Osman, 2025 Singh, n.d. Donovan and Kernighan, 2015 The Go Authors, n.d.-a The Go Authors, n.d.-b The Go Authors, 2025b This is DASC, 2020 Twilio, 2024 Donovan and Kernighan, 2015 Pike, 2012 Sebesta, 2019 The Go Authors, 2025a University College London, n.d. Hemmendinger, 2025 GeeksforGeeks, 2025b GeeksforGeeks, 2025a cppreference.com, n.d. Stroustrup, 2013 Stroustrup, 1985 Sebesta, 2016