Comparing C and C++

C and C++ are not only important programming languages but have also had big effect on the development of other languages. C, developed by Dennis Ritchie at Bell Labs, was designed to implement the Unix operating system. It quickly gained popularity and became the de facto language for systems programming, demonstrating the potential for a high-level language to interact closely with hardware. The simplicity and efficiency of C inspired the creation of many subsequent programming languages, including C++.

C++, created by Bjarne Stroustrup, was developed as an extension of C to introduce object-oriented features. It was first known as "C with Classes" and evolved into a powerful language for software development. C++'s adoption of OOP concepts paved the way for other object-oriented languages like Java and C#, which have found broad use in web and application development.

Both languages are renowned for their efficiency, flexibility, and strength, but they differ in significant ways, particularly in terms of syntax and semantics. In this essay, we will conduct a thorough analysis of C and C++, with a focus on their syntax and semantics, highlighting both their similarities and differences. But before I want to compare them with Language Evolution Criteria.

For **readability**, C has maybe the simplest syntax and because it doesn't have too many features it doesn't have many keywords, so this makes C a readable language. For C++ because it adds more feature to standard C, it is less readable than C.

For **writability**, C becomes the winner again. Because C is smaller and simpler than C++. Simplicity improves read/writability. For example, operator overloading can lead confusion in reading and writing. So, C++ is more complex and harder to write. At this point I want to mention orthogonality term. A language is orthogonal if its features are built upon a small, mutually independent set of primitive instructions. From this definition we understand that orthogonality improves read/writability.

If we consider **reliability**, (what we mean from reliability is for example, preventing memory leaks, working same in different platforms etc.) C is less reliable than C++ because it can't manage memory leaks itself. And, for the exception handling C has no feature.

And lastly if we consider cost, C is cheaper than C++ because C++ has more features and more platforms to build on, so this increases the programmer cost in general.

After the language evolution criteria let's talk about syntax and semantics.

Syntax

Because of C++ builds upon C's syntax their syntax rules don't have so many differences. In most cases C++ add more rules to basic C syntax.

The **keywords** are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. In C we have 32 supported keywords however C++ add 63 more and supports 95 keywords. Examples include "class", "private", "public", "protected", "virtual", "new", "delete", "try", "catch", "throw" and "namespace". C++ introduces access control keywords like private, protected, and public to specify the visibility of class members. These keywords are not present in standard C.

Identifiers are used as the general terminology for the naming of variables, functions, arrays and classes. In programming languages there are some rules when user defines it. These rules are same in both C and C++ for example an identifier must differ from any keyword and must start with a letter or an underscore. Also, there is no limit in character, but some compilers can put a limit on it.

The **constants** refer to the variables with fixed values. They are like normal variables but with the difference that their values cannot be modified in the program once they are defined. Both in C and C++ we use "const" keyword to define constant variable. However, we can use "#define" preprocessor directive to define constants.

In C **strings** are nothing but an array of characters ended with a null character ("/0"). This null character indicates the end of the string. In C++, a string is not array of characters like in C. It is a

class available in the STL library which provides the functionality to work with a sequence of characters, that represents a string of text.

Special symbols are the token characters having specific meanings within the syntax of the programming language. These symbols are used in a variety of functions, including ending the statements, defining control statements, separating items and more. In C and C++ there is no difference in special symbols. One of the most fundamental and shared aspects of syntax between C and C++ is the use of the semicolon (;) as the statement terminator. Both languages employ semicolons to separate statements within the code. Another shared aspect of syntax in C and C++ is the use of curly braces ({}) to define code blocks. Code blocks group multiple statements together and are used in various contexts, such as functions, loops, and conditional statements.

Operators are symbols that trigger an action when applied to variables and other objects. The data items on which operators act are called operands. C++ has almost all operator C has. However, in C++ most of them can be overloaded. And C++ has some different operators like scope resolution ("::") and three-way comparison ("<=>"). In C++, when handling exceptions using try, catch, and throw, the question mark ("?") and colon (":") are used to specify the exception type and to catch specific exceptions. These are not part of the standard C syntax. In C++, the ampersand ("&") is used to declare reference variables, allowing for more efficient and expressive code. In C, this is typically used as the address-of operator.

Additionally, C++ introduces literals like "true", "false" and user-defined literals that are not part of standard C. These are used to enhance type safety and expressiveness in C++. Also in C++, the "this" pointer is used to refer to the current object within a class. C has no equivalent concept. C++ introduces function overloading, where multiple functions with the same name but different parameter lists can coexist. Also, in C++ there are "typeid" and "dynamic_cast" operators for dynamic type checking and casting in the context of inheritance and polymorphism. These operators have no counterparts in standard C.

Semantics

One of the most fundamental differences between C and C++ lies in their programming paradigms. C is primarily a **procedural programming language**, focusing on the execution of procedures, functions, and a linear flow of control. In contrast, C++ is an **object-oriented programming** (OOP) language, emphasizing the organization of code around objects, classes, and the principles of encapsulation, inheritance, and polymorphism.

In C, the primary building blocks of code are functions. The main function serves as the entry point, and the program follows a **top-down approach**. It is basically dividing problem into parts and parts to make it easier to solve. It is well-suited for small to medium-sized projects, where a clear sequence of operations is sufficient.

C++, on the other hand, introduces the concept of classes, which encapsulate data and behavior within objects. These objects can interact with each other, offering a more modular and organized approach (especially **bottom-up approach**) to code. In bottom-up approach, we solve smaller problems and integrate it as whole and complete the solution. The use of classes and OOP makes C++ particularly suitable for large-scale software development, as it allows for code reuse and a more structured design.

C++ introduces the concept of constructor and destructor functions, which are executed when objects are created and destroyed, respectively. Constructors are responsible for initializing an object's state, while destructors clean up resources, making C++ well-suited for resource management.

In C, there are no equivalent mechanisms for automatic resource management. Memory allocation and deallocation are performed manually using functions like malloc and free. This manual memory management can be error-prone and challenging to maintain in complex applications.

C++ offers **inheritance** as part of object-oriented programming, allowing classes to be derived from other classes. This establishes an "is-a" relationship and provides control over the reuse and extension of class behavior.

Polymorphism is a key object-oriented concept in C++. It allows control over the selection of a method to execute based on the actual type of an object at runtime. This is achieved through virtual functions and dynamic binding.

While many object-oriented controls are introduced in C++, there are some control structures in C that are not part of the C++ language.

C supports the goto statement, which allows for unstructured control flow by transferring control to a labeled statement. C++ discourages the use of “goto” in favor of more structured control constructs.

C allows the creation of compound literals, which are used to group multiple values into a single expression. C++ does not support this feature.

C supports variable-length arrays, which allow the declaration of arrays with sizes determined at runtime. C++ does not support VLAs.

As I already mentioned in syntax part, one of the distinctive features of C++ is the ability to overload operators. In C++, operators like +, -, *, and / can be customized for user-defined types, allowing for more expressive and natural code. This operator overloading simplifies code and enhances readability.

In C, operators have fixed behaviors defined by the language, and there is no way to redefine their functionality for custom data types. This limitation can lead to less intuitive code when working with user-defined structures.

Again, as I said earlier, C++ provides built-in support for exception handling using try, catch, and throw blocks. Exception handling allows for the graceful management of errors, making code more robust and reliable. Developers can catch exceptions, handle errors, and ensure that resources are properly released, even in the presence of failures.

In C, as I said earlier exception handling is not natively supported. Instead, error management is typically done using return codes and manual checks. This approach can lead to more error-prone and less maintainable code, as developers need to explicitly check for errors after each function call.

C++ offers stronger type safety compared to C. Type safety is a crucial aspect of language semantics, as it affects the predictability and reliability of code. In C++, the type system is more robust, reducing the likelihood of type-related errors.

For example, C++ introduces the bool type for Boolean values, with true and false as the only valid values. This enhances code readability and reduces the chance of using non-boolean values where booleans are expected. In contrast, C does not have a native boolean type, typically using 0 for false and non-zero values for true, which can lead to subtle errors.

C++ includes the Standard Template Library (STL), which is a collection of template classes and functions for common data structures and algorithms. The STL provides a powerful toolset for developers to use, reducing the need to implement these data structures and algorithms from scratch.

In C, there is no direct equivalent to the STL. Developers working in C need to create or source their own libraries and data structures.

C++ allows for the declaration of inline functions using the inline keyword. Inline functions are expanded in the code at the call site, potentially reducing the overhead of function calls. This feature enhances performance in some cases and is not part of standard C semantics.

C++ introduces namespaces, which help developers organize their code and avoid naming conflicts. This is particularly useful when working with large codebases.

Availability

The availability of a programming language like C or C++ can be understood in terms of its usage, support, and accessibility across different platforms, systems, and industries.

C is widely used in system programming, including operating system kernels, device drivers, embedded systems, and firmware development. Its minimalistic features, close-to-hardware

capabilities, and ability to interact with memory directly make it suitable for these tasks. C has been used to develop numerous applications, from text editors (e.g., Vim) to databases (e.g., MySQL) and web servers (e.g., Nginx). While it's less common for new application development due to its limited abstraction, existing C-based software continues to be maintained and widely used. C's portability is one of its strengths. Code written in C can be compiled on various platforms with relatively few modifications.

C++ is frequently used in application development, particularly for complex and performance-critical software. It offers high-level abstractions through OOP, templates, and the Standard Template Library (STL). Applications like video games (e.g., Unreal Engine), 3D modeling software (e.g., Blender), and financial applications rely on C++'s features. In industries where speed and efficiency are paramount, such as finance (high-frequency trading), aerospace (flight control systems), and game development (real-time rendering), C++ is a preferred choice due to its control over low-level details and performance optimization. C++ provides a level of organization and code reusability that is highly beneficial in large-scale projects. Its support for classes, inheritance, and polymorphism promotes modular and maintainable code.

Effiency

Efficiency is a critical aspect to consider when choosing between C and C++. Both languages are known for their low-level programming capabilities, but there are nuances in their efficiency due to language features and how they handle certain tasks.

C has minimal high-level abstractions, which means that code written in C is very close to machine code. This can result in highly efficient programs, as you have full control over memory and hardware resources. In C, function calls are generally faster compared to C++ due to the absence of features like virtual functions and the additional bookkeeping required by C++'s method dispatch.

C++ offers a range of abstractions, including classes and objects, which can be optimized by the compiler. For example, in lining and various optimization techniques can be applied to C++ code to make it highly performant. While the STL offers powerful high-level abstractions, its use may

introduce some overhead. However, modern C++ compilers often perform substantial optimization on STL code. Features like dynamic dispatch (virtual functions) can introduce some overhead in C++ programs. However, this overhead is generally minimal and might not be significant in many applications.

C requires manual memory management using functions like malloc and free for dynamic memory allocation. While this provides fine-grained control, it can be error-prone, leading to memory leaks and segmentation faults if not done carefully. C does not support RAII, a programming idiom used in C++ to ensure that resources are properly managed and released. In C, you must explicitly manage resource acquisition and release.

C++ introduces RAII, a powerful idiom that ties the lifecycle of resources to the scope of objects. RAII enables automatic resource management, such as memory allocation and deallocation, file handling, and synchronization, through constructors and destructors of C++ objects. C++ provides smart pointers, including std::unique_ptr and std::shared_ptr, which automate memory management and provide strong safety guarantees. This not only enhances code reliability but also simplifies memory management. The Standard Template Library (STL) in C++ includes container classes like std::vector and std::string, which manage memory efficiently and automatically. This simplifies memory allocation and deallocation for data structures.

C is often regarded as a relatively simple and straightforward language, which contributes to its shorter learning curve. Learning C often emphasizes algorithmic thinking and problem-solving, which are transferable skills to other languages and domains. The simplicity of C can make it a good starting point for beginners in programming, as it allows them to focus on core programming concepts without the added complexity of more advanced language features.

C++ offers more advanced language features and a richer ecosystem, which can result in a steeper learning curve compared to C. C++'s steeper learning curve is mainly attributed to its feature-rich nature and object-oriented paradigm. However, for those who invest the time and effort, C++ offers powerful abstractions and a wide range of applications.

Summary

In conclusion, as I mentioned at introduction because of C++ mostly built on C programming language, C is simpler than C++ but this simplicity at some points is not good for efficiency and cost. So, when a programmer wants to choose between them he/she should look at the advantages and the disadvantages of both of them and decide based on the need of the project and expectations from the language.