Comparative Analysis of C and C++ Programming Languages

C and C++ are two programming languages with basically the same origin. C++ is an extended version of C. While it retains the features of C, it adds more features. C++ can be considered a superset of the C language. C++ shares the same basic syntax and language structure as C, so C code can be directly compiled by C++ compilers. Also, the way both languages are compiled generally falls under the category of 'native code compiler'.

The basic tokens are almost identical in both languages, so the lexical analysis for both languages is also similar. The C++ language contains more custom tokens than C because it adds OOP features. This is necessary for C++ to support object-oriented programming concepts. These special tokens are especially visible when using features such as classes, objects, inheritance and polymorphism.

However, both languages use similar tokens to express basic programming constructs, and the underlying language structures generally remain similar. This ensures that when switching from C to C++, the underlying language constructs remain mostly the same.

Apart from their similarities, the paradigm variants of these two languages are different. The C language basically follows the procedural paradigm, as well as the structural paradigm and the low-level paradigm. However, the C++ language is based on OOP principles, and of course, since it has the same origin as C, it also follows the principles of the procedural paradigm. C++ is also inspired by paradigms such as functional programming, generic programming and modular programming.

In terms of imperative-declarative, both languages are based on imperative principles. In C++, STL and Lambda expressions provide programmers with the ability to write code in a higher level, declarative programming style. STL provides a declarative interface for performing specific tasks, such as data structures and algorithms. Lambda expressions, on the other hand, allow for functional and declarative programming.

As a result, while C and C++ focus mainly on imperative programming paradigms, C++ has a higher level of declarative programming features, giving programmers a more flexible and powerful programming experience. For this reason, C++ is considered a more modern and versatile language.

Syntax and Semantic Comparison:

The general syntax of C and C++ is very similar. Keywords and operators used in C are also used in C++ for the same purpose. However, C++ has more keywords and an extended grammar than C. Some additional operators and keywords in C++ are as follows:

- Scope resolution operator (‘::’)

- Inheritance operator (and access determinant) (:)

- Template syntax (<>)

- References operator (&)

- Comment line (Available in later versions of C (C89)) (//)

- Lambda syntax ([](){})

- >> and << operators

- auto keyword

- new-delete keyword

- true-false keyword

- Type casting operators

- inline keyword

- Scope resolution operator ('::’):

  The existence of this operator is due to OOP and namespaces properties. This operator is used to resolve members of different namespaces with the same name, or to access an external global variable in a scope with the same name as a global variable.

```
#include <iostream>

namespace A

{

   int value = 5;

}

namespace B

{

   int value = 10;

}

int main()

{

   std::cout << A::value << std::endl;

   std::cout << B::value << std::endl;

   return 0;

}
```

```
#include <iostream>

int value = 15;

int main()

{

   int value = 20;

   std::cout << "Local variable : "

               << value << std::endl;

   std::cout << "Global variable: "

               << ::value << std::endl;

   return 0;

}
```

- Class inheritance operator (':'):

Actually, this operator is also used in C for switch statements and '(condition)?(expression):(expression)', but in C++, for OOP reasons, this operator is used after public-private-protected definitions in classes. It is also used for operations called inheritance that allow classes to derive from each other.

- Lamda syntax ('[](){}'):

  Lambda is a short and powerful feature for defining anonymous functions in C++. It is defined with the C++11 standard. It is used as follows:

  ```
  [capture_list] (parameters) -> return_type

  {

          //Body

  }
  ```

- References operator ('&'):

  References are used when passing a variable to a function or returning a value from a function in order to modify the original data. In other words, they have a similar function to pointers in C, but although they look similar, they are operators with a completely different structure. Once defined, references cannot be redirected to another variable. Null cannot be a reference. Reference semantics allows safer code writing and prevents memory errors when using references. In this respect, references make C++ a safer language than C. In summary, references provide a safer and clearer code writing process, but offer less flexibility.

- Boolean keyword:

  Boolean expressions are generally used in all programming languages where conditional evaluation is required. They even represent the 0 and 1 of the binary system, which is the basis of computer science. In C, an expression is false if it is 0 and true in all other cases. From the outside, this can be seen as advantageous, even as a feature that increases orthogonality. However, it is a very bad situation in terms of readability. For example, we are writing a time-based program and we need a variable in the form of a flag that takes the value false if the time exceeds a certain period of time and remains true otherwise. In C and C++, these are defined as follows:

In C Programming Language:

```
int timeFlag = 1; //or anything other than 0

 if (condition)

{

    timeFlag = 0;

}
```

In C++ Programming Language:

```
bool timeFlag = true;

 if (condition)

{

    timeFlag = false;

}
```

As can be seen, it is difficult for a programmer reading the code to understand what the timeFlag variable is in C unless it is specifically mentioned in the comment line (or even if it is mentioned in the comment line, it can be mistaken for any other variable and its value can be changed without realizing it). Because what we want is only to indicate whether an expression is true or false. Since there is no single statement in C that gives the truth attribute, this causes confusion for the reader of the code. In summary, the lack of keywords or similar constructs specific to boolean expressions in C makes C more ambiguous and less secure than C++.

- Type casting operators:

  In C++, there are four different type conversion operators such as 'static_cast', 'dynamic_cast', 'const_cast', 'reinterpret_cast'. These operators are used to perform conversions between different types. In C, type conversion is performed using the (type) conversion operator. In C, type conversion is performed with less control. This can lead to errors, especially if incompatible conversions between data types are performed, or if there are security risks such as memory overflows. As a result, type casting in C and C++ is done with different operators and even different logic. In this respect, C++ has an advantage over C because it is more secure.

- ‘auto’ keyword:

  It was added to C++ with C++11. The 'auto' keyword is used to allow the compiler to automatically assign a type to a variable whose type is not specified. Explicitly specifying the types of variables in C gives more control. This can help the programmer avoid unexpected type conversions, which gives the programmer more control and less confusion, but also less flexibility. In some cases, the same data can be represented by different types, and this flexibility can increase the applicability of needed features. The auto keyword can make C++ code more flexible and readable. Especially when working with complex types, long templates or iterators in modern C++ code, the 'auto' keyword can make the code clearer and at the same time reduce the possibility of making mistakes.

- ‘istream’ and ‘ostream’ operators:

  In C++, 'istream' and 'ostream' are classes that manage input and output operations. These classes are the basis for 'cin' and 'cout' objects. C has functions like 'printf' and 'scanf' for IO operations. C++'s IO system is more secure and flexible than C's IO functions. It is type-safe and reduces the risk of handling incorrect data types.

  IO operations in C++ also have rich features such as formatting and manipulators. For example, manipulators like 'setw', 'setprecision', 'hex' can be used to customize the output. While the IO system in C++ offers a more modern approach, C's IO functions are lower level and more limited. This allows for more control and customization of C++'s IO operations.

- ‘new’ and ‘delete’ operators:

  The 'new' and 'delete' operators are used for dynamic memory management in C++. In C, the functions 'malloc()' and 'free()' are used for this purpose. The 'new' operator can allocate memory for objects of C++ classes and call the constructor function of the class. This ensures that objects are created correctly. The 'new' and 'delete' operators provide a more user-friendly and easy way of memory management in C++. The operators can be used directly to allocate and release memory space. The 'new' operator, when allocating memory for objects, also calls the constructor function of the object. This can lead to unnecessary workload, especially if the memory allocated objects are not necessarily constructed. Some complex scenarios in C++ that require special memory management can be better handled with the 'malloc()' and 'free()' functions in C. These situations involve special cases that require low-level memory management.

- ‘inline’ keyword:

  The 'inline' keyword, when used in C++, requests that the contents of a function be placed directly in place of the function call during the compilation process. This allows the actual code of the function to be used instead of the function call. This can improve performance in small, frequently called functions. The 'inline' keyword is often used in short, frequently called functions. There is no 'inline' keyword in the C language. However, modern C compilers can automatically inline functions to provide performance optimization. This means placing small functions directly in their context instead of calling them, but without the ability to directly instruct the compiler to make this optimization.

  In C, compilers often optimize code using advanced optimization techniques. Optimizations such as function internalization can also be performed in C, but cannot be directly managed by the user.

  As a result, although there is no 'inline' keyword in the C language, modern C compilers often automatically perform internalization to optimize. This can enable performance optimization in C as well, but without the ability to directly instruct the compiler to perform inlining.

- Semantic differences in terms of types:

  The basic data types in C are 'int', 'double', 'float', 'char'. There are also some more complex data types such as array, pointer and struct. C++ uses the same data types as C, but in addition to these, C++ also includes class, object and the boolean types mentioned earlier. To see how the OOP technique contributes semantically to the C++ language, we will discuss the String data type in a little more detail. As a data type, string is a very important data type in programming languages like C and C++. Both C++ and modern C++ standards (C++11 and later) support the string data type, but it is slightly different in both languages. Let's also see how class structures in C++ and special data structures like String can contribute to the semantic flexibility of languages.

  Strings in C are very primitive, essentially arrays of characters, terminated by the null ('\0') character. There are a number of string functions (e.g. strcpy, strlen, strcat) for string manipulation, but it is the programmer's responsibility to provide direct checks against errors and overruns. In C++, however, the string class is a data type defined in the <string> header file. C++ strings are safer and easier to use. C++ strings automatically adjust their size and can be dynamically scaled up or down. The String class has a number of useful functions and operators (for example, concatenation with +, comparison with ==). This makes string manipulation easier and safer in C++. The C++ language supports OOP and allows programmers to create their own data structures. For example, you can create custom string structures such as the 'String' class. Such custom structures increase the semantic flexibility of the language, allowing you to create data types that are more specialized and meet specific needs.

  As a result, string and class-like structures in modern languages like C++ allow you to create more secure and easy-to-use data structures. This makes the language semantically richer and more flexible, because programmers can write higher-level and semantic code to represent more complex data structures.

- Semantic differences in control structures:

  The basic control structures (if-else, switch-case, while, do while and for) are almost identical in both languages. In addition, the 'goto' statement also exists for control structures, but is almost never used in modern coding. C++ also has more advanced control structures (exception handling). This is used to deal with more complex errors.

- Semantic differences in terms of functions:

  Global functions can be written in both languages. However, due to OOP and class concepts, in C++ functions can also be used as methods within classes. In addition, C++ allows functions to be called without using objects or in a state-preserving way thanks to tokens like 'static' and 'const'.

Other constructs that contribute to the semantic breadth of C++ are:

- 'inline' functions
- 'namespace'
- Function overloading
- Operator overloading
- Classes

Contribution of classes to semantic breadth:

- Classes increase semantic breadth. This is because classes can be used to define new data types and more complex data structures. For example, a 'Person' class can contain a person's name, age and other information. This means defining a data type that did not exist before and processing such data in a more meaningful way.

- Classes can contain not only data, but also methods to manipulate that data. This gives more flexibility to define how the data is used and processed.

- OOP is a way to create semantically richer and more meaningful programs because objects have the ability to better reflect real-world objects and more clearly express the relationships of those objects.

As a result, classes are a powerful tool used to increase semantic breadth and define more complex data structures and data types. Especially in OOP, they allow you to create more meaningful and organized programs by increasing semantic richness. This contributes to making programs better understandable and easier to maintain.

Availability Comparison:

In C++, options such as the OOP technique, extended library support, exception handling, overloading, references, namespaces and design patterns (which often help to make software design more modular and flexible) further increase the usability of C++. Thanks to these features, C++ is considered a more powerful language and is often considered a more appropriate choice for larger and more complex software projects.

Efficiency Comparison:

- In terms of memory management:

  C provides manual memory management. Dynamic memory allocation and release is controlled by the programmer using functions such as 'malloc()' and 'free()'.This manual memory management offers the possibility to optimize memory usage more directly, but can be more prone to errors. This is also an aspect that reduces reliability. C++ uses the 'new' and 'delete' keywords for dynamic memory management. It can also provide automatic memory management through smart pointers such as 'std::shared_ptr' and 'std::unique_ptr'. Smart pointers provide a safer option against memory leaks and memory overflows and can reduce errors caused by manual memory management.

- In terms of performance:

  C can run very fast, providing direct access to processor instructions and memory management. C code is often preferred for software that runs directly at the operating system level or hardware level. C++ is derived from C and provides the same low-level control. However, C++'s OOP features and standard libraries can affect performance. For example, the use of standard libraries can lead to performance loss in some cases.

- In terms of language features and library:

  C has minimal language features. This gives programmers direct memory and hardware control, but not higher-level constructs. Its standard library provides basic functions and data structures but is not as rich as C++'s larger standard library.

  C++ has higher-level language features such as OOP. This can make more complex projects more modular and understandable, but it can also affect performance. It has a large standard library (STL). STL contains a set of data structures, algorithms and functions and is generally efficient and fast.

  As a result, C is preferred in applications that require low-level control and performance. It is especially used in areas such as embedded systems, driver software and operating systems. C++ is generally considered a more appropriate choice for more complex projects and large software systems. Higher-level language features and a large set of standard libraries enable faster development processes and more modular code organization.

Learning Curve Comparison:

C is a low-level language and gives the programmer direct memory management and processor-level control. This can present initial difficulties that can lead to bugs and memory leaks. The use of high-level data structures and algorithms is not directly supported, so it is necessary to implement them manually. It is an excellent language for learning basic programming concepts. Understanding algorithms and being able to handle memory management manually can contribute to improving general programming skills.

More complex language features in C++, such as OOP and 'templates', can be a learning challenge at the beginning. Concepts such as class structures, objects and pointers may need to be learned. Monitoring and correcting errors in C++ can be time-consuming, especially when features such as operator overloading are misused. C++ supports modern programming paradigms such as OOP. This makes the code more modular and reusable. The STL (Standard Template Library), a large standard library, provides C++ programmers with a set of data structures, algorithms and data manipulation functions. This can reduce code writing time and complexity.

As a result, the learning curve in C may be suitable for those who want to learn basic programming concepts. C++ may be suitable for those who want to work on more complex projects, but the learning process is more complex.

Orthogonality Comparison:

Orthogonality is the ability to use a feature of a language in any situation and from any angle. Orthogonality also makes a language easier to learn, easier to write and easier to read. For example, Assembly is an extremely orthogonal language. Any instruction works the same in almost every situation. There are very few exceptions in Assembly. But this is not really what we want. When we try to do everything in one way for extreme orthogonality, the efficiency of the language suffers. Let's compare orthogonality for C and C++. The C language is very orthogonal. That is, the basic constructs of C (data types, operators, control structures, etc.) can be used independently and consistently. In C, for example, an operator can work with different data types and a control structure can be used with any data type. Because C is a lower-level language, it can be more prone to errors due to features such as memory management and pointers. Although there is orthogonality, mistakes can be made more easily.

C++ has an orthogonal language design, similar to C, but also includes OOP features. This makes C++ richer and more complex. In C++, OOP features (classes, objects, inheritance, polymorphism, etc.) can be used in harmony with basic C features (pointers, memory management, etc.). In fact, operator overloading increases C++ orthogonality sufficiently. For example, you can use the '+' operator to add any two objects in addition to its basic functions. However, a rich set of libraries, such as C++'s standard library (STL), can further complicate the use of the language. In addition to the higher-level constructs of the language, this library includes features such as algorithms and data structures. This reduces orthogonality.

In conclusion, both C and C++ offer a consistent structure based on the concept of orthogonality, but C++'s OOP features and rich library can increase the complexity of the language and the learning process.

In this article, we have done an in-depth analysis of the similarities and differences between the C and C++ programming languages. We found that while retaining the basic building blocks of the C language, C++ adds object-oriented programming (OOP) and other advanced features. The similar syntax of C and C++ and their ability to switch between their underlying structures show that both languages share the same basic logic. However, C++ has a larger standard library, operator overloading and a richer language structure, making it more suitable for larger and complex software projects. Factors such as memory management, performance, learning curve and semantic richness are important to consider when choosing between C and C++.

C and C++ are powerful languages, each with its own advantages and uses. Choosing the one that best suits the requirements of your projects is vital for successful software development.