Software nativo basado en C++ para aplicaciones de ingeniería. Una alternativa de alto rendimiento en comparación con el *software* gestionado*

Francisco Miguel Cruz Simões de Almeida

Resumen

En este artículo vamos a ver el resurgimiento de la importancia del lenguaje nativo C++ y cómo este ha evolucionado recientemente hacia un lenguaje moderno. Esperamos que esto proporcione el contexto para explicar por qué algunos desarrolladores de software de aplicación (third party developers) están considerando el uso de este lenguaje de programación multiparadigma en detrimento de otros, a menudo clasificados como productivos lenguajes de programación gestionados. Con lenguajes nativos nos referimos a aquellos en los que un compilador genera código de máquina ejecutable, y con lenguajes gestionados a aquellos en los que un compilador genera instrucciones de bytecode, destinadas a ser interpretadas y ejecutadas por una máquina virtual, que en este caso es un requisito para ejecutar bytecode. Java es un ejemplo de un lenguaje gestionado. Este artículo da por hecho que el lector tiene un conocimiento básico o familiaridad con lenguajes basados en Algol, incluyendo C, C++, Java y C#.

Palabras clave: programación, lenguaje nativo, lenguaje gestionado, alto rendimiento, C++, Java.

Native C++ software in engineering as a high-performance alternative to managed software

Abstract

In this article, we will make a case for the resurgence and importance of the C++ native language and an overview of how it has recently evolved into a modern language. We hope that this provides context to why many more developers are considering using this multi-paradigm programming language in detriment of other, often classified as productive, managed programming languages. We refer to native languages as those where a compiler generates native executable machine code, and managed languages as those where a compiler generates byte compiled instructions, intended to be interpreted and executed by a virtual machine, which is in that case a requirement to execute byte compiled code. Java is an example of such a language. This article assumes that the reader has a basic knowledge of, or familiarity with Algol based languages, including C, C++, Java and C#.

Keywords: programming, native language, managed language, high-performance, C++, Java

* Traducción del inglés: Mercedes Vitteri Quiroz.

Introducción

Ya sea que se utilicen para modelado, diseño de equipos, instrumentación, metrología o simulaciones, las computadoras son un activo esencial. El *software* es una parte fundamental de la ingeniería moderna en todas sus formas. Dada su ubiquidad, se considera que el *software* destinado para ingeniería es decisivo y muy importante en la sociedad. La computación eficaz tiene grandes costos, tanto en términos financieros como logísticos, y un *software* rápido y eficiente es una necesidad.

Los primeros años de la década de 1990 presenciaron el surgimiento de Java como una plataforma de *software* de desarrollo rápido, independiente del sistema operativo. Java es un lenguaje orientado a objetos de gran sencillez, con una gran cantidad de bibliotecas compuestas por múltiples funciones que hacen que cualquier aplicación sea muy fácil de implementar.

Dos innovaciones particularmente importantes que ha traído Java han sido la portabilidad de la máquina virtual y la recolección de basura (*garbage collection*), donde toda la memoria es gestionada de forma automática por la misma máquina virtual que ejecuta el código.

Esto, sin embargo, fue solo el inicio. Estas últimas décadas han visto el incremento de la popularidad de otros lenguajes gestionados orientados a objetos construidos sobre los conceptos de Java, como es el C#. La generación resultante de lenguajes gestionados ofreció a los desarrolladores de *software* la interesante disyuntiva de sacrificar algo de rendimiento a cambio de la flexibilidad y de la posibilidad de un desarrollo más fácil. Para muchos desarrolladores de *software*, los incrementos de la productividad para los programadores de *software* se consideraron una gran ventaja, incluso cuando se yuxtaponen a las pérdidas potenciales en la productividad de los usuarios del *software*. Las pérdidas en la productividad para los usuarios son un efecto secundario provocado por el rendimiento inferior del *software*. Actualmente, existen diversos paquetes de software que han adoptado estas tecnologías de desarrollo rápido.

Con el auge de las plataformas móviles y los sistemas embebidos, los lenguajes nativos que generan código de alto rendimiento han regresado a los escenarios. Esto es especialmente cierto para el *software* crítico que se ejecuta en servidores durante veinticuatro horas al día. En los servidores de disponibilidad prolongada, cada ciclo del CPU que se ahorra se traduce, a gran escala, en reducción de costos de energía y reducción de emisiones de carbono.

Los lenguajes nativos más utilizados son C y C++. En particular, C++ se ha vuelto muy relevante a la luz de sus recientes actualizaciones según las especificaciones

estándar. Aquí vamos a debatir sobre cómo algunos de estos cambios hacen que este lenguaje sea mucho menos intimidante de lo que solía ser.

1. *Software* utilizado en ciencias e ingeniería

Con el fin de contextualizar aún más la importancia de la performance del *software* en ciencias e ingeniería, mencionaremos aquí algunos ejemplos notables de *software* en uso.

1.1. SPSS

SPSS, de IBM, es una solución dedicada a la evaluación de datos estadísticos, usado comúnmente para la regresión y el análisis multinivel. SPSS interpreta su propio lenguaje de *scripts* y puede manejar datos importados desde varios formatos externos. Originalmente desarrollado en C, SPSS se convirtió a partir de SPSS 16 en una aplicación Java. Parte de la comunidad de usuarios de inmediato se dio cuenta de una ligera pérdida de velocidad como resultado, a pesar de que la interfaz de usuario se mantiene más o menos igual que antes.

1.2. Cálculos *Ab initio*

Ab initio significa los *primeros principios*. Los cálculos *Ab initio* son una metodología de la física en estado sólido, donde las propiedades de un cristal se pueden computar a través de cálculos pesados basados en principios fundamentales. Al momento de escribir este artículo, no se conoce de la existencia de ningún paquete de *software* comercial que adopte *Ab initio*. Este tipo de *software* se desarrolla en forma independiente en diferentes instituciones, y por lo general se escribe en lenguajes de procedimientos, como FORTRAN o lenguaje C.

1.3. MATLAB

MATLAB es el principal entre los paquetes exitosos de *software* para ingeniería; no solo es un entorno flexible de computación numérica, sino también por sus propios méritos, un lenguaje de programación de cuarta generación. MATLAB tiene una cantidad innumerable de aplicaciones en ingeniería, y en los últimos años ha ganado el prestigio de ser el paquete de computación más potente y robusto de cálculo numérico disponible en el mercado. Sus módulos principales se desarrollaron originalmente en FORTRAN, luego fue reescrito en C, y partes de él fueron luego reescritos en Java. Todavía se desarrolla utilizando esta plataforma. Al

ser su propio lenguaje de programación, MATLAB interpreta *scripts* y puede incluso compilar ejecutables independientes.

1.4. ROOT

ROOT es un marco abierto para el análisis de datos para física nuclear y de partículas. Fue escrito en C++ por los físicos de partículas que trabajan en el CERN y es extremadamente extensible debido a su capacidad para interpretar el código C++. Está en constante desarrollo, gracias en gran parte a las necesidades de la comunidad científica.

2. Modernas mejoras en el lenguaje C++

Al igual que las especificaciones estándar más antiguas de C++, el nuevo estándar corresponde a un lenguaje de programación multiparadigma. Sin embargo, se diferencia porque es menos propenso a errores e incorpora nuevos paradigmas, tales como la programación funcional. También refina su modelo de memoria, lo que mejora el código concurrente y cuenta con otras mejoras, entre ellas la gestión de memoria de montículo (*heap memory)*, con nuevas clases para los punteros inteligentes.

Una preocupación común, que es importante para los desarrolladores de *software*, es cuánto se puede hacer utilizando solo el núcleo del lenguaje y su biblioteca estándar.

2.1. Sintaxis simplificada

Entre las más simples de las nuevas características de C++ están aquellas con los mayores beneficios potenciales en el estilo de programación. Características tales como la palabra clave reservada 'auto', que proporcionan un poco de flexibilidad a un lenguaje que es demasiado rígido, debido a las estrictas definiciones de tipos estáticos.

En el siguiente ejemplo se muestra la palabra clave que se utiliza para declarar un puntero compartido de tipo *UserClass*:

```
class    User Class  { };
auto    p = std : : make_shared<UserClass >( ) ;
```

Otra característica de gran utilidad del ISO estándar C++ de 2011 (comúnmente referido como C++11) —que hace el código más fácil de escribir y leer— es la nueva sintaxis del bucle *for* basado en intervalo. Similar a otros lenguajes que tenían esta característica anteriormente, los bucles *for* basados en intervalo pueden ahora desreferenciar a los *iterators* en un contenedor estándar o *arrays* del estilo del C y permitir una semántica más simple en el código. Saltando la declaración detallada de los tipos de nombres, uno puede convertir este código de bucle.

```
Using namespace   std ;

vector<double>  v ( 5,  . 0 ) ;

for ( std : : vector<double >: : iterator   i  =  v . begin ( ) ;
                i  != v . end ( ) ;
              ++i )
{
     cout << *i << ' \n ' ;
}
```

A esta forma mucho más simple:

```
using namespace   std ;

vector<double>  v ( 5,  . 0 ) ;

for   ( auto element  :  v )
{
    cout  <<  element  <<  ' \n '
}
```

La palabra clave reservada 'auto' es, una vez más, útil para hacer el bucle *for* basado en rango fácil de escribir y comprender para cualquiera que lea el código.

2.2. Funciones sin nombre (*lambda*)

El concepto de las funciones *lambda* se han presentado como del C++11. Las *lambdas* son utilizables en cualquier circunstancia en la que un *"callable"* sea utilizable.

Esto significa que las *lambdas* pueden pasar como objetos *functor* o punteros de función.; ellas son, en realidad, el azúcar sintáctico que permite declarar un objeto *functor* en línea. En la nueva norma ISO de 2014 (llamado por muchos el lenguaje C++ 14), se han ampliado las *lambdas* para permitir la programación genérica por su cuenta. El ejemplo que se muestra consiste en una *lambda* genérica. Las *lambdas* genéricas no solo se expanden en objetos lógicos, el compilador mira y deduce el tipo de cada parámetro declarado como auto. Téngase en cuenta también cómo una *lambda* puede ser declarada como un objeto, y luego llamada más tarde como una función ordinaria.

```
auto add = [ ] ( auto x , auto y ) { return x + y ; };

double v = 0;
double w = 2;

double z = add (w, v ) ;
```

Las funciones *lambda* son de particular interés cuando se usan en conjunto con algoritmos de la biblioteca estándar de contenedores. En este ejemplo, al ordenar un vector de estructuras de datos que no tienen un orden natural, uno puede también definir la relación de orden fuera de la definición de estructura a través de un objeto *functor*. Las *lambdas* sirven para este propósito a la perfección, permitiendo al programador definir la función predicado de ordenamiento en línea, dentro de la llamada a la función estándar *sort*.

```
s t r u c t ValuesPair
{
    double a ;
    double b ;
 };

vector<ValuesPair> pairs ( 5 ) ;

std : : sort ( pairs . begin ( ) , pairs . end ( ) ,
[ ] ( ValuesPair a , ValuesPair b)
{
     return a . a + a . b < b . a + b . b ;
} ) ;
```

A los corchetes cuadrados en la declaración *lambda* se les conoce como la lista de captura. Con la lista de captura, el programador puede pasar los valores y referencias para utilizarlos en la definición del cuerpo *lambda*, sin tener que pasarlos como parámetros de la función. Como el puntero de este objeto también puede ser capturado por *lambda*, es especialmente potente en programación orientada a objetos, donde las devoluciones de llamada son lugar común. La capacidad de utilizar *lambdas* para definir las devoluciones de llamada y *functors* en línea que son sensibles al contexto va a contribuir en gran medida al desarrollo rápido de aplicaciones en C++.

2.3. Concurrencia

Aunque no es una nueva característica del lenguaje principal, sino más bien una extensión de la biblioteca estándar, los programadores ya no tendrán que depender de las bibliotecas de terceros, y pueden ahora, de hecho, escribir código portable, multi-hilo y asincrónico. En el siguiente ejemplo, un *flag* se declara *atomic*, permitiéndole estar libre de bloqueo y accesible desde múltiples hilos, mientras que la nueva clase *thread* de la biblioteca estándar se utiliza para ejecutar una simple tarea de contar en un hilo separado. Téngase en cuenta lo fácil que es proporcionar el objeto hilo con una función *lambda* que consiste en la función de hilo. Entonces, el hilo principal espera que el hilo de trabajo concluya su ejecución.

```
using namespace std ;

static   atomic<bool>  running  =  false ;

thread  job ([&running ] ( )
{
    unsigned long counter = 0 ;
    while ( running )
{
        If  (++counter  >  30000)
     {
            running = false ;
     }
   };
}) ;

job.join ( ) ;
```

Muchas más clases de ayuda a la concurrencia y modismos están disponibles en la biblioteca estándar de C++, tales como tareas asincrónicas, *futures* (que a su vez permiten el manejo de excepciones dilatorias) y muchos tipos de *mutexes*.

2.4. Plantillas *variadic*

Otra interesante (aunque más sofisticada) extensión para el lenguaje C++ es la introducción de plantillas *variadic*. Esta nueva sintaxis de plantilla permite a las bibliotecas implementar verdaderas funciones *variadic*. El siguiente ejemplo consiste en funciones de suma y promedio aceptando completamente cualquier cantidad de términos, de cualquier tipo básico.

```
template<typename  T>
double Sum (T param)
{
    return   static_cast <double>(param ) ;
}

template<typename T, typename . . . Args>
double  Sum (T param1 , Args . . . params )
{
    return   static_cast <double>(param1 )  + Sum( params . . . ) ;
}

template<typename . . . Args>
double   Average (Args . . . params )
{
    return  Sum ( params . . . ) / sizeof . . . ( params ) ;
}
```

Las funciones *variadic* —implementadas mediante la utilización de plantillas *variadic*— generalmente son declaradas en forma recurrente como plantillas de dos funciones: una por medio del uso de la elipsis y otra con unos pocos parámetros (o en este caso, un argumento) y sin elipsis. El compilador instancia la sobrecarga basada en elipsis, generando una llamada recurrente a sí misma mientras existan más argumentos. Cuando existe un solo argumento, se llama a la sobrecarga con un argumento y el compilador deja de generar más instancias.

Naturalmente, esta flexibilidad extendida se produce a costo del código inflado e incluso del rendimiento, especialmente si es utilizado en exceso. Las llamadas funciones recurrentes son mucho más costosas computacionalmente que las llamadas de función única, implementadas con una cantidad máxima fija de argumentos. Independientemente, las plantillas *variadic* son una herramienta muy poderosa para los escritores de bibliotecas genéricas.

3. Conclusión

Independientemente del mercadeo exitoso de los lenguajes gestionados, la compatibilidad entre C y C++ es un gran activo. Hay una enorme cantidad de bibliotecas *legacy* escritas en C, que no serán reemplazadas o reescritas en el corto plazo. Los compiladores de C++ también trabajan como compiladores de C, y vincular el código C++ a las bibliotecas estáticas C es una tarea trivial.

La compatibilidad con versiones anteriores a C, junto con la familiaridad y la amplia disponibilidad del código *legacy*, hacen de la elección de C++ el lenguaje de programación más pragmático en muchas situaciones críticas.

Una lista de compañías famosas o reconocidas que han utilizado o que utilizan C++ para sus proyectos, está a disposición del público por el creador del lenguaje, Bjarne Stroustrup. Esta lista incluye compañías como Adobe Systems, Apple, AT & T, Autodesk, Facebook, Google, IBM, Intel y, por supuesto, Microsoft.

Con las últimas iteraciones del lenguaje estándar C++, el interés en el desarrollo de *software* basado en C++ ha ido en aumento.

Referencias

C++ Standards Committee (s. f.). *Official Webpage of the ISO C++ Standard*. Recuperado de http://www.isocpp.org

CERN Staff (s. f.). ROOT *Data Analysis Framework*. Open source. Recuperado de http://root.cern.ch/drupal/

Czarnecki, K., & Eisenecker, U. W. (2002). *Generative programming : methods, tools, and applications*. 2nd printing. New York, USA: Addison Wesley.

IBM (s. f.). *IBM SPSS*. Proprietary. Recuperado de http://www 01.ibm.com/software/analytics/spss/

Josuttis, N. M. (2012). *The C++ standard library: a tutorial and reference*. 2nd edition. New Jersey, USA: Addison Wesley Longman.

Lewis, J., & Loftus, W. (2005). Java software solutions, foundations of program design. International edition. 4th edition. Boston, USA: Pearson/Addison Wesley.

Mathworks (s. f.). *MATLAB and Simulink for Technical Computing*. Proprietary. Recuperado de http://www.mathworks.co.uk/index.html?s tid=gn logo

Press, W. H., Teukolsky, S. A., Vetterling, W.T. & Flannery, B. P (2002). *Numerical Recipes in C++*. 2nd edition. New York, USA: Cambridge University Press.

Stroustrup, B. (s. f.). *C++. Applications*. Recuperado de http://www.stroustrup.com/applications.html

Native C++ software in engineering as a high-performance alternative to managed software

Francisco Miguel Cruz Simões de Almeida

Introduction

Whether when they are used for modelling, designing equipment, instrumentation, metrology, or simulations, computers are an essential asset. Software is a fundamental part of modern day engineering in all of their forms. Given its ubiquity, software intended for engineering is considered critical and very important in society. High performance computing has great costs, both in financial and logistical terms, and fast and efficient software is a must.

The early 1990s saw an emergence of Java as a rapid development software platform, which is independent of the operating system. Java is a very straightforward object oriented language with a plethora of feature-rich libraries that make any application very easy to implement.

Two particularly important innovations Java brought were the portability of the virtual machine and garbage collection, where all memory is automatically managed by the same virtual machine that executes the code.

This, however, was only the beginning. These past decades saw the rise in popularity of other managed object oriented languages building on the concepts of Java such as C#. The ensuing generation of managed languages presented software developers with the interesting tradeoff of sacrificing some performance for flexibility and easier development. For many software developers, the gains in productivity for software development were considered a great advantage, even when juxtaposed to potential losses in productivity of the software users. The losses in productivity for software users are a side-effect of the inferior software performance. We mention examples of software packages that adopted these rapid development technologies.

With the rise of mobile platforms and embedded systems, native languages that generate high-performance code have returned to the spotlight. This is especially true for critical software which runs in servers for twenty four hours a day. In servers of prolonged up-time, every CPU cycle that is avoided translates, in larges scales, to energy costs reductions and reduced carbon emissions.

The most widely used native languages are C and C++. In particular, C++ has become very relevant in light of its recent updates to the standard specification. Here, we will discuss how some of these changes make this language far less daunting than what it used be.

1. Software used in sciences and engineering

In order to further contextualize the importance of performant software to sciences and engineering, we will mention here a few examples of prominently used software.

1.1. SPSS

SPSS, from IBM, is a dedicated solution for statistical data evaluation, commonly used for regression and multilevel analysis. SPSS interprets its own scripting language and can handle data imported from various external formats. Originally developed in C, SPSS became as of SPSS 16 a Java application. Part of the user community immediately took notice of a slight loss in speed as a result, despite the user interface remaining more or less the same as before.

1.2. Ab initio calculations

Ab initio stands for first principles. Ab initio calculations are a methodology in solid state physics where properties of a crystal can be computed through heavy calculations based on fundamental principles. When this article was written, no known ab initio commercial software package existed. This kind of software is developed independently in different institutions, and usually written in procedural languages such as FORTRAN or C.

1.3. MATLAB

Chief among commonly cited successful engineering software packages is MATLAB. Not only is MATLAB a flexible numerical computing environment, but also on its own merits, a fourth-generation programming language. MATLAB has an innumerable amount of applications in engineering, and has over the years earned its reputation of being the most powerful and robust numeric computation package available in the market. Its core modules were originally developed in FORTRAN, then re-written in C, and parts of it were later re-written in Java. It still is developed using this platform to this day. Being its own programming language, MATLAB interprets scripts and can even compile standalone executables.

1.4. ROOT

ROOT is an open-source data analysis framework for nuclear and particle physics. It was written in C++ by particle physicists working at CERN and is extremely extensible due to its capability to interpret C++ code. It is in constant active development, thanks in no small part to the necessities of the scientific community.

2. Modern C++ language improvements

Like older standard specifications of C++, the new standard is a multi- paradigm programming language. Unlike them, however, it is less error prone and features new paradigms such as functional programming. It also refines its memory model, which improves concurrent code and features other improvements such as heap memory management, with new classes for smart pointers.

A common concern, which is important for software developers, is how much can already be done using just the core language and its standard library.

2.1. Simplified syntax

The simplest of the new features of C++ are among those with the most potential benefits on programming style. Features such as the auto keyword, which provide a bit of flexibility to a language which is too rigid, due to strict static type definitions. In the next example shown, the auto keyword is used to declare a shared pointer of type *UserClass*.

```
class   User Class  { };
auto   p = std : : make_shared<UserClass >( ) ;
```

Another very useful C++11 feature that makes code easier to write and read is the new range-based *for-loop* syntax. Similar to other languages who previously has this feature, *for-loops* can now dereference iterators in a standard container or C style arrays and allow simpler semantics in the code.

By skipping the declaration of known but verbosely named types, one can turn this loop code:

```
Using namespace   std ;

vector<double>  v ( 5,  . 0 ) ;

for ( std : : vector<double >: : iterator   i  =  v . begin ( ) ;
                i  != v . end ( ) ;
              ++i )
{
     cout << *i << ' \n ' ;
}
```

Into this, far simpler, form:

```
using namespace   std ;

vector<double>  v ( 5,  . 0 ) ;

for   ( auto element  :  v )
{
    cout  <<  element  <<  ' \n '
}
```

The auto keyword is, once again, useful for making the range based for loop easy to write and understand to anyone reading the code.

2.2. Unnamed (lambda) functions

The concept of lambda functions has been introduced as of the ISO C++ language standard of 2011 (commonly referred to as C++11). Lambdas are usable in any circumstance where a "callable" would be usable. This means that lambdas can be passed as functor objects or function pointers. They are, in reality, syntactical sugar that allows one to declare a functor inline. In the new ISO standard of 2014 (called by many the C++14 idiom), lambdas have been extended to support generic programming on their own. The shown example consists of a generic lambda. Generic lambdas not only expand into functor objects, the compiler looks and deduces the type of each parameter declared as auto. Also note how a lambda can be declared as an object, and then called later on as an ordinary function.

```
auto add = [ ] ( auto x , auto y ) { return x + y ; };

double v = 0;
double w = 2;

double z = add (w, v ) ;
```

Lambda functions are of particular interest when used in tandem with standard library algorithms for containers. In this example, when sorting a vector of data structures which do not have a natural order, one can also define the order relation outside the struct definition via a functor. Lambdas serve this purpose perfectly, by allowing the programmer to define the predicate sorting function inline, within the standard sort function call.

```
s t r u c t ValuesPair
{
    double a ;
    double b ;
 };

vector<ValuesPair> pairs ( 5 ) ;

std : : sort ( pairs . begin ( ) , pairs . end ( ) ,
[ ] ( ValuesPair a , ValuesPair b)
{
     return a . a + a . b < b . a + b . b ;
} ) ;
```

The square brackets in the lambda declaration are referred to as the capture list. With the capture list, the programmer is able to pass values and references to use in the lambda body definition, without having to pass them as function parameters. A this object pointer can also be captured by the lambda, making it especially powerful in object oriented programming where callbacks are common place. The ability to use lambdas to define callbacks and functors inline that are context aware is going to contribute greatly to rapid application development in C++.

2.3. Concurrency

While not a new core language feature, but rather an extension of the standard library, programmers will no longer need to rely on third party libraries, and can now, in fact, write portable multi-threaded and asynchronous code. In the following example, a flag is declared atomic, allowing it to be lock-free accessible from multiple threads, while the new standard library thread class is being used to run a simple counter task in a separate thread. Note how simple it is to provide the thread object with a lambda function which consists of the thread function. The main thread then waits for the job thread to conclude its execution.

```
using namespace std ;

static   atomic<bool>  running  =  false ;

thread  job ([&running ] ( )
{
    unsigned long counter = 0 ;
    while ( running )
{
        If  (++counter  >  30000)
     {
            running = false ;
     }
  };
}) ;

job.join ( ) ;
```

Many more concurrency helper classes and idioms are available in the C++ standard library, such as asynchronous tasks, futures (which in turn allow delaying exception handling) and several types of mutexes.

2.4. Plantillas variadic

Another interesting (albeit more sophisticated) extension to the C++ language is the introduction of variadic templates. This new template syntax allows libraries to implement true variadic functions. The following example consists of fully generic sum and average functions accepting any number of terms, of any basic type.

```
template<typename  T>
double Sum (T param)
{
    return   static_cast <double>(param ) ;
}

template<typename T, typename . . . Args>
double  Sum (T param1 , Args . . . params )
{
    return   static_cast <double>(param1 )  + Sum( params . . . ) ;
}

template<typename . . . Args>
double   Average (Args . . . params )
{
    return  Sum ( params . . . ) / sizeof . . . ( params ) ;
}
```

Variadic functions implemented using variadic templates usually are recursively declared as two function templates: one using the ellipsis and another with a few parameters (or in this case, one argument) and no ellipsis. The ellipsis based overload is instantiated by the compiler, generating a recursive call to itself for as long as more arguments exist. When only one argument exists, the overload with one argument gets called and the compiler stops generating more instantiations.

Naturally, this extended flexibility comes at the cost of code bloat and even performance, especially if overused. Recursive function calls are much more computationally expensive than single function calls implemented with a fixed maximum amount of arguments. Regardless, variadic templates are a very powerful tool for generic library writers.

3. Conclusion

Irrespective of the successful marketing of managed languages, compatibility between C and C++ is a big asset. There is a vast amount of legacy libraries written in C, which will not be replaced or re-written any time soon. C++ compilers also work as C compilers, and linking C++ code to static C libraries is a trivial task.

The backwards compatibility to C, coupled with the familiarity and extensive availability of legacy code, make C++ the most pragmatic programming language choice in many critical situations.

A list of famous and/or celebrated companies that have used, or are using C++ for their projects, is made publicly available by the creator of the language, Bjarne Stroustrup. This list features companies such as Adobe Systems, Apple, AT&T, Autodesk, Facebook, Google, IBM, Intel and of course, Microsoft.

With the latest iterations of the C++ language standard, the interest in software development based on C++ has been on the rise.

References

C++ Standards Committee (s. f.). *Official Webpage of the ISO C++ Standard*. Retrieved from http://www.isocpp.org

CERN Staff (s. f.). ROOT *Data Analysis Framework*. Open source. Retrieved from http://root.cern.ch/drupal/

Czarnecki, K., & Eisenecker, U. W. (2002). *Generative programming : methods, tools, and applications*. 2nd printing. New York, USA: Addison Wesley.

IBM (s. f.). *IBM SPSS*. Proprietary. Recuperado de http://www 01.ibm.com/software/analytics/spss/

Josuttis, N. M. (2012). *The C++ standard library: a tutorial and reference*. 2nd edition. New Jersey, USA: Addison Wesley Longman.

Lewis, J., & Loftus, W. (2005). Java software solutions, foundations of program design. International edition. 4th edition. Boston, USA: Pearson/Addison Wesley.

Mathworks (s. f.). *MATLAB and Simulink for Technical Computing*. Proprietary. Retrieved from http://www.mathworks.co.uk/index.html?s tid=gn logo

Press, W. H., Teukolsky, S. A., Vetterling, W.T. & Flannery, B. P (2002). *Numerical Recipes in C++*. 2nd edition. New York, USA: Cambridge University Press.

Stroustrup, B. (s. f.). *C++. Applications*. Retrieved from http://www.stroustrup.com/applications.html