Programación genérica en C++, usando Metaprogramación

Recepción: Febrero de 2007 / Aceptación: Mayo de 2007

(1) Eduardo Raffo Lecca

RESUMEN

La complejidad de nuestros programas, va en aumento con los problemas a los que nos enfrentamos. Una forma de darle solución, es repetir una y otra vez las mismas estructuras, tal como nos acostumbró la programación tradicional o imperativa. Este artículo, explora las relaciones entre plantillas C++, la programación genérica y la habilidad de desarrollar computaciones estáticas y generación de código. Muchos desarrolladores usan lenguajes de alto nivel para desarrollar aplicaciones en computación científica; y el mecanismo de las plantillas en C++ les permite resolver importantes problemas en el diseño de librerías de clase.

Palabras Clave: Programación Orientada a Objetos, plantillas en C++, programación genérica o funcional, metaprogramación, evaluación parcial.

GENERIC PROGRAMMING IN C++, USING METAPROGRAMMING

ABSTRACT

The complexity in our programs increases along with the problems we deal with. A way of solving them is to repeat more than once the same structures, just like traditional or imperative programming accustomed us to. This article explores the relationships among C++ templates, generic programming and the ability to develop statistics computations and generation of codes. Several developers use languages of high level to develop applications in scientific computation; and the mechanism of templates in C++ allows them to solve important problems in the design of libraries of class.

Key Word: Object Oriented Programming, C++ templates, generic or functional programming, metaprogramming, partial evaluation.

INTRODUCCIÓN

La Programación Orientada a Objetos (OOP), significa una gran actualización a la caja de herramientas de programación. Para aplicaciones complejas, es una oxigenación en lo concerniente al desarrollo y mantenimiento, debido al límite de lo factible con las técnicas de programación tradicional.

Las técnicas de plantillas en C++, permiten un grado de programación genérica, creando códigos especiales para problemas especializados[1].

Algunos teóricos siguen centrando su atención en los algoritmos. Algo que estaba ahí también desde el principio. Se dieron cuenta que frecuentemente las manipulaciones contienen un denominador común que se repite bajo apariencias diversas. Por ejemplo, la idea del valor mínimo o máximo, se repite infinidad de veces en la programación, aunque los objetos a evaluar varíen de un caso a otro por el tipo de datos. Sobre esta idea surgió un nuevo paradigma denominado programación genérica o funcional.

La programación genérica está mucho más centrada en los algoritmos que en los datos y su postulado fundamental puede sintetizarse en una palabra: generalización. Significa que, en la medida de lo posible, los algoritmos deben ser parametrizados al máximo y expresados de la forma más independiente posible de detalles concretos, permitiendo así que puedan servir para la mayor variedad posible de tipos y estructuras de datos.

La parametrización en los algoritmos, supone un aporte a las técnicas de programación, al menos tan importante, como ha sido la introducción del concepto de herencia; y que permite resolver nuevos problemas.

A menudo se tiene el pensamiento que un programa es la implementación de un mapeo (*mapping*).

El programa toma valores y mapea entonces a su salida. En la programación imperativa este *mapping* es realizado en forma indirecta, por los comandos que leen valores de entrada, manipulan estos y luego escriben las salidas. Los comandos influyen en los programas mediante el uso de las variables almacenadas en la memoria[2].

En la programación funcional, el *mapping* de valores de entrada a valores de salida es realizado de modo directo. El programa es una función o grupo de funciones; sus relaciones son muy simples: entre

(1) Ingeniero Industrial. Profesor del Departamento de Ingeniería de Sistemas e Informática, UNMSM.
E-mail: eraffol@unmsm.edu.pe

ellas se invocan. Los programas son escritos en un lenguaje de expresiones, funciones y declaraciones.

La programación funcional está caracterizada por el uso intensivo de expresiones y funciones. Este paradigma está basado en las funciones matemáticas y corresponde a uno de los más importantes estilos de lenguajes no imperativos[3]. Aquí se encuentran los lenguajes: LISP que es un lenguaje funcional puro, COMMOM LISP una amalgama de varios tempranos dialectos LISP en los años 80, ML un lenguaje funcional fuertemente tipeado con mas sintaxis convencionales que LISP, Miranda un lenguaje funcional puro parcialmente basado en ML y el pequeño Scheme entre otros.

Una función matemática es un *mapping* de miembros de un conjunto, llamado el conjunto dominio a otro llamado el conjunto rango. La definición de una función relaciona los conjuntos: dominio y rango. El *mapping* está descrito por una expresión.

Una de las características de las funciones matemáticas es que el orden de la evaluación del *mapping* de las expresiones es controlada por la recursión y las expresiones condicionales.

Tempranos trabajos sobre funciones, trajo consigo la notación *Lambda*, de Alonzo Church (1941), para proveer un método para definir funciones. Una expresión *lambda* especifica el parámetro y el mapping de la función. Considere:

$$\lambda(x)x * x * x$$

Cuando una expresión *lambda* es evaluada para un cierto parámetro, se dice que se está aplicando al parámetro. Una aplicación de la expresión *lambda* es denotada como:

$$\lambda(x)x * x * x(2)$$

resulta en el valor de 8. Las expresiones *lambda* pueden tener más de un parámetro.

Una función de alto orden o forma funcional, permite definir una función compleja; tal como ocurre con la composición de funciones, así:

$$h \equiv f \circ g$$

Aplicado al ejemplo:

$$f(x) \equiv x + 5$$
$$g(x) \equiv x * 2$$

Resulta:

$$h(x) \equiv f(g(x)) \equiv (x * 2) + 5$$

El equivalente funcional de un bucle o *loop*, es la recursividad. En un código para ML, el cómputo del factorial viene dado por:

```
fun factorialcall (n , p ) =
        if  n > 0
         then factorialcall ( n-1, p*n )
        Else p
```

Siendo el valor inicial para la llamada a la recursividad, como:

```
fun  factorial ( n ) = factorialcall ( n, 1 )
```

PLANTILLAS

El código es similar siempre, pero estamos obligados a rescribir ciertas funciones que dependen del tipo o de la clase del objeto que se almacena. Tómese por ejemplo el problema de encontrar el valor mayor de un par de datos; tanto para un entero, flotante, carácter, etc.

Desde el ANSI C, se hacen uso de las funciones prototipos; en nuestro caso para encontrar el mayor de dos valores para distintos datos, son:

```
int max(int,int);
double max(double,double);
char max(char,char);
```

El programa de la figura 1, hace uso de esta sobrecarga de funciones, para obtener el mayor entre dos enteros, doble flotantes y carácter. Ver figura 2.

Figura 1. Programa patron0.cpp

```
#include <iostream.h>
#include <stdlib.h>
// sobrecarga de funciones para max()
int max(int,int);
double max(double,double);
char max(char,char);
// patron0.cpp

void main()         // Eduardo Raffo Lecca
{
   int a=5,b=3;
   cout << " El mayor de los enteros es : " << max(a,b)<< endl;
   double c=5.5,d=10;
   cout << " El mayor de los flotantes es: " << max(c,d)<< endl;
   char e='a',f='A';
   cout << " El mayor de los char es    : " << max(e,f)<< endl;
   system("PAUSE");
}
int max(int a,int b)
{
     return a>b?a:b;
}
double max(double a,double b)
{
     return a>b?a:b;
}
char max(char a,char b)
{
     return a>b?a:b;
}
```

Figura 2. Resultado de ejecutar patron0.cpp

Figura 3. Programa patron01.cpp

```
#include <iostream.h>
#include <stdlib.h>
// plantilla para max()
template<class T>
T max(T a,T b)
{
     return a>b?a:b;
}
// patron01.cpp

void main()   // Eduardo Raffo Lecca
{
      int a=5,b=3;
      cout << " El mayor de los enteros es : " << max(a,b)<< endl;
      double c=5.5,d=10;
      cout << " El mayor de los flotantes es: " << max(c,d)<< endl;
      char e='a',f='A';
      cout << " El mayor de los char es : " << max(e,f)<< endl;
      system("PAUSE");
```

Las sobrecargas, para **int**, **double** y **char** son:

```
int max(int a,int b)
{
     return a>b?a:b;
}
double max(double a,double b)
{
     return a>b?a:b;
}
char max(char a,char b)
{
     return a>b?a:b;
}
```

Un patrón o plantilla es una forma de objeto, que se aplica a diferentes instancias, sin especificar el tipo de objeto a ser referenciado. El patrón con un simple código cubre un gran rango de funciones de sobrecarga denominadas funciones patrones o un gran rango de clases denominadas clases patrones [4].

Se dice que los patrones son como una factoría de ensamblaje, porque producen una variedad de objetos; es decir, dado un material se produce un determinado artículo.

Las plantillas (*templates*) permiten parametrizar estas clases para adaptarlas a cualquier tipo. Así el cambio de las tres sobrecargas, viene reemplazado por la plantilla o patrón siguiente:

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

Esta función patrón, trabaja para enteros, como también para flotantes y caracteres. Afortunadamente existen las plantillas que hacen la labor de programación más fácil.

Las plantillas, también denominadas tipos parametrizados, son un mecanismo de C++ que permite que un tipo pueda ser utilizado como parámetro en la definición de una clase o una función.

C++ permite crear plantillas de funciones y de clases. La sintaxis para declarar una plantilla de función es parecida a la de cualquier otra función, pero se añade al principio una presentación de la clase que se usará como referencia en la plantilla:

```
template<class | typename<id>[...]>
<tipo de retorno> <identificador>(<lista de parámetros>)
{
      // cuerpo de la función
}
```

La lista de clases que se incluye a continuación de la palabra reservada *template* se escribe entre las llaves "<" y ">"; y en este caso esos símbolos no indican que se debe introducir un literal.

El programa con la introducción de la plantilla se muestra en la figura 3.

Se ha considerado que la plantilla, radique en un archivo tipo *header*, donde se ha incluido la función plantilla min(), tal como se presenta en las figuras 4 y 5.

Una plantilla que ejecuta el valor máximo para tres valores, se muestra en el código de la figura 6, y su ejecución en la figura 7.

La idea central a resaltar aquí es que una plantilla genera la definición de una clase o de una función mediante uno o varios parámetros. A esta instancia concreta de la clase o función generada, se la denomina especialización o especialidad de la plantilla.

Consideremos otro caso, los arreglos. Una propiedad básica de los arreglos, es que contienen un número fijo de objetos. Por tanto se debe conocer por lo mínimo:

- ¿Qué clases de objetos contiene?
- ¿Cuántos objetos contiene?

Una primera solución, es hacer uso del constructor en la forma que se presenta en la figura 8:

Figura 4. Programa patron01.cpp

```
// minymax.h
#ifndef __MINYMAX_H
#define __MINYMAX_H
// Eduardo Raffo Lecca
template <class T>
T max(T a,T b)
{
return (a>b)?a:b;
}
template <class T>
T min(T a,T b)
{
return (a<b)?a:b;
}
#endif

#include <iostream.h>
#include <stdlib.h>
#include "minymax.h"
// patron1.cpp
void main() // Eduardo Raffo Lecca
{
int x1=4,x2=3;
double y1=7.5,y2=8.3;
char c1='a',c2='A';
cout << "min int    : " << min(x1,x2) << endl;
cout << "max double : " << max(y1,y2) << endl;
cout << "max char   : " << max(c1,c2) << endl;
    system("PAUSE");
}
```

Figura 5. Ejecución del programa patron01.cpp

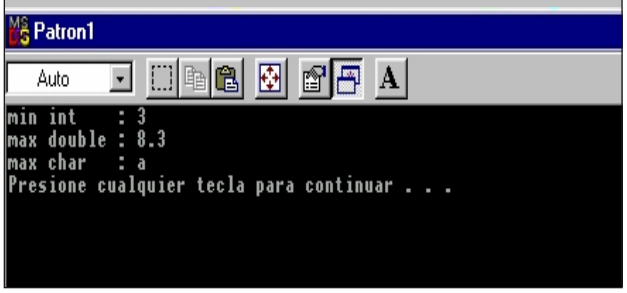
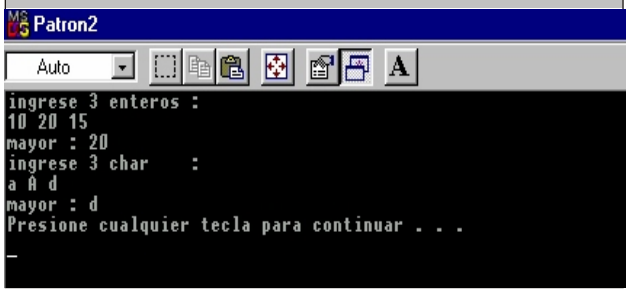


Figura 6: Programa patron2.cpp

```
#include <iostream.h>
#include <stdlib.h>

template <class T>
T maximo(T a,T b,T c)
{

T max=(a>b?a:b);
return(max>c?max:c);
}

// patron2
main()  // Eduardo Raffo Lecca
{

int a,b,c;
cout << "ingrese 3 enteros : " << endl;
cin >> a >> b >> c;
int j=maximo(a,b,c);
cout << "mayor : " << j << endl;
char d,e,f;
cout << "ingrese 3 char    : " << endl;
cin >> d >> e >> f;
char k=maximo(d,e,f);
cout << "mayor : " << k << endl;

      system("PAUSE");
}
```

Figura 7: Ejecución del programa patron2.cpp



La clase *Array*, asigna un nuevo arreglo de n Ts que apunta a *list*; haciendo uso de la asignación dinámica.

En cualquier código que llama al operador **new**, se debe preguntar que pasa cuando falla la asignación, tal como se muestra en la figura 9.

Este segmento de código, es un análisis de cómo responde *list* ante una falla de **new**, comúnmente por disponibilidad de memoria.

El conocer el tamaño de un arreglo antes del tiempo de ejecución (*run time*), implica el conocimiento de la longitud del arreglo en tiempo de compilación (*compiler time*).

Las plantillas C++, permiten incluir información en tiempos de compilación. No sólo las plantillas permiten parámetros tipos(*type parameters*), también ofrecen el *non-type parameters*; siendo enteros, enumerados, punteros / referencias a objetos/ funciones y punteros a miembros; como se describen en la figura 10, que aparece a continuación:

La sobrecarga del **operador[ ]**, permite el manejo del *rvalue* y el *lvalue*. Para un mejor manejo del índice del arreglo, un tipo **unsigned int** es reemplazado por **size_t**, el cual es el tipo **unsigned** resultante del operador **sizeof** (ver figura 11).

Las plantillas representan un método muy eficaz de generar código (definiciones de funciones y clases), a partir de definiciones relativamente pequeñas. Su utilización permite técnicas de programación avanzada, en las que implementaciones muy sofisticadas se muestran mediante interfaces que

Figura 8. Programa patron51.cpp

```
template <class T>
class Array {
T* lista;
public :
Array(size_t const n):lista(new T[n]){}
~Array()
{delete[] lista; }
T get(int i)const;
void set(T a,int i);
};
template <class T>
T Array<T>::get(int i)const
{
return (T)lista[i];
}

template <class T>
void Array<T>::set(T t,int i)
{
lista[i]=t;
}
// patron51.cpp#include <iostream.h>
#include <stdlib.h>
```

Figura 9. Programa patron52.cpp

```
template <class T>
class Array {
T* lista;
public :
Array(size_t const n):lista(NULL)
      {
         lista=new T[n];
         assert(lista);
      }
~Array()
{delete[] lista; }
T get(int i)const;
void set(T a,int i);
};
template <class T>
T Array<T>::get(int i)const
{
return (T)lista[i];
}
template <class T>
void Array<T>::set(T t,int i)
{
lista[i]=t;
}
// patron52.cpp
```

Figura 10.Programa patron53.cpp

```
template <class T,size_t N>
class Array {
T lista[N];
public :
Array(){}
~Array(){}
T get(int i)const;
void set(T a,int i);
};
template <class T,size_t N>
T Array<T,N>::get(int i)const
{
return (T)lista[i];
}
template <class T,size_t N>
void Array<T,N>::set(T t,int i)
{
lista[i]=t;
}
// patron53.cpp
```

Figura 11. Programa patron6.cpp

```
template <class T,size_t N>
class Array {
T lista[N];
public :
T const &operator[](const size_t i)const
                          {return (T)lista[i];
                                    }// rvalue
      T &operator[](const size_t i)
                {return lista[i];}// lvalue
};
// patron6.cpp
void main()  // Eduardo Raffo Lecca
{
Array<int,80> x;
    int i;
for(i=0;i<10;i++)
x[i]=i;
for(i=0;i<10;i++)
cout << x[i] << '\t';
cout << endl;
system("PAUSE");
```

ocultan al usuario la complejidad, mostrándola sólo en la medida en que necesite hacer uso de ella. Aquí se cumple la expresión *Small is Beautifull*: la belleza de lo pequeño.

Las plantillas, en un principio fueron creadas para dar el soporte a la programación funcional; siendo resumidas en "el reuso a través de la parametrización". Funciones genéricas y parámetros objetos que personalizan su conducta. Los parámetros son conocidos en tiempos de compilación.

Metaprogramación

Se denomina metaprogramación, al proceso por el cual un programa trata a los programas como datos. La metaprogramación es una práctica común en la programación en LISP, debido a que la estructura central del lenguaje es una estructura lista. La metaprogramación es codificar programas que a su vez codifican otros programas: *Write programs to generate other programs*.

En Teoría de Lenguajes, se define un metalenguaje como un lenguaje usado para describir otro lenguaje.

Un BNF(*Backus-Naur Form*) es un metalenguaje para lenguajes de programación. La forma *Backus-Naur* fue desarrollada para Algol y desde allí es utilizada para describir los lenguajes por medio de gramáticas formales.

Los programas YACC y LEX, el primero escrito por Steve Johnson; son una muestra de programas analizadores que convierten la especificación gramatical de un lenguaje en un analizador(para nuestro caso sintáctico y lexical)[5] y [6]. La entrada a YACC es una gramática de libre contexto en la forma extendida BNF(EBNF) y su salida es un programa que efectúa un *parser* a la gramática. YACC es un metaprograma y las especificaciones son denominadas metadatos, que no están escritas en C, sino en un metalenguaje.

El nombre de metaprogramación, nace en el intento de LISP de contar con una notación cercana a FORTRAN. Esta notación fue denominada notación M o metalenguaje(de M-language). Según McCarthy el autor de LISP, el procesamiento de listas sería usado para el estudio de la computabilidad, el cual en esos días se estudiaba con las máquinas de Turing. Mas tarde la notación fue cambiada a *lambda* para la especificación de las definiciones de funciones.

Todas las estructuras LISP(datos y código) son denominadas *S-expression*. En la práctica expresiones significa *S-expression*.

- Una *S-expression* es cualquiera:
- Un número,
- un string,
- un identificador, o
- una secuencia de *S-expression* entre paréntesis, (*S-expression*$_1$ ... *S-expression*$_n$).

Las excepciones a estas reglas, definen las denominadas formas especiales.

En Scheme, la forma especial **define**, une los identificadores a los valores.

```
(define pi 3.14159) ; constant 'pi'
(define (sqr x) ; function 'square'
        (* x x) )
```

Las expresiones condicionales son implementadas por la forma especial **if** de acuerdo a la sintaxis:

(**if** <predicado> <consecuente> <alternativa>)

Las plantillas de metaprogramación, son programas C++, interpretados en tiempos de compilación[7]; y fueron inventadas por Todo Veldhuizen.

En el año de 1994, Todd Veldhuizen leyó un programa en C++ que había escrito Erwin Unruh y que circuló en el *C++ standards committee meeting*[8]. El programa imprimía una lista de números primos en tiempo de compilación. Esto fue lo que le llevó a inventar las plantillas de metaprogramas. Un metaprograma es un programa que manipula otros programas; tal como ocurre con los compiladores y generadores de parser.

Erwin Unruh había inventado la programación de plantillas en *tiempos de compilación*; y desde este trabajo, Todd Veldhuizen inventa las plantillas de metaprogramación. La metaprogramación había sido descubierta por accidente, al reconocer que el mecanismo de la plantilla provee una rica facilidad para la computación en compile-time.

La metaprogramación es utilizada para generar códigos mas eficientes[9], o para desarrollar algunas evaluaciones en tiempo de compilación; evaluaciones que normalmente son postergadas hasta el *run-time*. En la figura 12, se presenta la plantilla metaprogramación para entregar un entero.

El programa meta1.cpp es denominado como

Figura 12. Programa meta1.cpp

```
#include <iostream.h>
#include <stdlib.h>
template <int i>
class c{
public:
     enum {value=i};
};

// meta1.cpp
void main()  //  Eduardo Raffo Lecca
{
      cout << c<2>::value << endl;
      system("PAUSE");
}
```

Figura 13. meta11.cpp, programa alternativo a meta1.cpp

```
#include <iostream.h>
#include <stdlib.h>
template <int i>
class c{
public:
     static const int value=i;
};

// meta11.cpp
void main()  //  Eduardo Raffo Lecca
{
      cout << c<2>::value << endl;
      system("PAUSE");
}
```

Figura 14. Metaprograma meta2.cpp

```
#include <iostream.h>
#include <stdlib.h>
template <int x>
class sqr{
public:

     enum {value=x*x};
};

// meta2.cpp
void main()  //  Eduardo Raffo Lecca
{

      cout << sqr<2>::value << endl;
      system("PAUSE");
}
```

Figura 15. Metaprograma meta3.cpp

```
#include <iostream.h>
#include <stdlib.h>

template<int N>
class Factorial {
public:
    enum {value=N*Factorial<N-1>::value};
};
class Factorial<1>{
public:
    enum {value=1};
};
// meta3.cpp
void main()    // Eduardo Raffo Lecca
{
      cout << Factorial<5>::value << endl;
      system("PAUSE");
}
```

metafunción. Se aprecia el diseño de la función para ser evaluada en *run-time*; donde el argumento de la metafunción, es pasado a la plantilla como un parámetro. Los componentes o valores son accesados a través del operador de ámbito (**::**).

El uso del tipo enumerado sirve para alojar el resultado de los cálculos. Es importante este tipo debido a la ausencia de la definición de constantes dentro de una clase, para algunos compiladores. Este mismo programa al estilo de Veldhuizen[10], hace que el valor de retorno(value) esté definido como **static const int** value (ver figura 13) **.**

En la figura 14, se presenta el código para la plantilla metaprogramación, que retorna el cuadrado de un entero. Las plantillas fueron creadas para dar soporte a la programación genérica, pero accidentalmente proveen la habilidad de desarrollar computaciones estáticas y generación de código.

Estas técnicas se han hecho populares porque resuelven importantes problemas en computación, como proveer librerías de dominio de abstracciones específicas, sin pérdida de performance[12].

La habilidad de los parámetros plantilla *non-type*, hace posible desarrollar computaciones enteras en *compile-time*; tal como ocurre con la plantilla que computa el factorial desde su argumento, según la figura 15.

La recursión es más importante en la metaprogramación, que el estilo de bucles. Aunque la recursión es natural en LISP, en C/C++ es dura su aplicación.

La clave para diferenciar una función factorial *compile-time* y *run-type*, se halla en la expresión de condición de finalización. El meta factorial usa una plantilla de especialización para describir el caso cuando N es uno, como un mecanismo *pattern-matching* para describir esta conducta. Este mecanismo es común en los lenguajes funcionales modernos, y se explica cuando una función es definida en varias ecuaciones, cada una con diferentes valores de asignación(*left-hand side*) y contiene un patrón en su parámetro formal. Este patrón deberá hacer la correspondencia o *matching* con el argumento de la función.

Cuando se pide la instanciación o se invoca a Factorial<5>, el compilador instancia Factorial<4> y esto requiere Factorial<3>, etc. Al final se requiere la instancia de Factorial<1> que está dado por una especialización explícita. Esta juega el rol de la base en la recursión.

En la figura 16, se presenta el metaprograma, para hallar el menor de dos enteros.

Figura 16. Metaprograma meta4.cpp

```
#include <iostream.h>
#include <stdlib.h>
template <int x,int y>
class min{
public:
     enum {value=x<y?x:y};
};

// meta4.cpp
void main()  //  Eduardo Raffo Lecca

{
      cout << min<2,1>::value << endl;
      system("PAUSE");
}
```

CONCLUSIONES

La parametrización en los algoritmos, es un aporte a las técnicas de programación, y permite resolver nuevos problemas.

La programación genérica está mucho más centrada en los algoritmos que en los datos y su postulado fundamental puede sintetizarse en una palabra: generalización.

La Metaprogramación es el proceso por el cual un programa trata a los programas como datos. La metaprogramación es codificar programas que a su vez codifican otros programas.

La motivación por la metaprogramación radica en la combinación de tres factores: eficiencia, expresión y corrección. Todo lo cual es opuesto a la programación tradicional, donde siempre existen conflictos entre expresión y corrección.

REFERENCIAS BIBLIOGRÁFICAS

1. Veldhuizen, Todd L. En: http://osl.iu.edu/~tveldhui/papers/2000/gcse00/. (Visitado: 14-06-06).

2. Watt, David A. (1989). Programing: Language Concepts and Paradigms. Prentice Hall. USA.

3. Sebesta, Robert W. (1996). Concepts of Programming Languages. Addison-Wesley Publishing Company, Third Edition. USA.**1.** Johnson, Stephen C. (1979). Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual , Vol. 2b, pp. 353-387.

4. Raffo Lecca, E. (1995). El Poder del Turbo C++ ver. 3.0. Lima, Perú.

5. Kernighan, Brian W., Pike, Rob (1987). El entorno de programación en UNIX. Prentice-Hall Hispanoamericana S.A., México.

6. Johnson, Stephen C. (1979). YACC: Yet Another Compiler Compiler. UNIX Programmer's Manual. Vol. 2b, pp. 353-387.

7. Veldhuizen, Todd, (1995). Using C++ Template Metaprograms, C++ Report, Vol. 7, No. 4, pp. 36-43.

8. Unruh, E. (1994). Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.

9. Veldhuizen, Todd, Ponnambalam, K. (1996). Linear Algebra with C++ Template Metaprograms, Dr. Dobbs Journal No. 256.

10. Veldhuizen, Todd. (1995). Expression templates. C++ Report, SIGS Publications Inc., ISSN 1040-6042, Vol. 7, No. 5, pp. 26-31.

11. Pescio, Carlo. (1997). Binary Constants Using Template Metaprogramming, C/C++ Users Journal, pp. 51-54.

12. Veldhuizen, Todd. (1999). C++ Templates as partial Evaluation, In Proceeding of PEPM' 99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulations, ed. O. Danvy, San Antonio, University of Aarhus , Dept. of Computer Science, pp.13-18.