**Siguiente:** [8 De C a](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node9.html) **Arriba:** [Introducción a la Programación Orientada a Objetos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/tutorial.html) **Anterior:** [6 Aún más Orientación a Objetos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node7.html)   
  
**Subsecciones**

* [7.1 El Lenguaje de Programación C](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00810000000000000000)

* + [7.1.1 Tipos de Datos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00811000000000000000)

* + [7.1.2 Instrucciones](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00812000000000000000)

* + [7.1.3 Expresiones y Operadores](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00813000000000000000)

* + [7.1.4 Funciones](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00814000000000000000)

* + [7.1.5 Apuntadores y Arreglos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00815000000000000000)

* + [7.1.6 Un Primer Programa](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00816000000000000000)

* [7.2 ¿Qué Sigue Después?](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm" \l "SECTION00820000000000000000)

**7 Introducción a C++**

Peter Müller   
Globewide Network Academy (GNA)   
*[pmueller@uu-gna.mit.edu](mailto:pmueller@uu-gna.mit.edu)*

Esta sección es la primera parte de la introducción a C++. Aquí nos enfocamos en el lenguaje C, del cuál C++ fué adoptado. C++ extiende el lenguaje de programación C con una fuerte tipificación, algunas características y - más importante aún - conceptos de orientación a objetos.

**7.1 El Lenguaje de Programación C**

Desarrollado en en la última parte de los 1970s, C se constituyó en un enorme éxito debido al desarrollo de UNIX que fue escrito casi por completo en este lenguaje [[4](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node12.html#kernighan:77a)]. En contraste con otros lenguajes de alto nivel, C fué escrito por programadores para programadores. De tal manera que algunas veces permite, por decirlo así, cosas extrañas que en otros lenguajes como Pascal están prohibidas debido a su mala influencia sobre el estilo de programación. De todas maneras, cuando se usa con alguna disciplina, C es un lenguaje tan bueno como cualquier otro.

Los comentarios en C se encierran en /\* ... \*/. Y no se pueden anidar .

**7.1.1 Tipos de Datos**

La Tabla [7.1](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm#l6typetab) describe los tipos de datos integrados de C. El *Size* (tamaño) especificado se mide en bytes en una PC 386 corriendo Linux 1.2.13. El *Domain* (dominio) provisto se basa en el valor del *Size* (tamaño). Puedes obtener información acerca del tamaño de los tipos de datos por medio del operador sizeof.

|  |
| --- |
| **Tabla 7.1:**   Tipos integrados. |
| \begin{tabular} {\vert l\vert p{0.199\textwidth}\vert c\vert l\vert} \hline {\bf...  ...  Unsigned short integer &   2 &   0..$2^{16} - 1$\space \\  \hline\end{tabular} |

Las variables de estos tipos se definen simplemente precediendo el nombre con el tipo:

int an\_int;

float a\_float;

long long a\_very\_long\_integer;

Por medio de struct puedes combinar varios tipos dirferentes en forma conjunta. En otros lenguajes, ésto es llamado algunas veces *record* (registro):

struct date\_s {

int day, month, year;

} aDate;

La *definición* de arriba de aDate es también la *declaración* de una estructura llamada date\_s. Podemos definir otras variables de este tipo referenciando la estructura por el nombre:

struct date\_s anotherDate;

No es obligatorio poner nombre a las estructuras. Si omitimos el nombre, simplemente no podemos reutilizarlas. Sin embargo, si damos nombre a una estructura, podemos simplemente *declararla* sin necesidad de definir una variable:

struct time\_s {

int hour, minute, second;

};

Podemos usar esta estructura como se mostró para anotherDate. Esto es muy similar al tipo de definición conocido en otros lenguajes, donde un *tipo* es *declarado* previamente a la *definición* de una variable de ese tipo.

Las variables deben ser definidas antes de usarse. Estas definiciones deben ocurrir antes de cualquier instrucción, así, constituyen al parte inicial o superior dentro de un *bloque de instrucciones*.

**7.1.2 Instrucciones**

C define todos las instrucciones comunes para control de flujo. Las instrucciones son terminadas por un punto y coma " ;". Podemos agrupar instrucciones múltiples dentro de bloques, encerrando aquéllas dentro de llaves. Dentro de cada bloque, podemos definir nuevas variables :

{

int i; /\* Define una variable i global \*/

i = 1; /\* Asigna a i el valor 0 \*/

{ /\* Inicio de un nuevo bloque \*/

int i; /\* Define una variable i local \*/

i = 2; /\* Pone su valor en 2 \*/

} /\* Fin del bloque \*/

/\* Aquí i es otra vez 1 en el bloque externo \*/

}

La Tabla [7.2](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm#l6stmttab) muestra todas las instrucciones de control de flujo:

|  |
| --- |
| **Table 7.2:**  Instrucciones. |
| \begin{tabular} {\vert p{0.4\textwidth}\vert p{0.5\textwidth}\vert} \hline {\bf ...  ...stmt} & Repeat {\em stmt} as long as {\em expr} is TRUE. \\  \hline\end{tabular} |

La instrucción for es la única instrucción que realmente difiere de las instrucciones for conocidas de otros lenguajes. Todas las otras instrucciones más o menos difieren solamente en la sintaxis. Lo que sigue son dos bloques exactamente equivalentes en su funcionalidad. Uno usa el bucle while mientras que el otro usa la variante for:

{

int ix, sum;

sum = 0;

ix = 0; /\* inicialización \*/

while (ix < 10) { /\* condición \*/

sum = sum + 1;

ix = ix + 1; /\* incremento \*/

}

}

{

int ix, sum;

sum = 0;

for (ix = 0; ix < 10; ix = ix + 1)

sum = sum + 1;

}

Para entender ésto, hay que saber que una asignación es una expresión.

**7.1.3 Expresiones y Operadores**

En C casi todo es una expresión. Por ejemplo, la instrucción de asignación "=" regresa el valor del operando derecho. Como un "efecto secundario", también establece el valor del operando izquierdo. Así,

ix = 12;

le da a *ix* el valor de 12 (asumiendo que *ix* es del tipo apropiado). Como la asignación también es una expresión, podemos combinar varias de ellas ; por ejemplo :

kx = jx = ix = 12;

¿Qué sucede ? La primera asignación le da a *kx* el valor de su derecha. Este es el valor de la asignación a *jx*. Pero ésta última es el valor de la asignación a *ix*. El valor de ix es 12, el cuál es regresado a *jx* que a su vez es regresado a *kx*. Así, hemos expresado

ix = 12;

jx = 12;

kx = 12;

en una línea.

Verdadero en C se define como sigue. El valor 0 (cero) significa FALSE (falso). Cualquier otro valor es TRUE (verdadero). Por ejemplo, la función estándar *strcmp()* lleva dos strings como argumentos y regresa -1 si el primero es inferior al segundo, 0 si son iguales y 1 si el primero es mayor que el segundo. Para comparar si dos strings *str1* y *str2* son iguales, seguido ves la siguiente construcción if:

if (!strcmp(str1, str2)) {

/\* str1 es igual a str2 \*/

}

else {

/\* str1 no es igual a str2 \*/

}

El signo de admiración indica el NOT booleano. Así, la expresión evalúa TRUE solamente si *strcmp()* regresa 0.

Las expresiones son combinaciones de ambos *términos* y *operadores*. Los primeros podrían ser constantes, variables o expresiones. De los segundos, C ofrece todos los operadores conocidos de otros lenguajes. Sin embargo, ofrece algunos operadores que podrían ser vistos como abreviaciones a combinaciones de otros operadores. La Tabla [7.3](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node8.htm#l6operatortab) muestra los operadores disponibles. La segunda columna muestra su prioridad, donde los números más chicos indican prioridad más alta y números iguales, prioridad igual. La última columna enlista el orden de evaluación.

|  |
| --- |
| **Table 7.3:**  Operadores. |
| \begin{tabular} {\vert c\vert c\vert p{0.4\textwidth}\vert l\vert} \hline {\bf O...  ...rom right \\  \hline , & 15 & Comma operator & from left \\  \hline\end{tabular} |

La mayoría de estos operadores ya son conocidos para ti. Sin embargo, algunos necesitan alguna descripción adicional. Primero que nada, nótese que los operadores booleanos binarios &, ^ and | son de menor prioidad que los operadores de igualdad == and !=. Consecuentemente, si quieres checar patrones de bits como en

if ((pattern & MASK) == MASK) {

...

}

debes encerrar la operación binaria entre paréntesis [[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1298).

Los operadores de incrementeo ++ y $-\,-$pueden ser explicados por medio del siguiente ejemplo. Si tienes la siguiente secuencia de instrucciones

a = a + 1;

b = a;

puedes usar el operador de preincremento

b = ++a;

En forma similar, si tienes el siguiente orden de instrucciones :

b = a;

a = a + 1;

puedes usar el operador de postincremento

b = a++;

De esta manera, el operador de preincremento primero incrementa su variable asociada y entonces, regresa el nuevo valor ; mientras que el operador de postincremento primero regresa el valor y después incrementa su variable. Se aplican las mismas reglas para los operadores de pre- y postdecremento $-\,-$.

Las llamadas a funciones, las asignaciones anidadas y los operadores de incremento/decremento provocan efectos secundarios cuando se aplican. Esto puede introducir dependencias del compilador debido a que el orden de evaluación en algunas situaciones es dependiente del compilador. Considera el siguiente ejemplo que demuestra esto:

a[i] = i++;

La pregunta es, si el nuevo o el viejo valor de *i* se usa como subíndice dentro del arreglo *a* depende del orden que use el compilador para evaluar la asignación.

El operador condicional ?: es una abreviatura para una instrucción if de uso común. Por ejemplo para asignarle a *max* el valor máximos de entre *a* y *b*, podemos usar la siguiente instrucción if:

if (a > b)

max = a;

else

max = b;

Estos tipos de instrucciones if pueden ser más cortas si se escriben así

max = (a > b) ? a : b;

El siguiente operador poco común es el operador de asignación. Nosotros seguido estamos usando asignaciones de la forma siguiente

expr1 = (expr1) op (expr2)

por ejemplo

i = i \* (j + 1);

En estas asignaciones, el valor de la izquierda también aparece en el lado derecho. Por medio de lenguaje informal, podríamos expresar esto como "*poner al valor de i i el valor actual de i multiplicado por la suma del valor de j más 1*''. Usando una manera más natural, diríamos más bien "*Multiplicar i por la suma del valor de j más 1*''. C nos permite abreviar estos tipos de asignaciones a

i \*= j + 1;

Nosotros podemos hacer éso con casi todos los operadores binarios. Nótese que el operador de asignación de arriba realmente implementa la forma larga "*j* + 1'' no esté entre paréntesis.

El último operador poco común es el operador coma (,). Se explica mejor por medio de un ejemplo:

i = 0;

j = (i += 1, i += 2, i + 3);

Este operador toma sus argumentos y los evalúa de izquierda a derecha y regresa el valor de la expresión de la derecha. Así, en el ejemplo de arriba, el operador primero evalúa "*i* += 1" lo cuál, como un efecto secundario, incrementa el valor de *i*. Después, es evaluada la siguiente expresión "*i* += 2" la cuál añade 2 a *i* conduciendo al valor de 3. La tercera expresión es evaluada y su valor es regresado como el resultado del operador. Así, se le asigna 6 a *j*.

El operador coma presenta una particular posiblidad de cometer el siguiente error cuando se usan arreglos de *n*-dimensiones siendo $n\gt 1$. Un error frecuente es usar una lista de índices separados por comas para tratar de accesar un elemento:

int matrix[10][5]; // matriz de 2-dimensiones

int i;

...

i = matrix[1,2]; // ¡¡NO FUNCIONARA ! !

i = matrix[1][2]; // OK

Lo que actualmente sucede en el primer caso es que la lista separada por comas es interpretada como un operador coma. Por consecuencia, el resultado es 2, lo que conduce a una asignación de la dirección a los terceros cinco elementos de la matriz.

Algunos de ustedes se podrían preguntar que es lo que C hace con valores que no son usados. Por ejemplo en las instrucciones de asignación que hemos visto anteriormente,

ix = 12;

jx = 12;

kx = 12;

tenemos tres líneas cada una de las cuáles regresa 12. La respuesta es que C ignora los valores que no se usan. Esto lleva a algunas cosas extrañas. Por ejemplo, tú podrías escribir algo como ésto :

ix = 1;

4711;

jx = 2;

Olvidemos estas cosas extrañas y regresemos a algo de más utilidad. Hablemos de *funciones*.

**7.1.4 Funciones**

Debido a que C es un lenguaje procedimental, permite la definición de *funciones*. Los procedimientos son "simulados" por funciones que no regresan "ningún valor". Este valor es un tipo especial llamado void.

Las funciones se declaran en forma similar a las variables, pero aquéllas encierran sus argumentos entre paréntesis (aún si no llevan argumentos, los paréntesis deben ser especificados):

int sum(int to); /\* Declaración de sum con un argumento \*/

int bar(); /\* Declaración de bar sin argumentos \*/

void foo(int ix, int jx);

/\* Declaración de foo con dos argumentos \*/

Para definir efectivamente una función, simplemente añade su cuerpo :

int sum(int to) {

int ix, ret;

ret = 0;

for (ix = 0; ix < to; ix = ix + 1)

ret = ret + ix;

return ret; /\* regresa el valor de la función \*/

} /\* sum \*/

C solamente permite que pases los argumentos de las funciones por valor. Por consecuencia, no puedes cambiar el valor de un argumento dentro de la función. Si tu debes pasar un argumento por referencia, debes programarlo por ti mismo(a). Para tal efecto usas *apuntadores*.

**7.1.5 Apuntadores y Arreglos**

Uno de los problemas más comunes al programar en C (y algunas veces en C++) es la comprensión de apuntadores y arreglos. En C (C++) ambos están altamente relacionados ; con algunas pequeñas pero esenciales diferencias. Tú declaras un apuntador poniendo un asterisco entre el tipo de datos y el nombre de la variable o función :

char \*strp; /\* strp es un "apuntador a char" \*/

Se accesa el contenido de un apuntador derreferenciándolo por medio de -otra vez- el asterisco :

\*strp = 'a'; /\* Un carácter simple \*/

Al igual que en otros lenguajes, se debe proveer algún espacio para el valor al cuál el apuntador está apuntando. Un apuntador a caracteres puede ser usado para apuntar a una secuencia de caracteres : el *string*. Los strings en C están terminados por un carácter especial NUL (0 o como char '${\backslash}0$'). Así, puedes tener strings de cualquier longitud. Los strings se encierran en dobles comillas:

strp = "hello";

En este caso, el compilador automáticamente agrega el carácter de terminación NUL. De este modo*strp* apunta a una secuencia de 6 caracteres. El primer carácter es 'h', el segundo es 'e' y así sucesivamente. Podemos accesar estos caracteres por medio de un índice en *strp*:

strp[0] /\* h \*/

strp[1] /\* e \*/

strp[2] /\* l \*/

strp[3] /\* l \*/

strp[4] /\* o \*/

strp[5] /\* \0 \*/

El primer carácter también es igual a "\**strp*" que puede ser escrito como "\*(*strp* + 0)". Esto nos lleva a algo llamado *aritmética de apuntadores* lo cuál constituye una de las poderosas cararacterísticas de C. Así, tenemos las siguientes ecuaciones :

\*strp == \*(strp + 0) == strp[0]

\*(strp + 1) == strp[1]

\*(strp + 2) == strp[2]

...

Nótese que estas ecuaciones son verdaderas para cualquier tipo de datos. La suma **no** está orientada a bytes, está orientada al tamaño del correspondiente tipo del apuntador.

El apuntador *strp* puede ser puesto en otras localidades. Su destino puede *variar*. En contraste con eso, *los arreglos* son apuntadores *fijos*. Apuntan a una predefinida área de memoria, la cuál es especificada por corchetes :

char str[6];

Puedes ver a *str* como un apuntador constante apuntando a una área de 6 caracteres. **No** podemos usar ésto del siguiente modo:

str = "hallo"; /\* ERROR \*/

debido a que esto significaría cambiar el apuntador para que apuntara a 'h'. Debemos copiar el string al área de memoria provista. Por lo tanto, usamos una función llamada strcpy() la cuál es parte de la biblioteca estándar de C.

strcpy(str, "hallo"); /\* Ok \*/

Nota sin embargo, que podemos usar *str* en cualquier caso donde un apuntador a carácter es utilizado, debido a que es un apuntador (si bien fijo).

**7.1.6 Un Primer Programa**

Presentamos aquí el primer programa tab frecuentemente usado : un programa que despliega "Hello, world !" en tu pantalla :

#include <stdio.h>

/\* Aquí deberían ir las variables globales \*/

/\* Las definiciones de funciones deberián ir aquí \*/

int

main() {

puts("Hello, world!");

return 0;

} /\* main \*/

La primera línea se ve algo extraña. Su explicación requiere alguna información acerca de como los programas en C (y C++) son manejados por el compilador. La compilación se divide gruesamente en dos pasos. El primer paso es llamado "preprocesamiento" y se usa para preparar código C "en bruto". En este caso, este paso toma la primera línea como un argumento para incluir un archivo llamado *stdio.h* dentro del código fuente. Los corchetes angulares simplemente indican que el archivo debe ser buscado en la trayectoria de búsqueda estándar configurada para tu compilador. El archivo en sí mismo provee algunas declaraciones y definiciones para input/output estándar. Por ejemplo, declara la función llamada *put()*. Este paso del preprocesador también borra todos los comentarios.

En el segundo paso, el "código en bruto" de C generado se compila como un ejecutable. Cada ejecutable debe definir una función llamada *main()*. Es esta función la que es llamada cuando el programa es arrancado. Esta función regresa un entero que significa el estatus de salida del programa.

La función *main()* puede llevar argumentos que vienen a representar los parámetos de la línea de comandos. Los presentamos aquí solamentes pero no damos ninguna explicación ulterior :

#include <stdio.h>

int

main(int argc, char \*argv[]) {

int ix;

for (ix = 0; ix < argc; ix++)

printf("My %d. argument is %s\n", ix, argv[ix]);

return 0;

} /\* main \*/

El primer argumento *argc* regresa simplemente el número de argumentos dados en la línea de comandos. El segundo argumento *argv* es un arreglo de strings. (Recuerda que todos los strings se representan por apuntadores a caracteres. Así, *argv* es un arreglo de apuntadores a caracteres.)

**7.2 ¿Qué Sigue Después?**

Esta sección está lejos de ser completa. Solamente queremos darte una expresión de lo que es C. También queremos presentar algunos conceptos básicos que usaremos en la siguiente sección. Algunos conceptos de C son mejorados en C++. Por ejemplo, C++ introduce el concepto de *referencias* lo cuál permite algo similar a llamar por referencia en las llamadas a funciones.

Sugerimos que tomes tu compilador local y empieces a escribir unos pocos programas (si no estás familiarizado con C, por supuesto). Un problema para los principiantes es que frecuentemente las funciones de biblioteca no son conocidas. Si tines un sistema UNIX trata de usar el comando man para obtener algunas descripciones. Especialmente podrías intentar :

man gets

man printf

man puts

man scanf

man strcpy

También sugerimos que te consigas un buen libro sobre C(o que encuentres uno de los tutoriales en-línea). Tratamos de explicar todo lo que presentamos en las siguientes secciones. Sin embargo, no hay nada de malo en tener alguna referencia a la mano.

**8.2 Primeras Extensiones Orientadas a Objetos**

  En esta sección presentamos como los conceptos orientados a objetos de la sección [4](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node5.html#lecture3) se usan en C++.

**8.2.1 Clases y Objetos**

C++ permite la declaración y la definición de clases. Las instancias de las clases se llaman *objects*. Recuerda nuevamente el ejemplo del programa de dibujo de la sección [5](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node6.html#lecture4). Ahí hemos desarrollado un clase *Point* (punto). En C++ esto se vería así:

class Point {

int \_x, \_y; // coordinadas del punto

public: // principio de la sección de interface

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Point apoint;

Esto declara una clase *Point* y define un objeto *apoint*. Se puede pensar de una definición de clase como una definición de una estructura con funciones (o "métodos"). Adicionalmente, tú puedes especificar los *derechos de acceso* en más detalle. Por ejemplo, *\_x* y *\_y* son **private** (privados), debido a que los elementos de las clases son privados "por default". Consecuentemente, nosotros debemos explícitamente "switchear" los derechos de acceso para declarar que los siguientes sean **públicos**. Logramos éso por medio de la palabra clave public seguida de dos puntos ( : ) Cada elemento siguiente a esta palabra clave será ahora accesible desde afuera de la clase.

Podemos volver a los derechos de acceso privados empezando otra sección privada con private:. Esto se puede hacer las veces que sea necesario:

class Foo {

// privado "por default" ...

public:

// lo que sigue es público hasta ...

private:

// ... aquí, donde regresamos a privado ...

public:

// ... y de regreso a público.

};

Recuerda que una estructura struct es una combinación de varios elementos de datos que son accesibles desde afuera. Ahora, podemos expresar una estructura con la ayuda de una clase, donde todos los elementos son declarados para que sean públicos :

class Struct {

public: // Los elementos de estructuras son públicos "por default"

// elementos, métodos

};

Esto es exactamente lo que C++ hace con struct. Las estructuras son manejadas como clases. Donde los elementos de clases (definidas con class) son privadas "por default", los elementos de de las estructuras (definidos con struct) son públicos. Sin embargo, también podemos usar private: para convertir una sección a privada dentro de una estructura.

Regresemos a nuestra clase *Point*. Su interface empieza con la sección pública donde definimos cuatro métodos. Dos por cada coordenada para establecer y obtener su valor. Los métodos para establecer (set) solamente están definidos. Su funcionalidad real está aún por definirse. Los métodos para obtener (get) tienen un cuerpo de función : Están definidos *dentro* de la clase o, en otras palabras, son *métodos insertados* (inlined methods).

Este tipo de definición de método es útil para cuerpos de función pequeños y sencillos. También mejoran el desempeño, debido a que los cuerpos de los métodos insertados son "copiados" dentro del código dondequiera que una llamada a tal método tenga lugar.

Por el contrario, las llamadas a los métodos para establecer (set methods) resultarían en una "real" llamada a función. Definimos estos métodos afuera de la declaración de la clase. Esto se hace necesario, para indicar a que clase pertenece una definición de método. Por ejemplo, otra clase pdría sencillamente definir un método *setX()* el cuál es totalmente diferente de aquél en *Point*. Debemos poder determinar el *ámbito* de la definición ; por lo tanto, usamos el operador de ámbito " : :" :

void Point::setX(const int val) {

\_x = val;

}

void Point::setY(const int val) {

\_y = val;

}

Aquí definimos el método *setX()* (*setY()*) dentro del ámbito de la clase *Point*. El objeto *apoint* puede usar estos métodos para establecer y para obtener información sobre sí mismo :

Point apoint;

apoint.setX(1); // Inicialización

apoint.setY(1);

//

// x es necesaria a partir de aquí, de modo que la definimos aquí y

// la inicializamos con el valor de la coordenada-x de apoint

//

int x = apoint.getX();

El problema estriba en como los métodos "saben" de cuál objeto son invocados. Esto se realiza pasando implícitamente un apuntador al objeto invocante, a dicho método. Podemos accesar este apuntador dentro de los métodos con la palabra this. Las definiciones de los métodos *setX()* y *setY()* hacen uso de los miembros de la clase (*\_x* y *\_y*, respectively). Si son invocados por un objeto, estos miembros son "automáticamente" mapeados al objeto correcto. Podríamos usar this para ilustrar los que sucede realmente:

void Point::setX(const int val) {

this->\_x = val; // Uso de this para referenciar al objeto

// invocante

}

void Point::setY(const int val) {

this->\_y = val;

}

Aquí nosotros usamos explícitamente el apuntador this para desreferenciar explícitamente el objeto invocante. Afortunadamente, el compilador "inserta" en forma automática estas desreferencias para los miembros de la clase, de ahí que, realmente podemos usar las primeras definiciones de *setX()* y *setY()*. Sin embargo, algunas veces tiene sentido saber que hay un apuntador this disponible que indica al objeto invocante.

En la práctica, nosotros necesitamos llamar los métodos "set" para inicializar un objeto point [[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1567). Sin embargo, nos gustaría inicializar el punto en el momento que lo definimos. Para ello, usamos los métodos especiales llamados *constructores*.

**8.2.2 Constructores**

  Los constructores son métodos que se usan para inicializar un objeto al momento de su definición. Extendemos nuestra clase *Point* de tal suerte que inicialice un punto con las coordenadas (0, 0):

class Point {

int \_x, \_y;

public:

Point() {

\_x = \_y = 0;

}

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Los constructores tienen el mismo nombre de la clase (de ese modo pueden ser identificados como constructores). No regresan ningún valor. Al igual que otros métodos, pueden llevar argumentos. Por ejemplo, nosotros podríamos querer inicializar un punto en otras coordenadas que no fueran (0, 0). Para tal efecto, definimos un segundo constructor que lleve dos argumentos integer dentro de la clase :

class Point {

int \_x, \_y;

public:

Point() {

\_x = \_y = 0;

}

Point(const int x, const int y) {

\_x = x;

\_y = y;

}

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Los constructores son llamados implícitamente cuando definimos objetos de sus clases :

Point apoint; // Point::Point()

Point bpoint(12, 34); // Point::Point(const int, const int)

Con los constructores podemos inicializar nuestros objetos al momento de la definición tal como lo hemos pedido en la sección [2](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node3.html#lecture1) para nuestra lista ligada sencilla. Ahora, nosotros podemos definir una clase *List* donde los constructores se ocupen de inicializar en forma correcta sus objetos.

Si queremos crear un punto a partir de otro punto, es decir, copiando las propiedades de un objeto a uno recién creado, algunas veces tenemos que tener cuidado con el proceso de copiado. Por ejemplo, considera la clase *List* que asigna memoria para sus elementos en forma dinámica. Si queremos crear una segunda lista que sea una copia de la primera, debemos asignar memoria y copiar los elementos individuales. En nuestra clase *Point* para ello nostros añadimos un tercer constructor que se ocupie de copiar correctamente los valores de un objeto al objeto recién creado :

class Point {

int \_x, \_y;

public:

Point() {

\_x = \_y = 0;

}

Point(const int x, const int y) {

\_x = x;

\_y = y;

}

Point(const Point &from) {

\_x = from.\_x;

\_y = from.\_y;

}

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

El tercer constructor lleva como argumento una referencia constante a un objeto de la clase *Point* y le asigna a *\_x* y a *\_y* los valores correspondientes del objeto provisto.

Este tipo de constructor es tan importante que tiene su propio nombre : *copy constructor* (constructor para copia). Es altamente recomendable que tu proveas dicho constructor para cada una de tus clases, aún si es tan simple como en nuestro ejemplo. El "copy constructor" es llamado en los siguientes casos :

Point apoint; // Point::Point()

Point bpoint(apoint); // Point::Point(const Point &)

Point cpoint = apoint; // Point::Point(const Point &)

Con la ayuda de los constructores hemos cumplido con uno de nuestros requerimientos de implementación de tipos de datos abstractos : La inicialización al momentod de la definición. Aún necesitamos un mecanismo que automáticamente "destruya" un objeto cuando ya no sea válido (por ejemplo, por haber abandonado su ámbito). Para tal efecto, las clases pueden definir *destructores*.

**8.2.3 Destructores**

  Considera una clase *List*. Los elementos de la lista son añadidos y desechados en forma dinámica. El constructor nos ayuda a crear una lista inicial vacía. Sin embargo, cuando salimos del ámbito de definición del objeto lista, nos debemos asegurar que la memoria asignada sea liberada. Para ello definimos un método especial llamado *destructor* el cuál es llamado una vez por cada objeto en el momento de su destrucción :

void foo() {

List alist; // List::List() inicializa una

// lista vacía.

... // añade/elimina elementos

} // ¡Llamada al destructor!

La destrucción de los objetos tiene lugar cuando el objeto abandona su ámbito de definición o es explícitamente destruído. Esto último sucede cuando nosotros dinámicamente asignamos un objeto y lo liberamos cuando ya no nos es necesario.

Los destructores se declaran en forma similar a los constructores. Así, también usan el nombre de la clase que definitoria prefijado con una tilde (~ ):

class Point {

int \_x, \_y;

public:

Point() {

\_x = \_y = 0;

}

Point(const int x, const int y) {

\_x = xval;

\_y = yval;

}

Point(const Point &from) {

\_x = from.\_x;

\_y = from.\_y;

}

~Point() { /\* ¡Nada qué hacer! \*/ }

void setX(const int val);

void setY(const int val);

int getX() { return \_x; }

int getY() { return \_y; }

};

Los destructores no llevan argumentos. Es hasta ilegal definir alguno ; debido a que los destructores son llamados implícitamente en el momento de eliminación : No tienes ninguna oportunidad de especificar argumentos reales.

* [9.1 Herencia](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001010000000000000000)

* + [9.1.1 Tipos de Herencia](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001011000000000000000)

* + [9.1.2 Construcción](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001012000000000000000)

* + [9.1.3 Destrucción](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001013000000000000000)

* + [9.1.4 Herencia Múltiple](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001014000000000000000)

* [9.2 Polimorfismo](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001020000000000000000)

* [9.3 Clases Abstractas](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001030000000000000000)

* [9.4 Sobrecarga de Operadores](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001040000000000000000)

* [9.5 Amigos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001050000000000000000)

* [9.6 Como Escribir un Programa](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001060000000000000000)

* + [9.6.1 Pasos de la Compilación](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001061000000000000000)

* + [9.6.2 Una Nota acerca del Estilo](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001062000000000000000)

* [9.7 Ejercicios](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm" \l "SECTION001070000000000000000)

**9 Más sobre C++**

Peter Müller   
Globewide Network Academy (GNA)   
[*pmueller@uu-gna.mit.edu*](mailto:pmueller@uu-gna.mit.edu)

Esta sección concluye nuestra introducción a C++. Presentamos conceptos orientados a objetos "reales" y constestamos la pregunta acerca de como se escribe realmente un programa en C++.

**9.1 Herencia**

 En nuestro pseudo lenguaje, formulamos la herencia con "hereda de". En C++ estas palabras son reemplazadas por dos puntos ( : ). Como ejemplo, diseñemos una clase para puntos en 3a. dimensión. Por supuesto que queremos reutilizar nuestra clase *Point* ya existente. Empezamos por diseñar nuestra clase como sigue :

class Point3D : public Point {

int \_z;

public:

Point3D() {

setX(0);

setY(0);

\_z = 0;

}

Point3D(const int x, const int y, const int z) {

setX(x);

setY(y);

\_z = z;

}

~Point3D() { /\* Nada que hacer \*/ }

int getZ() { return \_z; }

void setZ(const int val) { \_z = val; }

};

**9.1.1 Tipos de Herencia**

  Podrás notar nuevamente la palabra clave public usada en la primera línea de la definición de la clase (su *signature*, firma o rúbrica en español). Esto es necesario porque C++ distingue dos tipos de herencia: *pública* y *privada*. "Por default", las clases se derivan unas de otras en forma privada. Consecuentemente, debemos decirle explícitamente al compilador que use herencia pública.

El tipo de herencia influye sobre los privilegios de acceso a elementos de las diversas superclases. Utilizando la herencia pública, todo lo que es declarado private (privado) en una superclase, permanece private en la subclase. En forma semejante, todo lo que es public (público) permanece public. Cuando se usa la herencia privada, las cosas son muy diferentes, tal como se muestra en la tabla [9.1](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm#l8accesstab).

|  |
| --- |
| **Table 9.1:**  Privilegios de acceso y herencia. |
| \begin{tabular} {\vert l\vert l\vert l\vert} \hline  & \multicolumn{2}{\vert c\v...  ...rivate & protected \\  \hline  public & private & public \\  \hline\end{tabular} |

La columna de la izquierda enlista los privilegios de acceso posibles para los elementos de clases. También incluye un tercer tipo protected (protegido). Este tipo se usa para elementos que deberían ser usados directamente en las subclases pero que no debería estar accesibles desde afuera. Así, uno podría decir que elementos de este tipo están entre privados y públicos en el sentido que pueden ser utilizados dentro de la jerarquía de clase cuya raíz está representada por la clase correspondiente.

La segunda y tercera columna muestran los privilegios de acceso resultantes de los elementos de una superclase cuando la subclase es derivada en forma privada y en forma pública, respectivamente.

**9.1.2 Construcción**

  Cuando creamos una instancia de la clase *Point3D* es llamado su constructor. Desde el momento que *Point3D* se deriva de *Point* el constructor de la clase *Point* también es llamado. Sin embargo, este constructor es llamado *antes* que se ejecute el cuerpo del constructor de la clase *Point3D*. En general, anterior a la ejecución del cuerpo particular del constructor, los constructores de cada superclase son llamados para inicializar su parte del objeto creado.

Cuando creamos un objeto con

Point3D point(1, 2, 3);

es invocado el segundo constructor de *Point3D*. Antes de la ejecución del cuerpo del constructor, es invocado el constructor *Point()*, para inicializar la parte *point* del objeto *point*. Afortunadamente, hemos definido un constructor que no lleva argumentos. Este constructor inicializa las coordenadas de 2a. dimensión *\_x* y *\_y* a 0 (cero). Como *Point3D* se deriva solamente de *Point* no hay otras llamadas a constructores y el cuerpo de *Point3D(const int, const int, const int)* es ejecutado. Aquí, nosotros invocamos los métodos *setX()* y *setY()* para override explícitamente las coordenadas de 2a. dimensión. Subsecuentemente, es establecido el valor de la tercera coordenada *\_z*.

Esto es muy poco satisfactorio porque hemos definido un constructor *Point()* que lleva dos argumentos para inicializar sus coordenadas con ellos. Así, debemos solamente poder decir que, en lugar de usar el *constructor "de default"* *Point()*, debería de usarse el parametrizado *Point(const int, const int)*. Nosotros podemos hacer eso por medio de especificar los constructores deseados después de un signo de dos puntos ( : ) justamente antes del cuerpo del constructor *Point3D()*:

class Point3D : public Point {

...

public:

Point3D() { ... }

Point3D(

const int x,

const int y,

const int z) : Point(x, y) {

\_z = z;

}

...

};

Si hubieran más superclases, simplemente proveeríamos sus llamadas a constructor como una lista separada por comas. Usamos también este mecanismo para crear objetos contenidos. Por ejemplo, supongamos que la clase *Part* solamente define un constructor con un argumento. Entonces, para crear correctamente un objeto de la clase *Compound* debemos invocar *Part()* con su(s) argumento(s):

class Compound {

Part part;

...

public:

Compound(const int partParameter) : part(partParameter) {

...

}

...

};

Esta inicialización dinámica también puede ser usada con tipos de datos integrados. Por ejemplo, los constructores de la clase *Point* podrían ser escritos así:

Point() : \_x(0), \_y(0) {}

Point(const int x, const int y) : \_x(x), \_y(y) {}

Deberías usar este método de inicialización tan seguido como sea posible, porque ésto permite al compilador crear variables y objetos correctamente inicializados en lugar de crearlos con un valor "de deafult" y usar una asignación adicional (u otro mecanismo) para establecer su valor.

**9.1.3 Destrucción**

 Si un objeto se destruye, por ejemplo al dejar su ámbito de definición, se invoca el destructor de la clase correspondiente. Si esta clase es derivada de otras clases, sus destructores también son llamados ; induciendo una cadena de llamadas recursivas.

**9.1.4 Herencia Múltiple**

 C++ permite que una clase sea derivada de más de una superclase, como ya se mencionó brevemente en las secciones anteriores. Tú puedes derivar fácilmente de más de una clase especificando las superclases en una lista separada por comas :

class DrawableString : public Point, public DrawableObject {

...

public:

DrawableString(...) :

Point(...),

DrawableObject(...) {

...

}

~DrawableString() { ... }

...

};

No usaremos este tipo de herencia en el resto de este tutorial. Por ello no entraremos aquí en más detalles.

**9.2 Polimorfismo**

  En nuestro pseudo lenguaje, nosotros podemos declarar que los métodos de las clases sean virtual (virtuales), con el fin de forzar que su evaluación se base en el contenido de los objetos más que en su tipo. En C++ también podemos usar esto:

class DrawableObject {

public:

virtual void print();

};

La clase *DrawableObject* define un método *print()*, el cuál es virtual. De esta clase podemos derivar otras clases :

class Point : public DrawableObject {

...

public:

...

void print() { ... }

};

Nuevamente, *print()* es un método virtual, debido a que hereda esta propiedad de *DrawableObject*. La función *display()* que es capaz de dibujar cualquier tipo de objeto desplegable, puede por tanto ser definida como :

void display(const DrawableObject &obj) {

// prepare anything necessary

obj.print();

}

Cuando se usan métodos virtuales, algunos compiladores se quejan si el destructor de la clase correspondiente no es declarado también virtual. Esto es necesario cuando se usan apuntadores a subclases (virtuales) cuando llega el momento de destruirlas. Debido a que el apuntador está declarado como superclase, su destructor normalmente sería llamado. Si el destructor es virtual, el destructor del objeto real referenciado es llamado (y entonces, en forma recursiva, todos los destructores de sus superclases). He aquí un ejemplo adoptado de [[1](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node12.html#borland:93a)]:

class Colour {

public:

virtual ~Colour();

};

class Red : public Colour {

public:

~Red(); // Virtualidad heredada de Colour

};

class LightRed : public Red {

public:

~LightRed();

};

Usando estas clases, podemos definir una *paleta* del siguiente modo:

Colour \*palette[3];

palette[0] = new Red; // Creación dinámica de un nuevo objeto Red

palette[1] = new LightRed;

palette[2] = new Colour;

El operador new recién introducido crea un nuevo objeto del tipo especificado en la memoria dinámica y regresa un apuntador a dicho objeto. Así, el primer new regresa un apuntador a un objeto reservado de la clase *Red* y lo asigna al primer elemento del arreglo *palette*. Los elementos de *palette* son apuntadores a *Colour* y, debido a que *Red* es-un(a) *Colour*, la asignación es válida.

El operador opuesto a new es delete que explícitamente destruye un objeto referenciado por el apuntador provisto. Si aplicamos delete a los elementos de *palette* tienen lugar las siguientes llamadas a destructor:

delete palette[0];

// Llama al destructor ~Red() seguido de ~Colour()

delete palette[1];

// Llama a ~LightRed(), ~Red() y ~Colour()

delete palette[2];

// Llama a ~Colour()

Las diversas llamadas a destructor tienen lugar solamente debido al uso de los destructores virtuales. Si no los hubiésemos declarado virtuales, cada delete solamente habrían llamado *~ Colour()* (debido a que *palette[i]* es de tipo apuntador a *Colour*).

**9.3 Clases Abstractas**

  Las clases abstractas se definen justamente como las clases ordinarias. Sin embargo, algunos de sus métodos están designados para ser definidos necesariamente por sus subclases. Solamente mencionamos su *signature* (nombre del método más sus argumentos) incluyendo el tipo que regresa, pero no una definición. Se podría decir que omitimos el cuerpo del métdodo o, en otras palabras, no especificamos "nada". Esto se expresa añadiendo "= 0" después de las "signatures" de los métodos :

class DrawableObject {

...

public:

...

virtual void print() = 0;

};

Esta definición de clase forzaría a que cada clase derivada de la que se crearían objetos, definiera un método *print()*. Estas declaraciones de métodos también son llamadas *métodos puros*.

Los métodos puros también deben ser declarados virtual(es), debido a que nosotros solamente queremos usar objetos de clases derivadas. Las clases que definen métodos puros son llamadas *clases abstractas*.

**9.4 Sobrecarga de Operadores**

  Si recordamos el tipo de datos abstracto para números complejos, *Complex*, podríamos crear un clase de C++ como sigue :

class Complex {

double \_real,

\_imag;

public:

Complex() : \_real(0.0), \_imag(0.0) {}

Complex(const double real, const double imag) :

\_real(real), \_imag(imag) {}

Complex add(const Complex op);

Complex mul(const Complex op);

...

};

Entonces, podríamos hacer uso de números complejos y "calcular" con ellos :

Complex a(1.0, 2.0), b(3.5, 1.2), c;

c = a.add(b);

Aquí, asignamos a *c* la suma de *a* y *b*. Aunque es totalmente correcto, no provee una manera conveniente de expresión. Lo que nosotros más bien quisiéramos usar es el muy familiar signo "+" para expresar la adición de dos números complejos. Afortunadamente, C++ nos permite *sobrecargar* casi todos sus operadores por tipos recién creados. Por ejemplo, podríamos definir un operador "+" para nuestra clase *Complex*:

class Complex {

...

public:

...

Complex operator +(const Complex &op) {

double real = \_real + op.\_real,

imag = \_imag + op.\_imag;

return(Complex(real, imag));

}

...

};

En este caso, hemos hecho del + un miembro de la clase *Complex*. Una expresión de la forma

c = a + b;

es traducida a una llamada a método

c = a.operator +(b);

Así, el operador binario + solamente necesita un argumento. El primer argumento es provisto implícitamente por el objeto invocante (en este caso *a*).

Sin embargo, una llamada a operador puede también ser interpretada como una llamada a función, como en

c = operator +(a, b);

En este caso, el operador sobrecargado **no** es un miembro de una clase. Está más bien definido afuera como una función sobrecargada normal. Por ejemplo, podríamos definir el operador + de esta manera:

class Complex {

...

public:

...

double real() { return \_real; }

double imag() { return \_imag; }

// ¡No hay necesidada de definir el operador aquí!

};

Complex operator +(Complex &op1, Complex &op2) {

double real = op1.real() + op2.real(),

imag = op1.imag() + op2.imag();

return(Complex(real, imag));

}

En este caso, debemos definir métodos de acceso para las partes real e imaginaria debido a que el operador es definido afuera del ámbito de la clase. Sin embargo, el operador está tan cercanamente relacionado a la clase, que tendría sentido permitir al operador que accesara los miembros privados. Esto puede hacerse declarándolo *friend* (amigo) de la clase *Complex*.

**9.5 Amigos**

Nosotros podemos definir que funciones o clases sean amigos de una clase para permitirles acceso directo a sus miembros de datos privados. Por ejemplo, en la sección anterior nos gustaría que la función para el operador + tuviese acceso a los miembros de datos privados *\_real* e *\_imag* de la clase *Complex*. Con ese fin declaramos que el operador + sea amigo de la clase *Complex*:

class Complex {

...

public:

...

friend Complex operator +(

const Complex &,

const Complex &

);

};

Complex operator +(const Complex &op1, const Complex &op2) {

double real = op1.\_real + op2.\_real,

imag = op1.\_imag + op2.\_imag;

return(Complex(real, imag));

}

No deberías usar amigos muy seguido debido a que rompen con el principio del aislamiento de datos en sus fundamentos. Si tienes que usar amigos muy seguido, es señal de que es el momento de reestructurar tu gráfica de herencia.

**9.6 Como Escribir un Programa**

 Hasta ahora, solamente hemos presentado partes de o programas muy pequeños que fácilmente podrían ser manejados en un archivo. Sin embargo, proyectos más grandes, un programa de calendario, por ejemplo, debería ser repartido en secciones manejables, con frecuencia llamadas *módulos*. Los módulos se implementan en archivos separados y ahora discutiremos brevemente se realiza la modularización en C y en C++. Esta discusión está basada en UNIX y el compilador C++ de GNU. Si tú estás usando otras constelaciones, lo que sigue podría variar de tu lado. Esto es especialmente importante para aquéllos que están usando ambientes integrados de desarrollo (IDEs), por ejemplo, Borland C++.

Gruesamente hablando, los módulos consisten de dos tipos de archivos : *descripiciones de interface* y *archivos de implementación*. Para distinguir estos tipos, se usa un conjunto de sufijos cuando se compilan programas de C y C++. La Tabla [9.2](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm#l8suffixtab) muestra algunos de ellos.

|  |
| --- |
| **Tabla 9.2:**  Extensiones y tipos de archivo. |
| \begin{tabular} {\vert l\vert p{0.4\textwidth}\vert} \hline {\bf Extension(s)} &...  ...\  \hline {\tt .tpl} & interface description (templates) \\  \hline\end{tabular} |

En este tutorial usaremos .h para archivos de cabecera (header), .cc para archivos de C++ y .tpl para archivos de definición de plantillas (templates). Aún si "solamente" estamos escribiendo código en C, tiene sentido usar .cc para forzar al compilador a tratarlo como C++. Esto simplifica la combinación de ambos, desde el momento que el mecanismo interno de como el compilador arregla los nombres en el programa difiere en ambos lenguajes [[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1793).

**9.6.1 Pasos de la Compilación**

  El proceso de compilación toma los archivos .cc, los preprocesa (eliminando comentarios, añadiendo archivos de cabecera)[[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1794) y los traduce en *archivos objeto*[[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1760). Sufijos típicos para ese tipo de archivo son .o o .obj.

Después de una compilación exitosa, el conjunto de archivos objeto es procesado por un *linker*. Este programa combina los archivos, añade las bibliotecas necesarias [[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1795)y crea un ejecutable. Bajo UNIX este archivo es llamado *a.out* si no se especifica otro. Estos pasos se ilustran en la Figura [9.1](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node10.htm#l8compilation).

|  |
| --- |
| **Figura 9.1:**   Pasos de la Compilación. |
| \begin{figure} {\centerline{ \psfig {file=FIGS/l8compilation.eps} }}\end{figure} |

Con los compiladores modernos, ambos pasos pueden ser combinados. Por ejemplo, nuestros pequeños programas de ejemplo pueden ser compilados y enlazados ("linkeados") con el compilador C++ de GNU del siguiente modo ("ejemplo.cc" es por supuesto solamente un nombre de ejemplo) :

gcc ejemplo.cc

**9.6.2 Una Nota acerca del Estilo**

 Los archivos de cabecera se usan para describir la interface de archivos de implementación. Por consecuencia, son incluídos en cada archivo de implementación que use la interface del archivo de implementación en particular. Como se mencionó en las secciones anteriores, esta inclusión se logra por medio de una copia del contenido del archivo de cabecera en cada instrucción #include del preprocesador, llevando a un "enorme" archivo C++ en bruto.

Para evitar la inclusión de copias múltiples causadas por dependencias mutuas, usamos *codificación condicional*. El preprocesador también define instrucciones condicionales para checar varios aspectos de su procesamiento. Por ejemplo, podemos checar si una macro ya ha sido definida :

#ifndef MACRO

#define MACRO /\* define MACRO \*/

...

#endif

Las líneas entre #ifndef y #endif son incluídas solamente si MACRO no ha sido ya definida. Podemos usar este mecanismo para prevenir las copias múltiples :

/\*

\*\* Ejemplo para "checar" si un archivo de cabecera ya ha

\*\* sido incluído. Asumamos que el nombre del archivo de cabecera

\*\* es 'myheader.h'

\*/

#ifndef \_\_MYHEADER\_H

#define \_\_MYHEADER\_H

/\*

\*\* Las declaraciones de la interface van aquí

\*/

#endif /\* \_\_MYHEADER\_H \*/

\_\_MYHEADER\_H es un nombre único para cada archivo de cabecera. Tu podrías querer seguir la convención de usar el nombre del archivo con dos subguiones como prefijo. La primera vez que el archivo es incluído\_\_MYHEADER\_H no está definido, así que cada línea es incluída y procesada. La primera línea solo define una macro llamada \_\_MYHEADER\_H. Si en forma accidental el archivo debería ser incluído una segunda vez (mientras se procesa el mismo archivo de entrada), \_\_MYHEADER\_H es definida, así, todo lo que conduzca al #endif es ignorado.

**9.7 Ejercicios**

1.

*Polimorfismo*. Explica por qué

void display(const DrawableObject obj);

no produce la salida deseada.

* [10.1 Tipos Genéricos (Plantillas)](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001110000000000000000)

* [10.2 "Formas de Datos" y Recorrido](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001120000000000000000)

* [10.3 Propiedades de las Listas Ligadas Sencillas](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001130000000000000000)

* [10.4 Implementación de la "Forma"](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001140000000000000000)

* + [10.4.1 Plantillas para los Nodos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001141000000000000000)

* + [10.4.2 Plantillas para la Lista](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001142000000000000000)

* [10.5 Implementación del Iterador](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001150000000000000000)

* [10.6 Ejemplo de Uso](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001160000000000000000)

* [10.7 Discusión](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001170000000000000000)

* + [10.7.1 Separación de la "Forma" y Estrategias de Acceso](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001171000000000000000)

* + [10.7.2 Iteradores](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001172000000000000000)

* [10.8 Ejercicios](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm" \l "SECTION001180000000000000000)

**10 La Lista - Estudio de un Caso**

Peter Müller   
Globewide Network Academy (GNA)   
[*pmueller@uu-gna.mit.edu*](mailto:pmueller@uu-gna.mit.edu)

**10.1 Tipos Genéricos (Plantillas)**

  En C++ los tipos de datos genéricos son llamados *plantillas de clase*[[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#2051) o simplemente *plantillas* (templates) para abreviar. Una plantilla de clase se parece a la definición de una clase normal, en la que algunos aspectos son representados por *placeholders* (sustitutos). En el ejemplo de la lista por venir, usamos este mecanismo para generar listas de diversos tipos de datos :

template <class T>

class List : ... {

public:

...

void append(const T data);

...

};

En la primera línea introducimos la palabra clave template con la cuál se inician todas las declaraciones de plantillas. Los argumentos de una plantilla se encierran en corchetes angulares.

Cada argumento especifica un placeholder (sustituto) en la siguiente definición de clase. En nuestro ejemplo, queremos que la clase *List* sea definida para diversos tipos de datos. Uno podría decir, que queremos definir una *clase de listas*[[*]](http://www.desy.de/gna/html/cc/Tutorial/Spanish/footnode.html#1818). En este caso, la clase de listas está definida por el tipo de objetos que contienen. Usamos el nombre *T* como sustituto. Ahora, nosotros usamos *T* en todos los lugares donde normalmente se espera encontrar el tipo de los objetos reales. Por ejemplo, cada lista provee un método para añadir un elemento a sí misma. Nosotros ahora podemos definir este método tal como se muestra arriba utilizando *T*.

En la práctica, la *definición* de la lista ahora debe especificar el tipo de la lista. Si nos apegamos a la expresión de la clase usada anteriormente, tenemos que *crear una instancia de clase*. De esta instancia de clase, podemos entonces crear instancias de objetos "reales" :

List<int> integerList;

Creamos aquí una instancia de clase de una *List*, la cuál lleva integers como sus elementos de datos. Nosotros especificamos el tipo encerrándolo en corchetes angulares. El compilador aplica ahora el argumento "int" provisto y automáticamente genera una definición de clase donde el sustituto *T* es reemplazado por *int*, por ejemplo, genera la siguiente declaración de método para *append()*:

void append(const int data);

Las plantillas pueden llevar más de un argumento para proveer más sustitutos. Por ejemplo, para declarar una clase diccionario que provea acceso a sus elementos de datos por medio de una clave, uno puede pensar en la siguiente declaración :

template <class K, class T>

class Dictionary {

...

public:

...

K getKey(const T from);

T getData(const K key);

...

};

Usamos aquí dos sustitutos para poder usar diccionarios con varias claves y tipos de datos.

Los argumentos de plantillas pueden usarse también para generar definiciones de clase parametrizadas. Por ejemplo, una pila podría ser implementada por un arreglo de elementos de datos. El tamaño del arreglo podría ser especificado dinámicamente :

template <class T, int size>

class Stack {

T \_store[size];

public:

...

};

Stack<int,128> mystack;

En este ejemplo, *mystack* es una pila de integers utilizando un arreglo de 128 elementos. Sin embargo, no usaremos en lo siguiente clases parametrizadas.

**10.2 "Formas" de Datos y Recorrido**

  En la siguiente discusión distinguimos entre una *"forma"* de estructura de datos y sus *estrategias de recorrido*. Lo primero, es la "apariencia", la cuál ya provee mucha información acerca de los *bloques de construcción* de la estructura de datos.

Una estrategia de recorrido define el *orden* en el cuál los elementos de la estructura de datos serán visitados. Tiene sentido el separar la forma, de las estrategias de recorrido, debido a que las estructuras de datos pueden ser recorridas usando estrategias muy diversas.

El recorrido de una estructura de datos es implementado usando *iteradores*. Los iteradores garantizan la visita a cada ítem de su estructura de datos asociada en un orden bien definido. Deben proveer al menos las siguientes propiedades :

1.

*Elemento actual*. El iterador visita los elementos de datos uno a la vez. El elemento que se visita actualmente es llamado el "elemento actual" ("current element").

2.

*Función sucesor*. La ejecución del paso al siguiente elemento de datos depende de la estrategia de recorrido implementada por el iterador. La función sucesor se usa para regresar el elemento que será visitado en seguida : Regresa el sucesor del elemento actual.

3.

*Condición de terminación*. El iterador debe proveer un mecanismo que cheque si se han visitado todos los elementos, o si falta alguno por visitar.

**10.3 Propiedades de las Listas Ligadas Sencillas**

  Cuando se hace algo orientado a objetos, la primera pregunta a realizar es

*¿Cuáles son los bloques de construcción básicos del ítem a implementar?*

Echa un vistazo a la Figura [10.1](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm#l9sll), que muestra una lista consistente en cuatro rectángulos. Cada rectángulo tiene una bala en el centro, los primeros tres apuntan a su vecino de la derecha. Debido a que el último rectángulo no tiene vecino de la derecha, no hay apuntador.

|  |
| --- |
| **Figura 10.1:**  Bloques de construcción básicos de una lista ligada sencilla. |
| \begin{figure} {\centerline{ \psfig {file=FIGS/sll.eps,width=0.9\textwidth} }}\end{figure} |

Primeramente escojamos nombres para estos bloques de construcción. Hablar de rectángulos no es apropiado, debido a que uno puede pensar en una figura que use círculos o triángulos.

En el campo de las *gráficas* se usa el nombre *nodo*. Un nodo contiene un *apuntador* a su *sucesor*. Así, la lista en la figura consiste en nodos, cada uno de los cuáles tiene *exactamente un* apuntador asociado a él.

Se pueden distinguir tres tipos de nodos :

* El primer nodo (*cabeza*), el cuál no tiene predecesor,
* los nodos de en medio, los cuáles tienen exactamente un predecesor y exactamente un sucesor y
* el último nodo (*cola*), el cuál no tiene sucesor.

Nótese que los nodos no llevan ningún contenido. Esto se debe a que la mera estructura de datos *list* consiste solamente en nodos, puestos en fila uno tras otro. Desde luego que las aplicaciones reales necesitan nodos, llevando algún contenido. Pero en el sentido de la orientación a objetos, ésta es una especialización de los nodos.

Por la figura podemos ver que una lista solamente puede ser usada con una estrategia de recorrido : *cursor hacia adelante*. Inicialmente, la cabeza será el primer elemento actual. La función sucesor simplemente va tras el apuntador de el nodo actual. La función terminación checa que el elemento actual sea efectivamente la cola.

Nótese que no es posible regresar o empezar del medio de la lista. Esto último contradiría el requerimiento de que cada elemento deba ser visitado.

La siguiente pregunta es, ¿Cuáles son las operaciones ofrecidas por una lista ? Una lista solamente define dos nodos bien identificados *cabeza* y *cola*. Echémos un vistazo más profundo a éstos últimos.

Un nuevo nodo puede ser *put-in-front* (puesto en el frente) de la lista de tal modo que :

* su apuntador señale a la cabeza actual,
* el nuevo nodo se convierta en la nueva cabeza.

En forma similar, un nuevo nodo puede fácilmente ser *appended* (añadido) a la cola:

* el apuntador a la cola señala al nuevo nodo,
* el nuevo nodo se convierte en la nueva cola.

La función inversa a poner-en-el-frente es *delete-from-front* (eliminar del frente):

* el nodo sucesor de la cabeza se convierte en la nueva cabeza,
* el antiguo nodo cabeza es descartado.

Deberías poder figurarte por qué no hay una función append inversa barata.

Finalmente, existen otras tres primitivas baratas, cuyo significado es inmediato. Así, no las examinaremos más. Sin embargo, las presentamos aquí para efectos de no dejar huecos :

* *get-first* (obtiene el primero): regresa los datos del nodo cabeza,
* *get-last* (obtiene el último): regresa los datos del nodo cola y por último
* *is-empty* (¿vacía ?): regresa información sobre si la lista está vacía o no.

**10.4 Implementación de la "Forma"**

**10.4.1 Plantillas para los Nodos**

  El bloque básico de construcción de una lista es el *nodo*. Así, declaremos primero una clase para nodos. Un nodo no tiene nada más que un apuntador a otro nodo. Asumamos que este nodo vecino siempre está del lado derecho.

Echa un vistazo a la siguiente declaración de la clase *Node*.

class Node {

Node \*\_right;

public:

Node(Node \*right = NULL) : \_right(right) {}

Node(const Node &val) : \_right(val.\_right) {}

const Node \*right() const { return \_right; }

Node \*&right() { return \_right; }

Node &operator =(const Node &val) {

\_right = val.\_right;

return \*this;

}

const int operator ==(const Node &val) const {

return \_right == val.\_right;

}

const int operator !=(const Node &val) const {

return !(\*this == val);

}

};

Una mirada a la primera versión del método *right()* : contiene un const justamente antes del cuerpo del método. Cuando se usa en esta posición, const declara al método constante en lo que respecta a los elementos del objeto invocante. Por consecuencia, solamente te está permitido usar este mecanismo en declaraciones de métodos o definiciones, respectivamente.

Este tipo de modificador const también se usa para checar sobrecarga. Así,

class Foo {

...

int foo() const;

int foo();

};

declara dos métodos diferentes. El primero se usa en contextos constantes, mientras que el otro se usa en contextos variables.

Aunque la clase plantilla *Node* implementa un simple nodo, parece definir abundante funcionalidad. Hacemos esto porque es buena práctica el ofrecer al menos la siguiente funcionalidad para cada tipo de datos definido :

* *Copy Constructor*. El copy constructor (constructor para copia) se necesita para permitir la definición de objetos que son inicializados a partir de otros ya existentes.
* *operator =*. Todos los objetos deberían saber como asignarse otros objetos (del mismo tipo) a sí mismos. En nuestra clase de ejemplo, ésto es simplemente la asignación de apuntador.
* *operator ==*. Todo objeto debería saber como comparase a sí mismo con otro objeto.

El operador de desigualdad " !=" se implementa usando la definición del operador de igualdad. Recuerda que this apunta al objeto invocador, así,

Node a, b;

...

if (a != b) ...

resultaría en una llamada al operador *operator !=()* con this señalando a la dirección de *a*. Desreferenciamos this por medio del operador de desreferencia estándar "\*". Ahora, \*this es un objeto de clase *Node* el cuál es comparado con otro objeto, usando el operador *operator ==()*. Por consecuencia, se usa la definición del operador *operator ==()* de la clase *Node*. Al usar el operador booleano estándar NOT " !", negamos el resultado y obtenemos el verdadero valor de *operator !=()*.

Los métodos de arriba deberían estar disponibles para cada clase que definas. Esto asegura que tu puedas usar tus objetos tal como usarías cualesquiera otros objetos, por ejemplo integers. Si alguno de estos métodos no tienen mucho sentido por alguna razón, tú deberías declararlos en una sección privada de la clase para explícitamente marcarlos como no de uso público. De otro modo, el compilador de C++ sustituiría los operadores estándar.

Obviamente, las aplicaciones reales requieren que los nodos lleven datos. Como se mencionó arriba, esto significa especializar los nodos. Los datos pueden ser de cualquier tipo, de ahí que estemos usando la construcción de plantilla.

template <class T>

class DataNode : public Node {

T \_data;

public:

DataNode(const T data, DataNode \*right = NULL) :

Node(right), \_data(data) {}

DataNode(const DataNode &val) :

Node(val), \_data(val.\_data) {}

const DataNode \*right() const {

return((DataNode \*) Node::right());

}

DataNode \*&right() { return((DataNode \*&) Node::right()); }

const T &data() const { return \_data; }

T &data() { return \_data; }

DataNode &operator =(const DataNode &val) {

Node::operator =(val);

\_data = val.\_data;

return \*this;

}

const int operator ==(const DataNode &val) const {

return(

Node::operator ==(val) &&

\_data == val.\_data);

}

const int operator !=(const DataNode &val) const {

return !(\*this == val);

}

};

La plantilla *DataNode* de arriba simplemente especializa la clase *Node* para que transporte datos de cualquier tipo. Añade funcionalidad para accesar su elemento de datos y también ofrece el mismo conjunto de funcionalidad estándar : *Copy Constructor*, *operator =()* and *operator ==()*. Nótese como reutilizamos funcionalidades ya definidas por la clase *Node*.

**10.4.2 Plantillas para la Lista**

  Ahora ya podemos declarar la plantilla para la lista. Nosotros usamos aquí también el mecanismo de plantilla, debido a que queremos que la lista lleve datos de tipo arbitrario. Por ejemplo, queremos poder definir listas de integers. Empezamos con una plantilla de clase abstracta *ListBase* la cuál funciona como la clase de base para todas las otras listas. Por ejemplo, las listas doblemente ligadas obviamente comparten las mismas propiedades como lo hacen las listas ligadas sencillas.

template <class T>

class ListBase {

public:

virtual ~ListBase() {} // Fuerza al destructor a que sea

// virtual

virtual void flush() = 0;

virtual void putInFront(const T data) = 0;

virtual void append(const T data) = 0;

virtual void delFromFront() = 0;

virtual const T &getFirst() const = 0;

virtual T &getFirst() = 0;

virtual const T &getLast() const = 0;

virtual T &getLast() = 0;

virtual const int isEmpty() const = 0;

};

Lo que realmente hacemos es describir la interface para cada lista especificando los prototipos de los métodos requeridos. Hacemos éso para cada operación que hemos identificado en la sección [10.3](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm#l9properties). Adicionalmente, incluímos también un método *flush()* (purgar) que nos permite eliminar todos los elementos de una lista.

Para las operaciones *get-first* (obtener el primero) y *get-last* (obtener el último) hemos declarado dos versiones. Una para uso en un contexto constante y otro para un contexto variable.

Con esta plantilla para clases abstractas, nosotros somos capaces de realmente definir nuestra plantilla de clase para lista :

template <class T>

class List : public ListBase<T> {

DataNode<T> \*\_head, \*\_tail;

public:

List() : \_head(NULL), \_tail(NULL) {}

List(const List &val) : \_head(NULL), \_tail(NULL) {

\*this = val;

}

virtual ~List() { flush(); }

virtual void flush();

virtual void putInFront(const T data);

virtual void append(const T data);

virtual void delFromFront();

virtual const T &getFirst() const { return \_head->data(); }

virtual T &getFirst() { return \_head->data(); }

virtual const T &getLast() const { return \_tail->data(); }

virtual T &getLast() { return \_tail->data(); }

virtual const int isEmpty() const { return \_head == NULL; }

List &operator =(const List &val) {

flush();

DataNode<T> \*walkp = val.\_head;

while (walkp) append(walkp->data());

return \*this;

}

const int operator ==(const List &val) const {

if (isEmpty() && val.isEmpty()) return 1;

DataNode<T> \*thisp = \_head,

\*valp = val.\_head;

while (thisp && valp) {

if (thisp->data() != valp->data()) return 0;

thisp = thisp->right();

valp = valp->right();

}

return 1;

}

const int operator !=(const List &val) const {

return !(\*this == val);

}

friend class ListIterator<T>;

};

Los constructores inicializan los elementos *\_head* and *\_tail* de la lista a NULL que es el apuntador NULO en C y C++. Deberías saber como implementar los otros métodos a partir de tu experiencia en programación. Aquí solamente presentamos la implementación del método *putInFront()*:

template <class T> void

List<T>::putInFront(const T data) {

\_head = new DataNode<T>(data, \_head);

if (!\_tail) \_tail = \_head;

} /\* putInFront \*/

Si definimos métodos de una plantilla de clase afuera de su declaración, también debemos especificar la palabra clave template. Nuevamente usamos el operador new para crear dinámicamente un nuevo nodo de datos. Este operador permite la inicialización de su objeto creado con argumentos encerrados entre paréntesis. En el ejemplo de arriba, new crea una nueva instancia de la clase *DataNode$<$T$\gt$*. Consecuentemente, es llamado el constructor correspondiente.

Nótese también como usamos el sustituto *T*. Si nosotros creáramos una instancia de la plantilla de clase *List*, digamos, *List$<$int$\gt$* esto causaría también la creación de una instancia de clase de la plantilla de clase *DataNode*, a saber *DataNode$<$int$\gt$*.

La última línea de la declaración de la plantilla de clase declara la plantilla de clase *ListIterator* para que sea amiga de *List*. Nosotros queremos definir separadamente el iterador de la lista. Sin embargo, está my cercanamente relacionado, así que le permitimos que sea un amigo.

**10.5 Implementación del Iterador**

En la sección [10.2](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm#l9shapeandtraversal) presentamos el concepto de iteradores para recorrer una estructura de datos. Los iteradores deben implementar tres propiedades :

* *Elemento actual*.
* *Función sucesor*.
* *Condición de terminación*.

En términos generales, el iterador sucesivamente regresa datos asociados con el elemento actual. Obviamente, habrá un método, digamos *current()* que implemente esta funcionalidad. El tipo que regrese este método depende del tipo de datos almacenados en la estructura de datos particular. Por ejemplo, cuando se itera sobre *List$<$int$\gt$* el tipo regresado debería ser int.

La función sucesor, digamos *succ()*, utiliza información adicional que está almacenada en los elementos estructurales de la estructura de datos. En nuestro ejemplo de lista, dichos elementos son los nodos los que llevan los datos *y* un apuntador a su vecino de la derecha. El tipo de elementos estructurales usualmente difiere de aquél de los datos en bruto. Considera nuevamente nuestra *List$<$int$\gt$* donde *succ()* debe usar *DataNode$<$int$\gt$* como elementos estructurales.

La condición de terminación es implementada por un método, digamos *terminate()*, el cuál regresa *TRUE* si (y solamente si) todos los elementos de datos de la estructura de datos asociada han sido visitados. Mientras *succ()* pueda encontrar un elemento aún no visitado, este método regresa *FALSE*.

Nuevamente, nosotros queremos especificar una clase de iterador abstracto que defina las propiedades de cada iterador. Los pensamientos de arriba conducen a la siguiente declaración :

template <class Data, class Element>

class Iterator {

protected:

Element \_start,

\_current;

public:

Iterator(const Element start) :

\_start(start), \_current(start) {}

Iterator(const Iterator &val) :

\_start(val.\_start), \_current(val.\_current) {}

virtual ~Iterator() {}

virtual const Data current() const = 0;

virtual void succ() = 0;

virtual const int terminate() const = 0;

virtual void rewind() { \_current = \_start; }

Iterator &operator =(const Iterator &val) {

\_start = val.\_start;

\_current = val.\_current;

return \*this;

}

const int operator ==(const Iterator &val) const {

return(\_start == val.\_start && \_current == val.\_current);

}

const int operator !=(const Iterator &val) const {

return !(\*this == val);

}

};

Nuevamente, nosotros usamos el mecanismo de plantilla para permitir el uso del iterador para cualquier estructura de datos que almacene datos del tipo *Data* y que use elementos estructurales del tipo *Element*. Cada iterador "conoce" un elemento inicial (estructural) y el elemento actual. Hacemos accesibles a ambos desde las clases derivadas porque los iteradores derivados necesitan accesarlos para implementar las siguientes propiedades de un iterador. Ya deberías comprender como operan los constructores y por qué forzamos al destructor a que sea virtual.

Subsecuentemente especificamos tres métodos que deberían implementar las tres propiedades de un iterador. Agregamos también un método *rewind()* (rebobinar) que simplemente pone el elemento actual al inicio de los elementos. Sin embargo, las estructuras de datos complejas (por ejemplo las tablas hash) pudieran requerir algoritmos de rebobinado más sofisticados. Por esa razón, también especificamos este método como virtual, permitiendo a los iteradores derivados que lo redefinan para su estructura de datos asociada.

El último paso en el proceso de implementación del iterador es la declaración del iterador de la lista. Este iterador está altamente relacionado con nuestra plantilla de la clase *List*, por ejemplo, está claro que los elementos estructurales son plantillas de la clase *DataNode*. El único tipo "abierto" es el de los datos. Una vez más, usamos el mecanismo de plantilla para proveer iteradores de lista para los diferentes tipos de listas :

template <class T>

class ListIterator : public Iterator<T, DataNode<T> \*> {

public:

ListIterator(const List<T> &list) :

Iterator<T, DataNode<T> \*>(list.\_head) {}

ListIterator(const ListIterator &val) :

Iterator<T, DataNode<T> \*>(val) {}

virtual const T current() const { return \_current->data(); }

virtual void succ() { \_current = \_current->right(); }

virtual const int terminate() const {

return \_current == NULL;

}

T &operator ++(int) {

T &tmp = \_current->data();

succ();

return tmp;

}

ListIterator &operator =(const ListIterator &val) {

Iterator<T, DataNode<T> \*>::operator =(val);

return \*this;

}

};

La plantilla de clase *ListIterator* se deriva de *Iterator*. El tipo de datos es, por supuesto, el tipo para el cuál el iterador de la lista es declarado, de ahí que nosotros insertamos el sustituto *T* para el tipo de datos *Data* del iterador. El proceso de iteración se logra con la ayuda de los elementos estructurales del tipo *DataNode*. Obviamente, el elemento inicial es la cabeza de la *\_head* la cuál es del tipo *DataNode$<$T$\gt$ \**. Escogemos este tipo para el tipo del elemento *Element*.

Nótese que el iterador de la lista en la práctica esconde los detalles acerca de los elementos estructurales. Este tipo depende grandemente de la implementación de la lista. Por ejemplo, si nosotros hubiéramos escogido una implementación de arreglo, podríamos haber usado integers como elementos estructurales donde el elemento actual sería señalado por un índice de arreglo.

El primer constructor lleva la lista a ser recorrida como su argumento e inicializa su porción del iterador en concordancia. Como cada *ListIterator$<$T$\gt$* es un amigo de *List$<$T$\gt$* tiene acceso a los miembros privados de la lista. Usamos esto para inicializar el iterador a que apunte a la cabeza de la lista.

Omitimos el destructor porque no tenemos ningún miembro de datos adicional para el iterador de la lista. Consecuentemente, no hacemos nada especial para éso. Sin embargo, el destructor de la plantilla de clase *Iterator* es llamado. Recuerad que tenemos que definir este destructor para forzar a las clases derivadas a que también tengan uno virtual.

Los métodos siguientes solamente definen las tres propiedades requeridas. Ahora que tenemos elementos estructurales definidos *DataNode$<$T$\gt$ \** los usamos de la siguiente manera:

* el elemento actual es el dato llevado por el elemento estructural actual,
* la función sucesor es para que el elemento estructural actual se ponga en el vecino de la derecha y the successor function is to set the current structural element to its right neighbour and
* la condición de terminación es para checar si el elemento estructural actual es el apuntador NULL. Nótese que esto solo puede suceder en dos casos :

1.

La lista está vacía. En este caso, el elemento actual ya es NULL porque el miembro *\_head* de la cabeza de la lista es NULL.

2.

El elemento actual llegó al último elemento. En este caso la anterior llamada a la función sucesor pone el elemento actual en el vecino de la derecha del último elemento, el cuál es NULL.

Hemos incluído también un operador de postincremento "++" sobrecargado. Para distinguir este operador del operador de preincremento, lleva un argumento integer (anónimo) adicional. Como nosotros solamente usamos este argumento para declarar un prototipo de operador correcto, y porque nosotros no usamos el valor del argumento, omitimos el nombre del argumento.

El último método es el operador de asignación sobrecargado para iteradores de lista. En forma similar a los anteriores operadores de asignación, solamente reutilizamos las ya definidas asignaciones de las superclases ; *Iterator$<$T$\gt$::operator =()* en este caso.

Los otros métodos y operadores, a saber, *rewind()*, *operator ==()* y *operator !=()* son todos heredados de la plantilla de clase *Iterator*.

**10.6 Ejemplo de Uso**

  La plantilla de la lista tal como se presentó en las secciones anteriores, puede usarse del siguiente modo :

int

main() {

List<int> list;

int ix;

for (ix = 0; ix < 10; ix++) list.append(ix);

ListIterator<int> iter(list);

while (!iter.terminate()) {

printf("%d ", iter.current());

iter.succ();

}

puts("");

return 0;

}

Como hemos definido un operador de postincremento para el iterador de la lista, el bucle puede ser escrito también así :

while (!iter.terminate())

print("%d ", iter++);

**10.7 Discusión**

**10.7.1 Separación de la "Forma" y Estrategias de Acceso**

El ejemplo presentado se enfoca sobre una perspectiva orientada a objetos. En las aplicaciones reales las listas ligadas sencillas podrían ofrecer más funcionalidades. Por ejemplo, la inserción de ítems nuevos de datos no debería presentar ningún problema debido al uso de apuntadores :

1.

Toma el apuntador sucesor del nuevo elemento y haz que apunte al elemento que debería convertirse en su vecino de la derecha.

2.

Toma el apuntador sucesor del elemento después del cuál el nuevo elemento debería ser insertado y haz que apunte al nuevo elemento.

Dos operaciones simples. Sin embargo, el problema es designar el elemento después del cuál el nuevo elemento debería ser insertado. Nuevamente, se necesita un mecanismo para recorrer la lista. Esta vez, sin embargo, el recorrido se detiene en un elemento en particular : El elemento donde la lista (o la estructura de datos) es *modificada*.

En forma similar a la existencia de diferentes estrategias de recorrido, uno puede pensar en diferentes *estrategias de modificación*. Por ejemplo, para crear una lista ordenada, donde los elementos son ordenados en sentido ascendente, usa un *modificador ascendente*.

Estos modificadores deben tener acceso a los elementos estructurales de la lista, y así, deberían ser declarados amigos también. Esto llevaría a la necesidad de que *cada* modificador deba ser amigo de su estructura de datos. Pero ¿Quién puede garantizar que no sea olvidado ningún modificador ?

Una solución es que las estrategias de modificación no sean implementadas por clases "externas" como son los iteradores. En lugar de éso, son implementadas por herencia. Si se necesita una lista ordenada, se trata de una especialización de la lista general. Esta lista ordenada añadiría un método, digamos *insert()*, el cuál inserta un nuevo elemento de acuerdo a la estrategia de modificación.

Para hacer esto posible, la plantilla de lista presentada debe ser cambiada. Porque ahora, las clases derivdas deben tener acceso a los nodos cabeza y cola para implementar estas estrategias. Consecuentemente, *\_head* and *\_tail* deberían ser protected (protegidos).

**10.7.2 Iteradores**

La implementación del ieterador presentada asume que la estructura de datos no sea cambiada durante el uso del iterador. Considera el siguiente ejemplo para ilustrar esto :

List<int> ilist;

int ix;

for (ix = 1; ix < 10; ix++)

ilist.append(ix);

ListIterator<int> iter(ilist);

while (!iter.terminate()) {

printf("%d ", iter.current());

iter.succ();

}

printf("\n");

ilist.putInFront(0);

iter.rewind();

while (!iter.terminate()) {

printf("%d ", iter.current());

iter.succ();

}

printf("\n");

Este fragmento de código imprime

1 2 3 4 5 6 7 8 9

1 2 3 4 5 6 7 8 9

en lugar de

1 2 3 4 5 6 7 8 9

0 1 2 3 4 5 6 7 8 9

Esto es debido al hecho de que nuestro iterador de lista solamente almacena *apuntadores* a los elementos estructurales de la lista. Así, el elemento inicial *\_start* se pone en un principio a que señale la localización a la que apunta el nodo cabeza de la lista *\_head*. Esto conduce simplemente a que haya *dos* apuntadores diferentes referenciando la misma localización. Por consecuencia, cuando se cambia un apuntador, tal como sucede cuando se invoca *putInFront()*, el otro apuntador no es afectado.

Por esa razón, cuando se rebobina el iterador después *putInFront()* el elemento actual se pone al elemento inicial, el cuál fué establecido en el momento que el constructor del iterador fue llamado. Ahora, el elemento inicial de hecho está referenciando el *segundo* elemento de la lista.

**10.8 Ejercicios**

1.

En forma similar a la definición del operador de postincremento en la plantilla de clase *ListIterator*, se podría definir un operador de preincremento así :

T &operator ++() {

succ();

return \_current->data();

}

¿Qué problemas ocurren?

2.

Añade el siguiente método

int remove(const T &data);

a la plantilla de clase *List*. El método debería eliminar la primera occurrencia de *data* en la lista. El método debería regresar 1 si efectivamente eliminó un elemento o 0 (cero) si no lo hizo.

¿Qué funcionalidad debe proveer *data*? Recuerda que puede ser de cualquier tipo, ¡Especialmente clases definidas por el usuario!

3.

Deriva una plantilla de clase *CountedList* de *List* la cuál cuente sus elementos. Añade un método *count()* de tipo arbitrario que regrese el número actual de elementos almacenados en la lista. Trata de reutilizar de *List* tanto como sea posible.

4.

En relación al problema del iterador discutido en la sección [10.7](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node11.htm#Discussion). ¿Cuáles serían las posibles soluciones para permitir que la lista fuera alterada mientras un iterador de aquélla está en uso?

* [A.1 Una Revisión a las Técnicas de Programación](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html" \l "SECTION001310000000000000000)

* [A.2 Tipos de Datos Abstractos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html" \l "SECTION001320000000000000000)

* [A.3 Conceptos de Orientación a Objetos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html" \l "SECTION001330000000000000000)

* [A.4 Más Conceptos de Orientación a Objetos](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html" \l "SECTION001340000000000000000)

* [A.5 Más sobre C++](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html" \l "SECTION001350000000000000000)

* [A.6 La Lista - Estudio de un Caso](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html" \l "SECTION001360000000000000000)

**A Soluciones a los Ejercicios**

  Esta sección presenta ejemplos de solución a los ejercicios de las lecciones anteriores.

**A.1 Una Revisión a las Técnicas de Programación**

1.

Discusión del módulo *Singly-Linked-List-2*.

(a)

Definición de la Interface del módulo *Integer-List*

MODULE Integer-List

DECLARE TYPE int\_list\_handle\_t;

int\_list\_handle\_t int\_list\_create();

BOOL int\_list\_append(int\_list\_handle\_t this,

int data);

INTEGER int\_list\_getFirst(int\_list\_handle\_t this);

INTEGER int\_list\_getNext(int\_list\_handle\_t this);

BOOL int\_list\_isEmpty(int\_list\_handle\_t this);

END Integer-List;

Esta representación presenta problemas adicionales que son causados por no separar el recorrido, de la estructura de los datos. Como podrás recordar, para iterar sobre los elementos de la lista, hemos usado un bucle con la siguiente condición :

WHILE data IS VALID DO

*Data* se inicializó con una llamada a *list\_getFirst()*. El procedimieno de la lista integer *int\_list\_getFirst()* regresa un entero (integer), por consecuencia, no existe algo como un "integer no válido", lo cuál podríamos usar para un chequeo de terminación del bucle.

2.

Diferencias entre programación orientada a objetos y otras técnicas. En la programación orientada a objetos, los *objetos* intercambian *mensajes* entre sí. En otras técnicas de programación, *los datos* son intercambiados entre procedimientos bajo el control de un programa principal. Objetos del mismo tipo pero cada uno con su propio status pueden coexistir. Esto contrasta con el enfoque modular donde cada módulo tiene solamente un status global.

**A.2 Tipos de Datos Abstracto**

1.

TDA *Integer*.

(a)

Ambas operaciones *add* y *sub* pueden ser aplicadas para cualquier valor de *N*. Así, estas operaciones pueden ser aplicadas en cualquier momento: No hay restricción a su uso. Sin embargo, tú puedes describir ésto con una precondición que sea igual a *verdadero*.

(b)

Definimos aquí tres operaciones nuevas : *mul*, *div* y *abs*. Esta última debería regresar un valor absoluto del entero. Las operaciones se definen como sigue :

mul(k)

div(k)

abs()

La operación *mul* no requiere ninguna precondición. Esto es similar a *add* y *sub*. La postcondición es por supuesto *res* = *N*\**k*. La siguiente operació *div* requiere que *k* no valga 0 (cero). Consecuentemente, definimos la siguiente precondición : *k* no igual a 0. La última operación : *abs* regreas el valor de *N* si *N* es positivo o vale 0 o, -*N* si *N* es negativo. Nuevamente, no importa que valor tiene *N* cuando se aplica esta operación. Aquí está su postcondición :

if *N* >= 0 then

*abs* = *N*

else

*abs* = -*N*

2.

TDA *Fraction*.

(a)

Una fracción simple consiste de numerador y denominador. Ambos son números enteros. Esto es similar al ejemplo de los números complejos presentados en la sección. Podríamos escoger al menos dos estructuras de datos para contener los valores : un *arreglo* o un *registro*.

(b)

Esquema de la interface. Recuerda que la interface es solamente el conjunto de operaciones visible al mundo exterior. Podríamos describir una interface para una fracción en una manera verbal. Consecuentemente, necesitamos las operaciones :

* para *obtener (get)* el valor numerador/denominador,
* para *establecer (set)* el valor numerador/denominador,
* para *sumar* una fracción, regesando la suma,
* para *restar* una fracción regresando la diferencia,
* ...

(c)

Tenemos aquí algunos axiomas y precondiciones para cada fracción que valen también para el TDA :

 El denominador no debe ser igual a 0 (cero), de otro modo, el valor de la fracción no es definido.

 Si el numerador es igual a 0 (cero) el valor de la fracción es 0 para cualquier valor del denominador.

 Todo número entero puede ser representado por una fracción cuyo numerador es el número y el denominador es 1.

3.

Los TDAs definen las propiedades de un conjunto de instancias. Proveen una vista abstracta de estas propiedades aportando un conjunto de operaciones que pueden ser aplicadas a dichas instancias. Es este conjunto de operaciones, la *interface*, el que define las propiedades de las instancias. El uso de un TDA está restringido por axiomas y precondiciones. Ambos definen las condiciones y las propiedades de un ambiente en el cuál instancias del TDA pueden ser usadas.

4.

Necesitamos declarar axiomas y definir precondiciones para asegurar el uso correcto de las instancias de los TDAs. Por ejemplo, si nosotros no declaramos que 0 sea un elemento neutral en la adición de enteros, podría haber un TDA *Integer* que hiciera algo extraño cuando se sumara 0 a *N*. Esto no es lo que se espera de un entero. Así, los axiomas y precondiciones proveen un medio para asegurarse que los TDAs "funcionen" tal como queremos que lo hagan.

5.

Descripción de relaciones.

(a)

Una instancia es una representación funcional de un TDA. Es por lo tanto un "ejemplo" de él. Cuando un TDA declara usar un "número entero con signo" en su estructura de datos, una instancia contiene de hecho un valor, digamos, "-5".

(b)

Los TDAs genéricos definen las mismas propiedades de su TDA correspondiente. Sin embargo, están dedicados a otro tipo en particular. Por ejemplo, el TDA *List* define propiedades de listas. Así, podríamos tener una operación *append(elem)* que agrega un nuevo elemento *elem* a la lista. No decimos que tipo *elem* es en realidad, solamente que será el último elemento de la lista después de esta operación. Si ahora usamos un TDA genérico *List*, el tipo de elemento es conocido : es provisto por el parámetro genérico.

(c)

Las instancias del mismo TDA genérico podrían ser vistos como "hermanos". Serían "primos" de instancias de otro TDA genérico si ambos TDAs genéricos comparten el mismo TDA.

**A.3 Conceptos de Orientación a Objetos**

1.

Clase.

(a)

Una *clase* es la implementación real de un TDA. Por ejemplo, un TDA para integers pudieran incluir la operación *set* para establecer el valor de su instancia. Esta operación se implementa en forma diferente en lenguajes tales como C o Pascal. En C, el signo igual "=" define el conjunto de operaciones para integers, mientras que en Pascal se usa la cadena de caracteres " :=". Consecuentemente, las clases implementan operaciones al proveer *métodos*. En forma similar, la estructura de datos del TDA es implementado por *atributos* de la clase.

(b)

Clase *Complex*

class Complex {

attributes:

Real real,

imaginario

methods:

:=(Complex c) /\* Poner el valor a lo que vale c \*/

Real realPart()

Real imaginaryPart()

Complex +(Complex c)

Complex -(Complex c)

Complex /(Complex c)

Complex \*(Complex c)

}

Escogemos los bien conocidos símbolos de operador "+" para la adición, "-" para la resta, "/" para la división y "\*" para la multiplicación al implementar las operaciones correspondientes del TDA *Complex*. Así, objetos de la clase *Complex* pueden ser usados del siguiente modo:

Complex c1, c2, c3

c3 := c1 + c2

Podrás notar, que podríamos escribir la instrucción de la adición como sigue :

c3 := c1.+(c2)

Podrías querer remplazar el "+" con "add" para aproximarse a una representación que ya hemos usado. Sin embargo, deberías poder entender que "+" no es otra cosa que un nombre diferente para "add".

2.

Objetos interactuantes.

3.

Perspectiva del objeto.

4.

Mensajes.

(a)

Los objetos son entidades autónomas que solamente proveen una bien definida interface. Quisiéramos hablar de objetos como si fueran entidades activas. Por ejemplo, los objetos "son responsables" de sí mismos, "ellos" podrían prohibir la invocación de un método, etc.. Esto distingue un objeto de un módulo, el cuál es pasivo. Por lo tanto, no hablamos de llamadas a porocedimientos. Hablamos de mensajes con los cuáles "le pedimos" a un objeto que invoque uno de sus métodos.

(b)

El internet provee varios objetos. Dos de los más conocidos son "cliente" y "servidor". Por ejemplo, tú usas un cliente (objeto) de FTP para acceder a datos almacenados en un servidor (objeto) de FTP. De ahí que, se podría ver ésto como si el cliente "mandara un mensaje" al servidor pidiéndole que le provea de datos almacenados ahí.

(c)

En el ambiente cliente/servidor, tenemos realmente dos entidades actuando en forma remota : los procesos de cliente y de servidor. Típicamente, estas dos entidades intercambian datos en forma de mensajes de Internet.

**A.4 Más Conceptos Orientados a Objetos**

1.

Herencia.

(a)

Definición de la clase *Rectángulo*:

class Rectangle inherits from Point {

attributes:

int \_width, // Base del rectángulo

\_height // Altura del rectángulo

methods:

setWidth(int newWidth)

getWidth()

setHeight(int newHeight)

getHeight()

}

En este ejemplo, definimos un rectángulo por su esquina superior izquierda (las coordenadas tal como se heredaron de *Point*) y sus dimensiones. Alternativamente, lo podríamos haber definido por su esquina superior izquierda y su esquina inferior derecha.

Añadimos métodos de acceso para la base y la altura del rectángulo.

(b)

Objetos en 3a. dimensión. Una esfera se define por un centro en un espacio en 3a. dimensión, y por un radio. El centro es un punto en un espacio en 3a. dimensión, así, podemos definir la clase *Sphere* (esfera) como:

class Sphere inherits from 3D-Point {

attributes:

int \_radius;

methods:

setRadius(int newRadius)

getRadius()

}

Esto es similar a la clase círculo para un espacio en 2a. dimensión. Ahora*3D-Point* es solamente un *Point* con una dimensión adicional:

class 3D-Point inherits from Point {

attributes:

int \_z;

methods:

setZ(int newZ);

getZ();

}

Consecuentemente, *3D-Point* y *Point* tienen una relación es-un(a).

(c)

Funcionalidad de *move()*.

*move()* como se definió en la sección le permite a los objetos de 3a. dimensión moverse a lo largo del eje-X, es decir, en una sola dimensión. Hace esto, al modificar solamente la parte de 2a. dimensión de los objetos de 3a. dimensión. Esta parte de 2a. dimensión está definida por la clase *Point* heredada directamente o indirectamente por los objetos de 3a. dimensión.

(d)

Gráfica de herencia (ver la Figura [A.1](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node13.html#solig)).

|  |
| --- |
| **Figura A.1:**  Gráfica de herencia de algunos objetos desplegables. |
| \begin{figure}  {\centerline{ \psfig {file=FIGS/solig.eps,width=9cm} }}  \end{figure} |

(e)

Gráfica de herencia alternativa. En este ejemplo, la clase *Sphere* hereda de *Circle* y simplemente añade una tercera coordenada. Esto tiene la ventaja de que una esfera puede ser manejada como un círculo (por ejemplo, su radio puede ser fácilmente modificado por métodos/funciones que manejan círculos). Tiene la desventaja de que "distribuye" el manejador (el centro en el espacio de 3a. dimensión) sobre la jerarquía de herencia : va desde *Point* pasando sobre *Circle* a *Sphere*. De ahí que este manejador no esté accesible como un todo.

2.

Herencia múltiple. La gráfica de herencia en la Figura [5.9](http://www.desy.de/gna/html/cc/Tutorial/Spanish/node6.html#l4multiple2) obviamente introduce conflictos de nomenclatura por las propiedades de la clase *A*.

Sin embargo, estas propiedades se identifican en forma única siguiendo la trayectoria hacia arriba desde *D* hasta *A*. Así, *D* puede cambiar las propiedades de *A* heredadas por *B* al seguir la trayectoria de herencia a través de *B*. En forma similar, *D* puede cambiar las propiedades de *A* heredadas por *C* al seguir la trayectoria de herencia a través de *C*. Consecuentemente, este conflicto de nomenclatura no necesariamente conlleva a error, mientras las trayectorias estén designadas.

**A.5 Más sobre C++**

1.

Polimorfismo. Cuando se usa la "signature"

void display(const DrawableObject obj);

nótese primero, que en C++ los parámetros de funciones o métodos son pasados por valor. Consecuentemente, *obj* sería una *copia* del argumento de llamada a función realmente provisto. Esto significa que *DrawableObject* debe ser una clase de la cuál se pueden crear objetos. Este **no** es el caso, si *DrawableObject* es una clase abstracta (tal como sucede cuando *print()* se define como un método puro.)

Si existe un método virtual *print()* que está definido por la clase *DrawableObject*, entonces (como *obj* es solamente una copia del argumento real) este método es invocado. Este **no** es el método definido por la clase del argumento real (¡debido a que ya no juega ningún rol significativo !)

**A.6 La Lista - Estudio de un Caso**

1.

Operador de preincremento para iteradores. El operador de preincremento tal como se define en el ejercicio no checa la validez de *\_current*. Como *succ()* pudiera poner su valor a *NULL* ésto pudiera causar acceso a este apuntador NULL y, de ahí, bloquear el programa. Una posible solución pudiera ser definir el operador como :

T &operator ++() {

succ();

return(\_current ? \_current->data() : (T) 0);

}

Sin embargo, esto no funciona como estamos asumiendo ahora algo sobre *T*. Debe ser posible convertirlo a un tipo de valor "NULL".

2.

Escritura del método remove. No damos la solución codificada. En lugar de eso, proporcionamos el algoritmo. El método *remove()* debe iterar sobre la lista hasta que alcance el elemento con el ítem de datos solicitado. Hecho lo cuál, elimina el elemento y regresa 1. Si la lista está vacía, o, si el ítem de datos no puede ser encontrado, regresa 0 (cero).

Durante la iteración, *remove()* debe comparar el ítem de datos provisto sucesivamente con aquéllos en la lista. Consecuentemente, podría existir una comparación como :

if (data == current->data()) {

// se encontró el ítem

}

Usamos aquí el operador de ecuación "= =" para comparar ambos ítemes de datos. Como estos ítemes pueden ser de cualquier tipo, pueden ser especialmente objetos de clases definidas por el usuario.   
La cuestión es : ¿Cómo se define la "igualdad" para esos tipos nuevos ? Consecuentemente, para permitir a *remove()* que trabaje adecuadamente, la lista debería ser usada solamente para tipos que propiamente definan los operadores de comparación (a saber, "= =" y " !="). De otro modo, se usan las comparaciones de "default", lo que llevaría a resultados extraños.

3.

La clase *CountedList*. Una lista contada es una lista que sigue la pista al número de elementos en ella. Así, cuando un ítem de datos es añadido, el número es incrementado en uno, cuando un ítem es eliminado, se decrementa en uno. Nuevamente, no proporcionamos la implementación completa, más bien mostramos un método (*append()*) y como el número es alterado:

class CountedList : public List {

int \_count; // El número de elementos

...

public:

...

virtual void append(const T data) {

\_count++; // lo incrementa y ...

List::append(data); // ... usa el append de la lista

}

...

}

No todos los métodos pueden ser implementados de esta manera. En algunos métodos, uno debe checar si *\_count* necesita ser alterado o no. Sin embargo, la idea principal es que cada método de la lista es solamente expandido (o *especializado*) para la lista contada.

4.

Problema del iterador. Para resolver el problema del iterador se podría pensar en una solución donde el iterador almacene una referencia a su lista correspondiente. En el momento de la creación del iterador, esta referencia es inicializada para referenciar la lista provista. Los métodos del iterador deben ser modificados para usar esta referencia en lugar del apuntador *\_start*.