FACULTAD DE INGENIERIA U.N.A.M.
DIVISION DE EDUCACION CONTINUA

CURSOS INSTITUCIONALES

PROGRAMACION ORIENTADA A OBJETOS CON C++

TELEINDUSTRIA ERICSSON, S.A. DE C.V.

DEL 9 AL 25 DE ENERO DE 1995

CONCEPTOS DEL DISEÑO ORIENTADO A OBJETOS

ING. EDWIN NAVARRO PLIEGO

PALACIO DE MINERIA

1995

Capítulo I

Conceptos del diseño orientado a objetos

La velocidad a la que avanza la tecnología de hardware de computadoras es sorprendente; año con año se logra avances que conducen a la construcción de computadoras más veloces, más compactas y más baratas. Sin embargo, en el aspecto de software no parece un desarrollo similar. Mientras los costos de hardware han disminuido continuamente, los software han hecho lo contrario. La construcción de software no es una tarea fácil y en muchas ocasiones los proyectos de programación sobre giran los presupuestos de tiempo y dinero.

1.1 El problema de mantenimiento de software.

La construcción de software ha recibido la atención de los expertos desde hace mucho tiempo; en la década de los 70's se consiguieron avances significativos hacia el desarrollo de metodologías de forma sistemática y a bajo costo. Como resultado de esos esfuerzos surgieron técnicas que, como el diseño estructurado y el desarrollo descendente (top-down), durante mucho tiempo ha sido las herramientas utilizadas por los programadores para construir software. Aunque dichas técnicas han sido empleadas durante mucho tiempo en proyectos realmente complejos, la mayoría de los ingenieros de software coinciden en afirmar que sufren de tres grandes deficiencias:

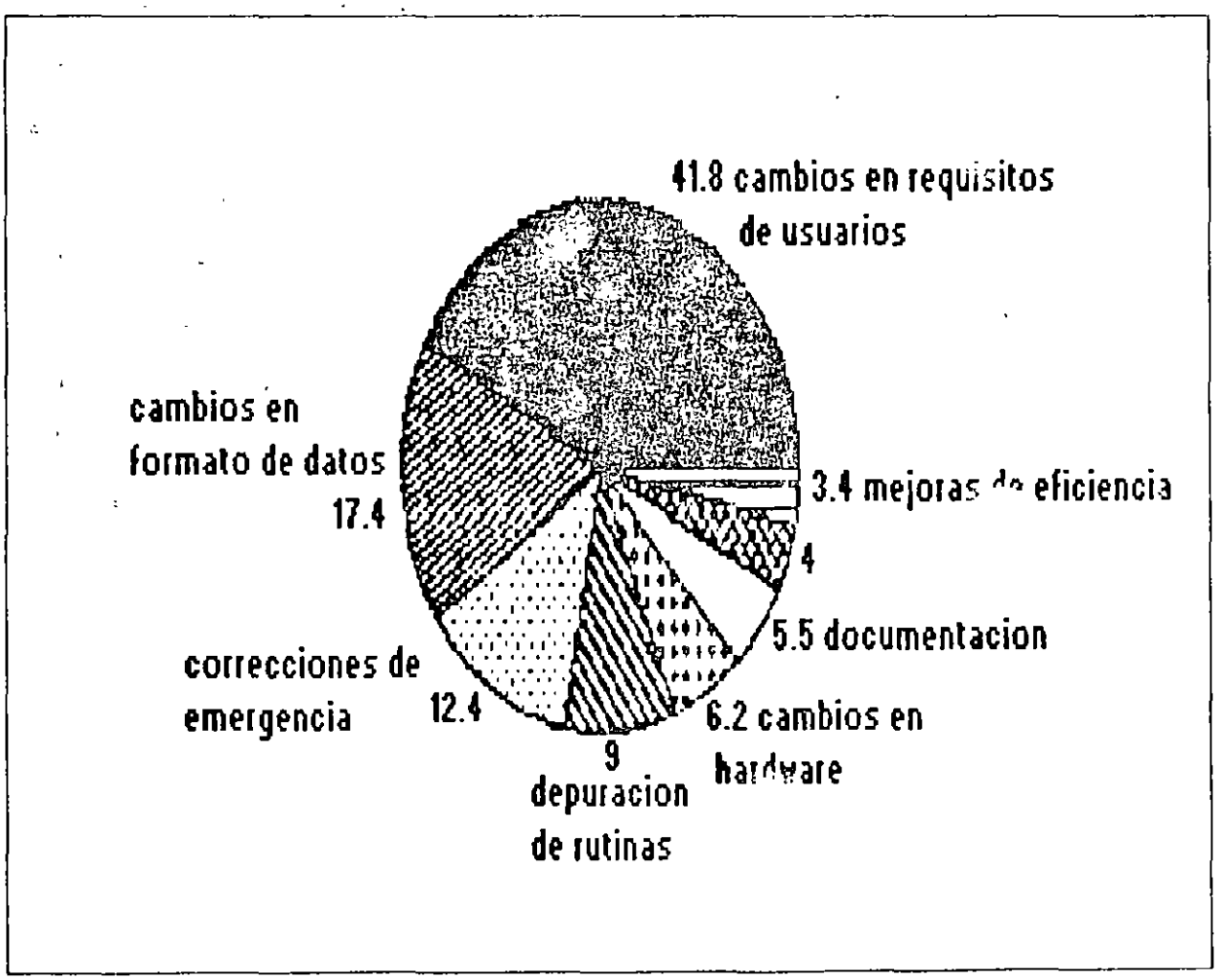
1 Los productos que resultan al emplear éstas técnicas son poco flexibles.

2 Los programadores que las usan tienden a concentrarse en el diseño y la implementación del sistema, sin tomar en cuenta su vida posterior.

3 No alientan al programador a aprovechar el trabajo de proyectos anteriores.

La desventaja de utilizar una metodología que se concentra en el diseño inicial del sistema se hace evidente si se toma en cuenta

que la vida útil de un producto de software puede ser cinco o seis veces más grande que el lapso en que se desarrolla; por ejemplo, un sistema que se desarrolla en uno o dos años puede mantenerse trabajando durante un período que va de cinco a quince años. Los gastos que se hacen durante éste último período (gastos de mantenimiento) representan alrededor del 70% del costo total del sistema. La siguiente ilustración muestra la forma en que se distribuyen éstos gastos.



La gráfica muestra que cerca del 60% de los costos de mantenimiento de un sistema (alrededor del 42% del costo total) se tiene que hacer por cambios en las especificaciones del usuario o en los formatos de los datos. Es por ello que la extensibilidad (la facilidad con que se modifica un sistema para que se realice nuevas funciones) debe ser uno de los objetivos primarios de la etapa de diseño.

La fase de mantenimiento es tan importante que cualquier método de diseño debe tener como objetivo principal producir sistemas que faciliten su propio mantenimiento. Pero, ¿Cómo alcanzar éste objetivo?. Los expertos recomiendan una serie de actividades y heurísticas que pueden servir como guía durante el desarrollo del sistema. La tabla I agrupa tales actividades de acuerdo a la etapa de desarrollo en que se deben realizar; más tarde nos preocuparemos con mayor cuidado de la *modularidad*, el *ocultamiento de la información* y la *abstracción de datos*.

Actividades de análisis
- Establecimiento de estándares (formatos de documentos, codificación, etc.).
- Especificar procedimientos de control de calidad.
- Identificar probables mejoras del producto.
- Estimar recursos y costos de mantenimiento.

Actividades de diseño
- Establecer la claridad y modularidad como criterios de diseño.
- Diseñar para facilitar probables mejoras.
- Usar notaciones estandarizadas para la documentación, algoritmos, etc.
- Seguir los principios de ocultamiento de información, abstracción de datos y descomposición jerárquica de arriba hacia abajo.
- Especificar efectos colaterales de cada módulo.

Actividades de implementación
- Usar estructuras de una sola entrada y una sola salida.
- Usar sangrado estándar en las diferentes estructuras.
- Usar un estilo de codificación simple y claro.
- Usar constantes simbólicas para asignar parámetros a las rutinas.
- Proporcionar prólogos estándares de documentación en cada módulo.

Otras actividades
- Desarrollar una guía de mantenimiento.
- Desarrollar un juego de pruebas.
- Proporcionar la documentación del juego de pruebas.

Tabla 1. Actividades que facilitan el mantenimiento de un sistema

1.2 Reutilización de código.

La *reutilización de código* es otro de los factores que no se toma en cuenta en los métodos de diseño tradicionales. Cualquier programador que desarrolla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, entre otras, se repiten continuamente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente por él mismo

o por otros programadores.

Tradicionalmente se han empleado tres tipos de reutilización: de personal, de diseño y de código fuente.

En la primera, a un proyecto nuevo se le asignan programadores que tienen experiencia en el desarrollo de proyectos semejantes.

La segunda consiste en emplear el diseño de un sistema para desarrollar otro sistema similar.

La tercera y última forma de reutilización se da cuando un programador utiliza parte un programa escrito con anterioridad para crear un nuevo programa. Sin embargo, éstas tres formas de reutilización se han empleado en forma muy limitada, por lo que es necesario buscar técnicas más generales.

1.3 Modularidad.

La modularidad es una de las herramientas de diseño más poderosas para facilitar el desarrollo y mantenimiento de sistemas de software. La modularidad permite definir un sistema complejo en términos de unidades más pequeñas y manejables; cada una de esas unidades (ó *módulos*) se encarga de manejar un aspecto local de todo el sistema, interactuando con otros módulos para cumplir con el objetivo global.

La mayoría de los lenguajes de programación actuales alientan el uso de la modularidad: en lenguajes estructurados como C ó Pascal, la modularidad se basa en el concepto de función (también llamada procedimiento o subrutina); en lenguajes más evolucionados, como Ada o Modula-2, los módulos corresponden a conjuntos de funciones relacionados con las estructuras de datos que manipulan esas funciones (paquetes, en la terminología de Ada).

Sin embargo, para ser realmente útil, el concepto de módulo debe ser aún más sofisticado. Según Meyer, las propiedades de un módulo deberían ser las siguientes:

a) *Decomponibilidad*: Un método de diseño modular debe permitir descomponer un problema de diseño en subproblemas más pequeños que pueden resolverse en forma independiente.

b) *Componibilidad*: Una vez que se cuente con un conjunto de módulos que realizan una función específica, se debe alentar al programador a usar esos m para construir nuevos programas.

c) *Comprensibilidad*: El lector de un programa o librería debe ser capaz de entender el funcionamiento de cada módulo sin necesidad de consultar el texto de otros módulos.

d) *Continuidad*: Un cambio pequeño en las especificaciones de un

programa debe causar cambios en un sólo módulo ó en un conjunto pequeño de ellos.

e) *Protección*: Un error de ejecución en el funcionamiento de un módulo no debe expandirse hacia los demás módulos.

Estas cinco propiedades pueden alentarse con las siguientes estrategias:

a) Un módulo debe corresponder con una unidad sintáctica del lenguaje (una subrutina, un paquete, una *clase*); esta unidad debe poder ser compilada por separado, tal vez para ser almacenada en una librería (con esto se mejoran la componibilidad y comprensibilidad del sistema).

b) Los módulos deben tener pocas interfases (medios de comunicación con otros módulos), y éstas deben ser pequeñas. Un número pequeño de interfases aumenta la independencia de los módulos, lo cual hace más fácil el proceso de componer nuevos sistemas a partir de módulos prefabricados, ayuda a evitar que los errores en un módulo se propaguen por todo el sistema y hace que cada módulo de un sistema sea más comprensible.

c) Cada módulo debe ocultar su implementación y algoritmos

internos al resto del sistema (principio de ocultamiento de información). No se debe permitir que un módulo modifique los elementos internos de otros módulos; sino que la comunicación entre ellos debe realizarse mediante interfases explícitas y bien definidas (esto favorece la comprensibilidad y la protección modular).

1.4 La programación orientada a objetos.

La programación orientada a los objetos (OOP) es un método de diseño que tiene como objetivo establecer técnicas de desarrollo de software para producir sistemas modulares. Un sistema orientado a objetos se puede entender fácilmente, por lo que su desarrollo, depuración y mantenimiento se facilitan en gran medida.

En la primera instancia, la OOP se basa en una idea relativamente sencilla: un programa de computadora es un modelo que representa un subconjunto del mundo real; la estructura de ese programa simplifica en gran medida, si cada una de las entidades (u objetos) del problema que se esta modelando corresponde directamente con un objeto que se pueda manipular internamente en el programa. Un ejemplo sencillo puede ser un programa de nómina: cualquier empleado de cierta empresa constituye una entidad real que tal vez represente dentro de un programa con un conjunto de variables o con una estructura (o registro).

El proceso de representar entidades reales con elementos internos a un programa recibe el nombre de *abstracción de datos*. La abstracción de datos no se concentra en la representación interna de un objeto (un entero, una cadena de bits, un arreglo), sino en sus atributos (con el nombre, sueldo y edad de un empleado) y en las operaciones que se pueda realizar sobre ese objeto (como

calcular el sueldo de un empleado o los impuestos que debe pagar). De esta forma, un tipo de dato abstracto se pude describir concentrándose en las operaciones que manipulan a los objetos de ese tipo, sin caer en los detalles de representación y manipulación de los datos.

La abstracción de datos es un concepto común en los lenguajes de programación moderna. Muchos lenguajes proporcionan un tipo de datos para números de punto flotante; cuando un programador utiliza datos de ese tipo, puede hacer operaciones como suma, multiplicación y exponenciación con esos datos, pero no es necesario que se preocupe por la representación de los números (p.e. cuatro palabras de máquina en las que se almacenan la mantisa y el exponente, junto con sus signos) ni por la forma en que el procesador realiza las operaciones (suma de enteros, comparación, desplazamiento lógico, etc.).

Sin embargo, la mayoría de los programas, los objetos involucrados son mucho más complejos que un número de punto flotante. La POO trata de describir tipos de datos de más alto nivel, por ejemplo: listas, ventanas, menús, figuras geométricas, procesos industriales, etc.. Tomemos, por ejemplo, un tipo llamado STACK; el programador que defina variables de ese tipo (tal vez para evaluar una expresión matemática o para implementar el recorrido de un árbol) únicamente necesita tomar en cuenta la

propiedad LIFO (Last Input First Output) de los stacks, algunos atributos (p.e. stack_lleno, tamaño) y las operaciones que se pueden realizar sobre ellos (p.e. *push, pop, crea_stack*). Para ese programador no es necesario conocer la estructura de datos con la que se implementa el stack (p.e. un arreglo o una lista ligada) ni los algoritmos con los que se implementan dichas operaciones.

La creación de tipos abstractos permite al programador adaptar el lenguaje de programación a sus necesidades específicas; el usuario de un lenguaje debe ser capaz de definir tipos de datos que se ajusten al problema que esta resolviendo y que puedan ser manipulados como los tipos internos del lenguaje. Cuando el programador cuenta con esta facilidad, se puede concentrar en los objetos que manipula su sistema y las relaciones entre esos objetos, haciendo a un lado los detalles de representación de los datos, para lograr una mejor comprensión del problema.

En la terminología del diseño orientado a objetos, un tipo abstracto es llamado *clase*. La OOP modela al mundo real utilizando objetos (instancias de una clase); el centro de atención son los datos. Este enfoque resulta particularmente exitoso porque durante el ciclo de vida de un sistema, los datos que se manipulan sufren pocos cambios, mientras que las acciones que se deben realizar sobre esos datos cambian constantemente.

Una clase describe un conjunto de objetos semejantes. Dicha descripción se hace en dos partes; los datos que especifican las propiedades de los objetos de esa clase (llamados *atributos*) y las funciones que manipulan esos datos (llamados *métodos*). Un objeto puede recibir mensajes de otros objetos; entonces debe escoger uno de sus métodos y dar una respuesta basándose en los datos que representan su estado (los atributos pueden ser modificados por los métodos).

Cada objeto cuenta con *elementos privados*, que sólo pueden ser usados por objetos de su misma clase, y *elementos públicos*, a los que tiene acceso cualquier otra entidad. Los elementos públicos representan la *interfase* de un objeto con su medio ambiente; únicamente esos elementos pueden modificar a los datos privados (los cuales representan el estado del objeto). Observe que este esquema se ajusta al principio de ocultamiento de información.

La programación orientada a objetos propone dos estrategias para la reutilización de código: la **composición** y la **herencia**. La *composición* permite definir una nueva clase de objetos mediante la unión de un conjunto de clases ya existentes. Por ejemplo, una clase que permita definir objetos que simulen procesos industriales puede necesitar de una forma de medir el tiempo, que tal vez sea proporcionada por una clase llamada *cronómetro*.

Por otro lado, la *herencia* permite crear (derivar) una nueva clase basándose en otra clase más general. Una *clase derivada* adquiere todas las propiedades y métodos de la clase de la que se derivó (*clase base*). De esta forma, puede ser posible derivar una clase **pentágono** a partir de un clase **polígono**, de tal forma que la primera adquiera todos los atributos (color, centro, relleno, etc.) y métodos (dibujar, borrar, escalar, etc.) de la segunda, tal vez añadiendo nuevos elementos o modificando ligeramente los ya existentes. El ejemplo anterior se basa en la llamada *herencia sencilla*; la *herencia múltiple* proporciona un método para derivar una clase de un conjunto de ellas, por ejemplo: un sistema de ventanas puede obtenerse de las clase *stack*, *ventana* y *editor*. Los procesos de composición y de herencia múltiple son muy parecidos, más adelante se tratará éste aspecto.

Además de facilitar la reutilización de código, la herencia es el medio ideal para crear sistemas con una alta extensibilidad. Otra ventaja de ésta técnica es que permite manipular objetos de clases diferentes como si fueran de la misma clase (**polimorfismo**), con lo cual es posible definir interfases uniformes para diferentes tipos de objetos.

1.5 Lenguajes orientados a objetos.

Las técnicas en las que se basa la programación orientada a objetos (como ya mencionamos; ocultamiento de la información, abstracción de datos, manejo automático de memoria, polimorfismo) eran conocidas y utilizadas por los ingenieros de software desde hace muchos años; lenguajes como Ada ó Modula-2 alientan a los programadores a usar algunas de esas técnicas.

Sin embargo, son pocos los lenguajes que brindan todas las facilidades para escribir programas orientados a objetos. Existen diversas opiniones acerca de cuáles deben ser dichas facilidades; para Bertrand Meyer (autor de Eiffel, uno de los lenguajes orientados a objetos más populares) deben ser las siguientes:

a) Estructura modular basada en objetos. Los sistemas se deben modularizar tomando como base sus estructuras de datos.

b) Abstracción de datos. El lenguaje debe permitir al programador definir tipos de datos abstractos.

c) Manejo de memoria automático. La memoria ocupada por objetos cuya utilidad ha terminado debe ser liberada por mecanismos internos al lenguaje, sin intervención del programador.

d) Clases. Cada tipo no simple es un módulo y cada módulo es un tipo.

e) Herencia. El lenguaje debe permitir definir clases como extensiones o restricciones de otras clases.

f) Polimorfismo y asociación dinámica de tipos. Las entidades internas de un programa deben poder manejar conjuntos objetos de diferentes clases de la misma forma en que manej conjuntos de objetos iguales. Una operación puede comportarse de varias formas de acuerdo a la clase de objeto que manipula.

g) Herencia múltiple. Una clase debe poder ser derivada de más de una clase.

Entre los lenguajes orientados a objetos más populares, se encuentran Simula67, un lenguaje de simulación con facilidades para manipular eventos discretos; Smalltalk, que se ha usado principalmente para desarrollar interfases de usuario gráficas. Eiffel se ha aplicado en áreas diversas. Los lenguajes orientados a objetos no habían tenido hasta recientemente una aceptación amplia entre la comunidad de programadores. Características como el manejo de memoria automático y la asociación dinámica de tipos imponen sobrecargas demasiado grandes a la ejecución de programas escritos en dichos lenguajes. Recientemente han surgido dos

estrategias para disminuir esa sobrecarga; una de ellas consiste en utilizar lenguajes orientados a objetos para desarrollar los componentes de más alto nivel de un sistema y lenguajes funcionales para escribir las partes de bajo nivel, críticas para la ejecución (una forma común de este tipo de combinaciones es utilizar Smalltalk y C). La otra estrategia consiste en desarrollar nuevos lenguajes que no proporcionen todas las facilidades de la programación orientada a objetos, pero que no impongan sobrecargas de ejecución demasiado altas (a estos lenguajes se les ha llamado *híbridos*); resultados de este enfoque son lenguajes como C Objetivo, C++ y algunas versiones de Pascal.

1.6 El lenguaje de programación C++.

C++ es una extensión orientada a objetos del lenguaje C. El objetivo principal de C++ es disminuir u ocultar la complejidad de cualquier proyecto de programación de tal forma que un sólo programador, o un grupo pequeño de ellos, pueda desarrollarlo y darle mantenimiento con poca dificultad.

El diseño de C++ se realizó a inicios de los 80's en los laboratorios Bell de la AT&T, dirigido principalmente por Bjarne Stroustrup. Stroustrup habla acerca de los orígenes del lenguaje:

"El nombre C++ es una invención reciente (verano de 1983). Desde 1980 se venían usando versiones previas del lenguaje, llamadas colectivamente 'C con clases'. El lenguaje se inventó originalmente por que el autor deseaba escribir cierta simulaciones manejadas por eventos, para lo cual Simula67 hubiera sido ideal, excepto por consideraciones de eficiencia. 'C con clases' se utilizó en proyectos de simulación más grandes en los que se debían usar tiempos y espacios mínimos. C++ se instaló por primera vez fuera del grupo de investigación del autor en Julio de 1983. El nombre simboliza la naturaleza evolutiva del lenguaje, '++' es el operador de incremento de C. El nombre C+ es un error de sintaxis, porque ya ha sido usado para otro lenguaje. El lenguaje no se llamó D porque es una extensión de C y no trata de remediar sus problemas quitando características".

C fue elegido como el lenguaje base de C++ por varias razones: C es un lenguaje de propósito general conocido por una gran cantidad de programadores; existen compiladores de C para un vasto conjunto de computadoras; C es un lenguaje expresivo y eficiente y, finalmente, se ha escrito una gran cantidad de código y librerías en C que no pueden ser despreciadas. La popularidad de C facilita el aprendizaje de C++, pues no es necesario aprender un nuevo lenguaje desde cero, sino que un programador puede ir aprendiendo nuevas características del lenguaje conforme las necesite. Como los dos lenguajes son compatibles, es posible utilizar todo el código

escrito en C desde C++; la compatibilidad tiene otra ventaja: se puede escribir un compilador de C++ en poco tiempo aprovechando el 'back end' de un compilador de C.

El diseño de C++ tuvo desde sus inicios el compromiso de conservar en la medida de lo posible la eficiencia a tiempo de corrida de C. Aunque la sobrecarga de un programa en C++ es mayor que la de su equivalente en C, todos los elementos del primero han sido cuidadosamente diseñados para minimizar esa diferencia (en programas grandes, el código en C++ tiende a ser más pequeño, además de que la diferencia de rendimiento es despreciable).

C++ es un lenguaje que ha evolucionado y sigue evolucionando rápidamente; los cambios en el lenguaje han surgido principalmente de problemas encontrados por usuarios o de innovaciones propuestas por el autor. La liberación 1.0 del lenguaje fue la especificada por Stroustrup en "El lenguaje de programación C++" (1986); más tarde se publicaron las liberaciones 1.1, y 1.2. La popularización del lenguaje se inició con la liberación 1.2, sin embrago, al crecer la población de programadores de C++ se hizo evidente la necesidad de nuevas mejoras. La siguiente versión de C++ (2.0) apareció en el verano de 1989 y es la que se encuentra vigente. Actualmente no existe una definición formal de C++, aunque se aceptan como estándares las versiones de AT&T, aunque se espera que para fines de 1992, la ANSI publique un estándar del lenguaje.

C++ se ha empleado en gran medida en ambiente UNIX. También existen compiladores que corren bajo otros sistemas operativos. En el ambiente MS-DOS, los compiladores más populares son Zortech, Glonckpesfield y Borland.

El lenguaje C++ se utiliza actualmente en el desarrollo de manejadores de bases de datos, sistemas operativos, compiladores, sistemas de comunicación, productos de CASE, redes y robótica.

Los programas de este texto fueron probados utilizando el compilador Turbo C++ ver 2.0 de Borland; aunque en ellos sólo se utilizaron características estándares, el lector debe estar consciente de que tal vez necesiten pequeños cambios para ejecutarlos utilizando otro compilador. Los mismo se aplica para la descripción del lenguaje que se hace aquí.

CAPITULO II

Programación funcional en C++

El lenguaje de programación C++ está diseñado para:

- Ser un mejor C
- Soportar abstracción de datos
- Soportar la programación orientada a objetos

En este capítulo se presentaran los mecanismos de programación que ofrece C, y apartir de aquí, se presentan los nuevos mecanismos que ofrece C++. El lenguaje de programación C es un subconjunto de C++, es decir, C++ es un mejor C porque C++ ofrece un mejor soporte para los nuevos estilos de programación que C no ofrece. C++ lo hace sin perder generalidad o eficiencia.

Debido a que el capítulo es muy extenso, este se presentará en tres secciones: *TIPOS, OPERADORES Y EXPRESIONES*, *CONTROL DE FLUJO* y *FUNCIONES*.

2.1 TIPOS, OPERADORES Y EXPRESIONES

Un programa es un modelo de un pequeño subconjunto del mundo real; en ese modelo los objetos externos se representan con cantidades que pueden ser manipuladas directamente por la computadora. La mayoría de los objetos modelados en un programa se mantienen en un cambio continuo, por lo que deben ser asociados con entidades cuyo valor pueda ser modificado por el programador, de acuerdo a las características de su modelo. En esta sección se analiza la forma de introducir variables en un programa, los tipos de datos que se pueden representar con ellas y algunos de los operadores que permiten manipular a esas variables.

Un programa complicado necesita tipos de datos de más alto nivel que los que se presentan aquí; C++ brinda al programador facilidades para definir sus propios tipos de datos y operadores sobre esos tipos, sin embargo, la discusión de esas facilidades se pospone hasta el capítulo 4.

2.1.1 Identificadores

Algunos identificadores se encuentran reservados para construir las frases propias del lenguaje; estos identificadores, llamados palabras reservadas. Por tanto, un identificador no puede ser una palabra reservada. La lista de palabras es la siguiente:

asm	auto	break	catch	case
cdeclr	char	class	const	continue
default	delete	do	double	else
enum	extern	float	for	friend
goto	if	inline	int	long
new	operator	overload	private	protected
public	register	return	short	signed
sizeof	static	struct	switch	template
this	typedef	union	unsigned	virtual
void	volatile	while		

El identificador puede estar formado por letras, dígitos y "_":

- El primer carácter debe ser una letra.

- El carácter "_" es utilizado como carácter de inicio de identificadores dentro de las rutinas de la biblioteca estándar.

Las letras minúsculas y mayúsculas son distintas.

Solamente los primeros 31 caracteres son significativos.

2.1.2 Tipos y tamaños de datos.

Un tipo de dato es un conjunto de valores y un conjunto de operaciones que se pueden realizar con ellos. Básicamente C++ sólo cuenta con dos tipos de datos: enteros y números de punto flotante. A partir de ellos es posible construir tipos más complejos como apuntadores, arreglos, estructuras, uniones y clases.

Existen tres grupos básicos de tipos en C++:

- Enteros
- Carácter
- De punto flotante

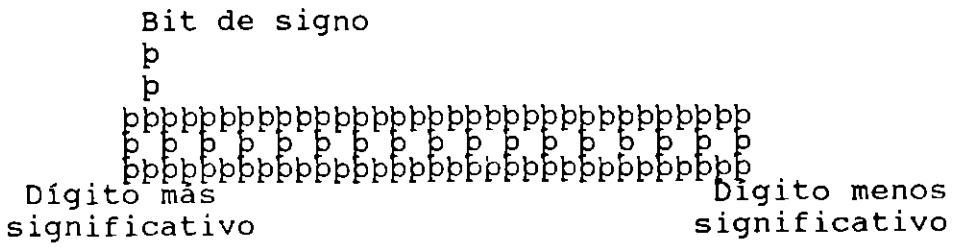
Enteros

El tamaño de los tipos enteros depende de la máquina.

Los tipos enteros signados son:

- short [int]
- int
- long [int]

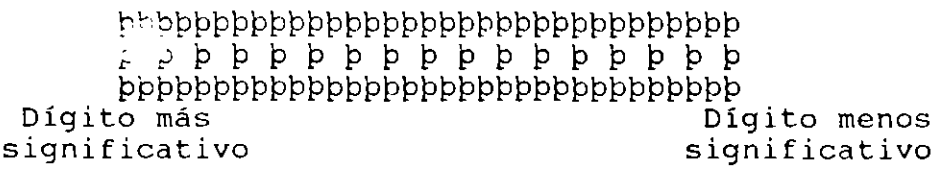
Representación:



Los enteros sin signo son:

- unsigned [short]
- unsigned [int]
- unsigned long [int]

Representación:



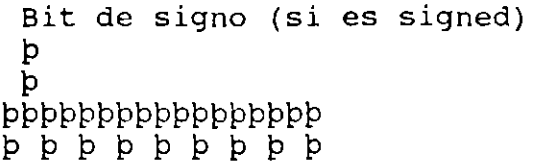
Carácter

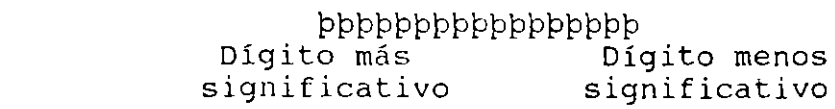
Un carácter es realmente un entero que contiene el código binario de ese carácter; por ejemplo, un carácter con valor 'A' es equivalente a otra con valor 65. Generalmente este tipo de variables ocupan un sólo byte de memoria.

Los tipos carácter son:

- [unsigned] char

Representación:





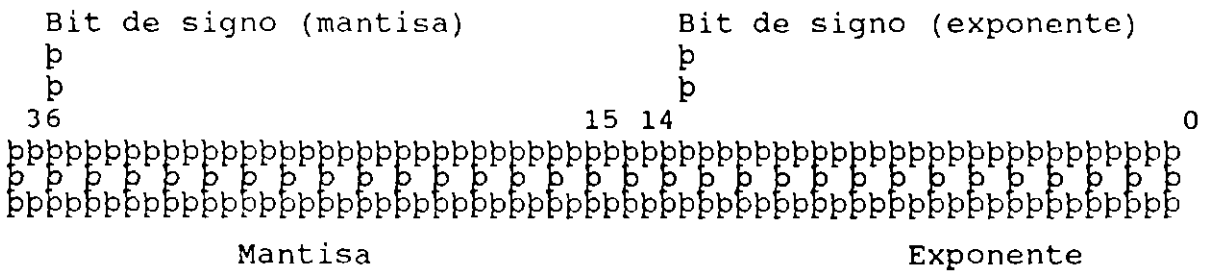
Sus valores son enteros.

Punto flotante

Los tipos de punto flotante son:

- float
- double
- long double

Representación:



Constantes

Si una declaración se califica con el modificador **const**, el identificador será una constante, no una variable (a este tipo de identificadores se les conoce como *constantes simbólicas*). El compilador no reserva espacio en memoria para las constantes simbólicas de los tipos básicos, sino que sustituye su valor en las expresiones en donde son empleadas (tal como lo hace el preprocesador de C con las macros). A diferencia de C, las constantes simbólicas pueden ser utilizadas en cualquier contexto en donde se necesite un valor fijo, por ejemplo para especificar la dimensión de un arreglo:

```
#include <stdio.h>
const int max = 100;
main(){
     //...
     char s[max];
     for (int i=0; i < max && (s[i]=getchar()) != '\n' ; i++ )
               ;
     s[i-1] = '\0';
     //...
}
```

Enteras:

- Decimal: 12, 125
- Octal: 007, 057
- Hexadecimal: 0xa95, 0xff23

De punto flotante:

- Pueden ser escritas como:

```
.0034
12.5
3e1
1.0E-3
```

Carácter:

- Se almacena el valor numérico del carácter.
- Pueden ser utilizadas en expresiones numéricas.
- Se escriben como: 'a', '+', '1'.
- Algunos caracteres se representan por más de un carácter:

```
'\n'      '\t'      '\f'
'\a'      '\b'
```

- También se pueden representar: '\033', '\0xff'

Enumerados

Los *enumerados* son otro tipo de constantes simbólicas. Un enumerado consiste de un serie de identificadores asociados con valores enteros. Una enumeración es una lista de valores enteros

constantes:

```
enum boolean = {FALSO,VERDADERO};

enum mes = { enero, febrero, marzo, /* .., */ diciembre };
```

El primer nombre en la lista de enumerados toma un valor de cero, el siguiente uno, y así sucesivamente. El nombre del enumerado (en este caso 'mes') se convierte en un sinónimo de **int**:

```
enum mes = { enero, febrero, marzo, /* .., */ diciembre };
const nmeses = 12;
main(){
     float sueldos[nmeses];
     //...
     float total = 0;
     for (mes i = enero; i <= diciembre ; i++)
          total += sueldos[i];
     //...
}
```

Se pueden cambiar los valores que toman los elementos de la lista:

```
enum letras { alpha, beta, gamma = 30, epsilon, zeta = 65 };
```

Los valores que toman son:

```
alpha = 0
beta = 1
gamma = 30
epsilon = 31
zeta = 65
```

Se pueden declarar variables de tipo enumerado, que serán manejadas como int y a las cuales se les puede asignar alguno de los valores de la lista:

```
enum boolean x;

x = zeta;
```

Los enumerados solamente son utilizados para propósitos de

documentación.

Tamaño de tipos de datos

El tamaño de los tipos de datos depende de la máquina y de la implementación del lenguaje. En el compilador Turbo C++ se utilizan los siguientes tamaños:

Tipo	Tamaño
char	8 bits
short	16 bits
int	16 bits
long	32 bits
float	32 bits
double	64 bits
long double	80 bits

El operador **sizeof** da como resultado el tamaño en bytes de su operando

```
sizeof( double ) == 8

sizeof( int ) == 2
```

por lo tanto, la tabla anterior se puede obtener en cualquier compilador con el siguiente programa (cout es un objeto que despliega un mensaje en la pantalla; su declaración se encuentra en iostream.h).

```
#include <iostream.h>
main() {
  cout << "El tipo char ocupa        " << sizeof(char)   << " bytes\n",
  cout << "El tipo int ocupa         " << sizeof(int)    << " bytes\n",
  cout << "El tipo long ocupa        " << sizeof(long)   << " bytes\n",
  cout << "El tipo short ocupa       " << sizeof(short)  << " bytes\n";
  cout << "El tipo float ocupa       " << sizeof(float)  << " bytes\n",
  cout << "El tipo double ocupa      " << sizeof(double) << " bytes\n",
  cout << "El tipo long double ocupa " << sizeof(long double) << " bytes\n";

}
```

2.1.3 Declaraciones y definiciones.

Todas las variables de un programa en C++ deben declararse antes de ser usadas. Una declaración consiste de un nombre de tipo seguido de una lista de identificadores, por ejemplo:

```
int i, j, k;
float dist, modulo, angulo;
```

Una declaración en C++ es una proposición, por lo que se puede colocar en casi cualquier parte dentro de un bloque. En lenguajes como C las declaraciones deben colocarse al inicio de un bloque y antes de cualquier proposición ejecutable. Lo que obliga al programador a declarar las variables mucho antes de que las use, dificultando la lectura del programa. En C++, la declaración de

cada variable se puede retrasar hasta justo antes de asignarle un valor:

```
#include <stdio.h>

main(){
     //...

     char s[10];
     for (int i = 0 ; i < 9 && (s[i]=getchar()) != '\n' ; i++)
                    ;
     s[i]='\0';
     //...
}
```

Las declaraciones están prohibidas en ciertos contextos, como en el siguiente ejemplo:

```
int y = 7;
//...
if ( y > 10 )
     int x = 7;    //error
```

Una definición es una declaración en la que se reserva memoria.

Las variables y las funciones deben ser declaradas antes de que sean usadas.

Las variables pueden ser inicializadas al momento de definirse:

```
main()
{
     int       r=2,i,j;
     float     pi=3.1415;
     char      car = 'a';

     //...
}
```

No es válido:

```
int  i = j = 0;
```

Al definir una variable se puede agregar el calificativo **const** para indicar que su valor no será cambiado. A diferencia de ANSI C, el compilador de C++ tomará una constante símbolica (const) como si se usara un *#define* (no reserva memoria), pero si detecta que se quiere accesar la dirección (p.e. una función puede necesitar una dirección o apuntador y no un valor) entonces la constante es tratada como una variable.

```
const  double  pi = 3.1415;
```

2.1.4 Conversiones de tipos.

Una expresión puede involucrar variables y constantes de diferentes tipos:

```
...
char      c;
int       i;
float     f;
double    d;

d = f * (i + c);

...
```

Reglas de conversión de tipos

En una expresión binaria si los operandos son de diferentes tipos el de menor grado es convertido al de mayor grado y el resultado de la expresión es del tipo de mayor grado.

La jerárquica de tipos de mayor a menor:

```
short,char,bit
unsigned int
int
long int
unsigned long int
float
double
long double
```

En el ejemplo anterior ¿De qué tipo es el resultado de la expresión asignado a la variable d?

Conversión de tipos en asignaciones

El tipo a la derecha del operador de asignación es convertido al tipo de la variable del lado izquierdo, de acuerdo a las reglas que se indican en la siguiente tabla:

Tipo =	Tipo	Resultado
float	int	Conversión exacta
int	float	Parte decimal es descartada; si el valor no esta en el rango, el resultado es desconocido.
int	char	Conversión exacta
char	int	Si el char es unsigned, los bits menos significativos son copiados; los restantes son descartados. Si el char es signed, depende de la implementación.
double	float	Conversión exacta
long double	float	Conversión exacta
float	double	Si el double esta fuera del rango de un double, el resultado es desconocido.

Conversión explícita de tipos (cast)

Cualquier conversión de tipo se puede forzar mediante el operador de conversión (*cast*) con cualquiera de las notaciones:

(tipo) expresión o *tipo (expresión)*

por ejemplo:

```
x = (int)c;                (char *) p

int (i * 3.14158)          float( i ) / float( 3 )

(int)(i * 3.14158)
```

2.1.5 Operadores

Para poder manejar los tipos de datos descritos con anterioridad, C++ utiliza operadores. Estos operadores son aritméticos, relacionales, lógicos y para manejo de bits.

Operadores aritméticos

Binarios: +, - , *, /, %

Unuarios: +, -

Precedencia: +, - (unuarios)
*, /, %
+, -

Asociatividad: izquierda a derecha

Operadores de relación, igualdad y lógicos

C no proporciona un tipo boolean:

- La evaluación de una expresión puede resultar en un valor 0 (falso) o diferente de cero (verdadero).

Los operadores de relación son los siguientes:

<	<=
>	>=

Los operadores de igualdad son los siguientes:

!=	==

Lo operadores lógicos son:

&& (and) || (or) ! (not)

La asociatividad de los operadores de relación, igualdad y lógicos

es de izquierda a derecha.

La precedencia de los operadores anteriores, de mayor a menor es la siguiente:

```
        !
<, <=, >, >=
   ==, !=
     &&
     ||
```

El operador ! (not) es unuario y cuando se evalúa con una expresión falsa 0 da como resultado 1; por otra parte cuando se evalúa con una expresión diferente de cero da por resultado 1, de esta forma, si x tiene un valor de 5 por ejemplo:

```
x != !(!x)
```

En una expresión que involucra operadores lógicos, cuando el resultado de la expresión se conoce, la evaluación termina.

Operadores de incremento y decremento

Los operadores de incremento y decremento son:

```
++    --
```

Pueden ser utilizados como prefijo o posfijo:

```
++x      x++
--x      x--
```

þ ++x incrementa x antes de utilizar su valor

þ x++ incrementa x después de utilizar su valor

Los operadores se pueden aplicar únicamente a variables; en particular

```
(i + j)++
```

no es una expresión válida.

```
/*  operc.cpp
Se compilo en Turbo C++ ver 2.0
*/
#include <stdio.h>

main() {
     int  a=0, b=0, c=0;

     a = ++b + ++c;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 2 1 1 */
     a = b++ + c++;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 2 2 2 */
     a = ++b + c++;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 5 3 3 */
     a = b-- + --c;
     printf("\n%d  %d  %d", a,b,c);  /* se imprime 5 2 2 */
     a = ++c + c;
     printf("\n%d  %d  %d", a,b,c);  /* depende de la maquina 6 2
3*/
     a = ++c + ++b;
     printf("\n%d  %d  %d", a,b,c);  /* depende de la maquina 7
4*/
}
```

Operadores de asignación

Existen dos tipos: simples y compuestos.

El operador de asignación simple es: =

El operador = asigna el valor de la derecha a la variable de la izquierda.

Se asocia de derecha a izquierda.

Cuando se lleva a cabo una asignación con =, el tipo del operando de la izquierda se convierte al de la derecha de acuerdo a las reglas de conversión vistas.

La asignación es una expresión, que da como resultado el valor y tipo del operando izquierdo, por lo tanto la siguiente operación es válida:

```
i = j = k = 0;
```

es equivalente a:

```
i = (j = (k = 0));
```

Para expresiones con el formato:

var = var operador expresión

donde: var = nombre de una variable

operador = alguno de los operadores:
+, -, *, /, %,
<<, >>, &, ^, |

expresión = cualquier expresión

Se pueden utilizar los operadores de asignación compuestas, para lo cuál la expresión anterior se puede transformar a:

var operador= expresión

No debe existir blanco entre operador y =

```
/*
        Este programa muestra el comportamiento de los operadores
        de asignación
*/

#include <stdio.h>

main(){
     int  a=12,b=5;

          a += b;   /* equivalente:  a = a + b  */
          a -= b;   /* equivalente:  a = a - b  */
          a *= b+5; /* equivalente:  a = a * (b+5) */
}
```

Operadores para manejo de bits

C++ permite al programador manipular los bits individuales de una variable utilizando los operadores & (and), | (or), ^ (xor), << (shitf left), >> (shift rigth) y ~ (complemento a uno). Haciendo uso de estos operadores es posible encender, apagar o invertir un grupo específicos de bits de un valor entero.

Estos operadores operan bit a bit en la variable o constante. La siguiente tabla muestra su comportamiento:

x	y	x & y	x ^ y	x | y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

Ejemplo (asumiendo que se tiene una representación de enteros de 2 bytes):

```
int i = 0xb765;

int j = i & 0xff00;
```

asigna el valor *0xb700* a la variable *j*, pues

```
          1011011101100101 == 0xb765 == 47949
     &    1111111100000000 == 0xff00 == 65280
     ppppppppppppppppppppppppppppppp
          1011011100000000 == 0xb700 == 46848
```

Operador de complemento a uno

El operador de complemento a uno ~, es un operador unuario.

Su comportamiento se muestra en la siguiente tabla:

x	~x
0	1

```
1          0
```

Ejemplo:

```
n = 499;        /* 0000 0001 1111 0011 */
x = 16;         /* 0000 0000 0001 0000 */
y = ~n;         /* 1111 1110 0000 1100 (-500) */
z = ~x;         /* 1111 1111 1110 1111 (-17)  */
```

Operadores de corrimiento de bits

Los operadores de corrimiento de bits son binarios y son: >> y <<.

En el caso de <<, se desplazan a la izquierda n bits indicados por el operador de la izquierda en el operador de la derecha:

- Los bits de exceso son descartados.
- Se colocan bits cero (0) en la derecha.

Ejemplo:

```
n = 16;         /* 0000 0000 0001 0000 */
c = n << 3;     /* 0000 0000 1000 0000  (128) */
```

En el caso de >>, se desplazan a la derecha n bits indicados por el operador de la izquierda en el operador de la derecha:

- Los bits de exceso son descartados.
- Los bits que entran por la izquierda son:

  para unsigned bit cero (0)
  para signed, depende de la implementación

Ejemplo:

```
n = 16;         /* 0000 0000 0001 0000  (16)*/
c = n >> 3;     /* 0000 0000 0000 0010  (2) */
```

en forma general:

x << n es equivalente a x * 2^n
x >> n es equivalente a x / 2^n

El operador condicional

El operador condicional de C++ se puede utilizar para evaluar expresiones que dependen de una condición. El valor de
expresión1 ? expresión2 : expresión3

será el resultado de expresión2 si expresión1 es diferente de cero ó el resultado de expresión3 si expresión1 es cero.

Ejemplo:

```
z = ( a > b ) ? a : b;
```

```
/* dump16.cpp
     Programa que muestra la representación binaria de un entero de
     dieciséis bits.
*/
#include <iostream.h>
main(){
     unsigned x;
     cin >> x;
     cout << x << " = ";
     for (int i = 0; i < 16; i++) {
          cout << ( (x & 0x8000) ? '1' : '0' );
          x <<= 1;
     }
     cout <<  o\n"
}
```

```
/* mayor3.cpp

     Programa que imprime el mayor de tres números
*/


#include <stdio.h>

main(){
     int  x=5, y=8, z=2;

          printf("%d es el numero mayor entre %d, %d y %d\n",
               ((x>y) ? ((x>z) ? x : z) : (y>z) ? y:z)),
               x, y,z);
}
```

Tabla de precedencia y asociatividad

Operador	Asociatividad
::	
() [] ->	izquierda a derecha
sizeof ! ~ ++ -- - (tipo) * & - + new delete	derecha a izquierda
* / %	izquierda a derecha
.* ->*	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
&	izquierda a derecha
^	izquierda a derecha
|	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
? :	derecha a izquierda
= op=	derecha a izquierda
,	izquierda a derecha

2.2 CONTROL DE FLUJO

Esta sección analiza la ejecución de las sentencias de control en C++. Deberá familiarizarse con ellas antes de poder leer códigos en C o en C++. C++ emplea todas las sentencias de control de flujo de C como **if-else, while, do-while, for, switch.**

Expresiones y sentencias

Una expresión puede ser:

- una variable o constante
- una llamada a una función
- una combinación de operandos y operadores

Una expresión es *verdadera* si devuelve un valor entero distinto de cero. Una expresión es *falsa* si lo que devuelve es un cero (entero).

Ejemplos de expresiones:

```
int  a,b,c;

a=5  a++        135.4          a*b/c          sin(a)
```

Una sentencia es una expresión terminada con ";", ejemplos:

```
a = 5;
sin(a);
a++;
```

Un bloque es una colección de sentencias agrupadas por "{" y "}" que se les considera como una sola sentencia, ejemplo:

```
{
        a = 5;
        sin(a);
        a++;
}
```

if

Un if es en si una sentencia condicional

Su sintaxis es la siguiente:

```
if (expresión)
     sentencia
else
     sentencia
```

La parte *else* es opcional

En construcciones anidadas, la parte *else* termina el if más interno, el compilador no toma en cuenta el sangrado:

```
...

if (n>b)
     if (n > c)
          z = c;
else
     z = 0;
...
```

Existen errores comunes como el siguiente:

```
...

if ( x=5)
     printf("valor correcto\n");
else
     printf("valor incorrecto");

...
```

Una decisión múltiple puede implementarse con una serie de if anidados; sin embargo, el sangrar cada una de las sentencias provocaría que el tamaño de la línea creciera demasiado, para ello se emplea una construcción como la siguiente:

```
if (expresión)
     sentencia
else if (expresión)
     sentencia
else if (expresión)
     sentencia
else if (expresión)
     sentencia
else if (expresión)
     sentencia
else
     sentencia
```

En la construcción anterior, las expresiones se evalúan en orden, cuando alguna de ellas es verdadera, la sentencia asociada se ejecuta y con esto se termina la construcción.

La sentencia del último else se ejecuta cuando ninguna expresión es verdadera.

Ejemplo:

```
...

if (x > y)
     printf("%d es mayor que %d\n",x,y);
else if (y > x)
     printf("%d es mayor que %d\n",y,x);
else
     printf("%d y %d son iguales\n",x,y);

...
```

while

Sintaxis:

```
while (expresión)
     sentencia
```

La *sentencia* se ejecuta mientras la evaluación de la *expresión* sea verdadera

do

Sintaxis:

```
do
     sentencia
while (expresión);
```

La secuencia de ejecución es la siguiente:

1. Se ejecuta la *sentencia*.
2. Se evalúa la *expresión:*

   si la evaluación es falsa termina el ciclo.
   si la evaluación es verdadera se vuelve al paso 1

for

Sintaxis:

```
for(expresión1 ; expresión2 ; expresión3)
     sentencia
```

La secuencia de ejecución es la siguiente:

1. Se ejecuta la *expresión1*.
2. Se evalúa la *expresión2*:

   si la evaluación es falsa, termina el for.
   si la evaluación es verdadera, se continua en el paso 3.

3. Se ejecuta la *sentencia*.
4. Se evalúa la *expresión3*.
5. Se regresa al paso 2

Cualquiera de las expresiones se puede omitir

Si se omite la segunda expresión, se trata de un ciclo infinito.

Operador coma

Este operador sirve para agrupar dos expresiones como una sola, frecuentemente es utilizado en la sentencia *for* para colocar expresiones múltiples en la expresión1 o en la expresión3, para procesamiento de índices en paralelo.

La sintaxis es la siguiente:

expresión1, expresión2

El resultado y tipo de la expresión anterior son el resultado y tipo de expresión2.

Ejemplo:

```
/*
     Programa que despliega dos columnas de números, una en forma
     ascendente y otra en forma descendente.
*/
#include <stdio.h>
#define   N    10
main(){
     int  i, j;
     for(i=0, j=N; i<=N && j>=0; i++, j--)
               printf("%d     %d\n",i,j);
}
```

Palabras reservadas *break* **y** *continue*

Dentro del cuerpo de cualquiera de las estructuras repetitivas, se puede controlar el flujo de éstas, utilizando **break** y **continue**. *break* permite salir del ciclo (bucle,loop) sin ejecutar el resto de las sentencias que haya dentro de él. Un break provoca que el ciclo o **switch** más interno que lo encierra termine inmediatamente. *continue* detiene la ejecución de la iteración en curso y vuelve al principio del ciclo para comenzar una nueva iteración.

Un break causa una salida inmediata de las siguientes construcciones:

- while
- for
- do
- switch

La siguiente función, trim, elimina espacios en blanco, tabuladores y carácter nueva línea al final de una cadena, utilizando un *break* para salir de un ciclo cuando se encuentra el no-blanco, no-tabulador o no-nueva línea de más a la derecha.

```
/*
     trim: elimina blancos, tabuladores y nueva línea al final de
     la cadena.
*/
int trim ( char s[] ){
     int n;

     for (n = strlen(s)-1; n >= 0 ; --n)
          if ( s[n] != ' ' && s[n] != '\t' && s[n] != '\n' )
                    break;
     s[n+1] = '\0';
     return n;
}
```

La proposición *continue* está relacionada con el *break*, pero se utiliza menos. Se aplica solamente a ciclos, no a *switch*. El siguiente fragmento procesa sólo los elementos positivos que están en el arreglo *a*; los elementos negativos son ignorados.

```
     for (i = 0; i < n ; i++){
          if ( a[i] < 0 ) //ignora elementos negativos
               continue;
          // trabaja con elementos negativos
          //...
     }
```

switch

La proposición switch permite la implementación de decisiones múltiples con valores enteros.

Sintaxis:

```
switch (expresión) {
     case exp-const1:  sentencias; break;
     case exp-const2:  sentencias; break;
     //...
     default: sentencias;
}
```

donde: exp-const = expresión constante entera

La expresión se evalúa y el resultado se compara con las expresiones constantes; si alguna de ellas coincide, el control del programa se traslada a ese punto. Posteriormente se ejecutan las sentencias hasta encontrar la palabra reservada *break* o llegar a la llave de fin de bloque del *switch*.

La expresiones constantes deben ser enteras y no se deben repetir.

Las sentencias después de la expresión constante no se necesitan agrupar como bloque.

La cláusula *default* es opcional e indica el lugar a donde se traslada el control del programa en el caso en que ninguna de las etiquetas *case* coincidan con el valor de la expresión.
Ejemplo:

```
...
int  x = 3;
switch (x) {
     case 1: printf("*\n");
     case 2: printf("**\n");
     case 3: printf("***\n");
     case 4: printf("****\n");
}

...
```

La salida del fragmento anterior sería:

```
***
****
```

debido a que el valor de *3* coincide con la etiqueta *case 3:*, siendo este el punto por donde entra el flujo al *switch*. Se ejecuta la sentencia printf("***\n") y al no encontrar la palabra *break* se ejecuta la sentencia printf("****\n") y termina el *switch*.

```
/* cuendbo.cpp
     Cuenta dígitos, espacios blancos, y otros
*/
#include <stdio.h>
main(){
     int c,i,nblan,notr,ndigit[10];

     nblan = notr = 0;
     for (i=0 ; i < 10 ; i++)
          ndigit[i] = 0;
     while ((c = getchar())!=EOF){
          switch( c ){
          case '0': case '1': case '2': case '3': case '4':
          case '5': case '6': case '7': case '8': case '9':
               ndigit[c-'0']++;
               break;
          case ' ':
          case '\n':
          case '\t':
               nblan++;
               break;
          default:
               notr++;
               break;
          }
     }
     printf("dígitos=");
     for (i = 0; i < 10; i++)
          printf(" %d \n", ndigit[i]);
     printf(" espacios en blanco = %d, otros = %d \n",
          nblan, notr);
}
```

2.3 FUNCIONES

Tradicionalmente la modularidad de los sistemas de software se ha basado en el concepto de función. Aunque el centro de atención de la programación orientada a objetos no son las funciones en que se puede dividir un programa, sino los objetos que manipula ese programa, en lenguajes como C++ las funciones siguen siendo una parte fundamental en el desarrollo de sistemas. Un programa en C++ consiste en un conjunto de declaraciones de tipo de datos abstractos y un conjunto de funciones que manipulan variables de esos tipos. Las funciones en C++ se utilizan además para describir los métodos de los objetos de cierta clase.

En este capítulo, se discuten las características del lenguaje C++ relacionadas con la definición de funciones, tales como paso de parámetros, funciones en línea, sobrecarga de funciones y argumentos por omisión.

2.3.1 Definición de funciones.

La forma general de la definición de una función es:

tipo nombre (declaración de parámetros)
{
    proposiciones
}

Una función puede o no regresar un valor. Si lo regresa, el tipo de ese valor se debe especificar en su definición, además de que se deba incluir en el cuerpo de la función una proposición **return** para especificar cuál es ese valor. La proposición

return *expresión;*

evalúa la expresión especificada, convierte el resultado al tipo de la función y regresa el control a la función que hizo la llamada de la función actual. Cuando no se especifica el tipo de una función, muchos de los compiladores asumen que ésta regresa un valor entero; sin embargo, es una buena práctica de programación especificar siempre el tipo, aún si es **int**. El valor que regresa una función puede ser ignorado por la función que la llama.

Cuando una función no regresa ningún valor, se debe especificar el tipo **void** en su definición. En este tipo de funciones, si se utiliza la proposición **return**, debe ser de la forma

```
return;
```

para indicar que el control debe regresar a la función que hizo la llamada.

Los parámetros de una función se deben declarar en el encabezado de su definición en una lista de la forma

tipo identificador, tipo identificador, ...

En particular

```
int f(int x, y, z) { /* ... */ }
```

produce un error de sintaxis.

Existe una gran diferencia entre las listas de argumentos vacías de C (tanto ANSI como K&R) y las C++. En C la declaración

```
int func2();
```

significa "una función con cualquier tipo y número de argumentos", mientras que en C++ "una función sin argumentos". Por tanto, si se declara una función con una lista de argumentos vacía en C++, recuerde que se hace en forma diferente que en C. Tanto las variables definidas dentro del cuerpo de una función como sus parámetros son **locales** a ella.

2.3.2 Paso de parámetros.

Cuando se invoca a una función los argumentos se pasan por valor; esto quiere decir que *cada argumento se evalúa y su valor se copia hacia una variable local* a la función que se está invocando. De esta forma, si una variable se pasa como parámetro a una función, el valor almacenado en esa variable no cambia después de la invocasión. Por ejemplo, el programa de la figura 2.1 tendrá como salida

```
x = 10, y = 20
.x = 20, y = 10
x = 10, y = 20
```

```
#include <stdio.h>

void swap(int x, int y);

main(){
     int x = 10, y = 20;
     printf("x = %d, y = %d\n", x, y);
     swap(x, y);
     printf("x = %d, y = %d\n", x, y);
}

void swap(int x, int y){
     int t = x;
     x = y;
     y = t;
     printf(".x = %d, y = %d\n", x, y );
}
```

Figura 2.1. Paso de parametros por valor.

Cuando un arreglo se pasa como parámetro a una función, no se hace una copia de cada uno de sus valores, sino que únicamente se *copia la dirección de memoria de su primer elemento*. Este tipo de paso de parámetros (conocido como *paso por referencia*), la función sí puede modificar el valor de los elementos del arreglo.

Las funciones de C++ se pueden invocar en forma *recursiva;* esto es, una función puede hacer una llamada a sí misma directa o indirectamente.

2.3.3 Prototipos.

Todas las funciones deben declararse antes de que otra función las pueda invocar. Una declaración de función en ANSI C y en C++ da el nombre de función, los tipos de argumentos pasados a la función y el valor resultante de la misma. La declaración de una función se hace mediante un *prototipo*, por ejemplo:

```
double newton_raphson (double x0, double precisión);
```

que especifica que la función **newton_raphson** recibe dos parámetros de tipo **double** *(x0 y precisión)* y regresa un valor de tipo **double**. Con esta información el compilador puede detectar errores en el uso de la función:

```
double newton_raphson (double x0, double precisión);

main(){
  //.....
  newton_raphson(3.0);          //error: la función debe
                                //recibir dos argumentos.
  newton_raphson(3.0, "hola");//error: el segundo argumento
                                //no es del tipo adecuado.
}
```

En un prototipo no es necesario especificar el nombre de los parámetros, sólo basta especificar el tipo. Por lo que la siguiente declaración es equivalente a la anterior:

```
double newton_raphson ( double , double );
```

Si una función se invoca antes de haber especificado su prototipo el compilador asume que la función regresa un valor entero y que puede recibir cualquier número de argumentos de cualquier tipo (para algunos compiladores el no especificar el prototipo de una función genera un error, otros toman la primer llamada a la función como prototipo).

2.3.4 Argumentos por omisión.

El lenguaje C++ permite especificar *argumentos por omisión* al momento de declarar una función; cada argumento de este tipo representa un valor que se utiliza cuando no se especifica el valor

en la llamada a la función. Los argumentos por omisión se especifican cuando se declara la función.

```
void func( int i=0 );
```

Se puede llamar a la función como **func()** (que es la misma que **func(0)**). Observe que el nombre de la variable **i** es opcional en la declaración, incluso en los argumentos por omisión.

Puede tener más de un argumento por omisión en la lista pero todos los argumentos por omisión deben estar al final de la lista, de esta forma:

```
void punto(int x, int y, int color=0, int ancho=1, int relleno=1);
```

Con esta definición, se puede dibujar un punto con el color 0, ancho 1 y un relleno de 1; con la siguiente llamada:

```
punto( 15,27 );
```

En el ejemplo de la figura 2.2, la primer llamada a la función **newton_raphson** asigna un valor de $1x10^{-5}$ al parámetro *precisión*, mientras que la segunda le asigna un valor de $1x10^{-11}$.

```
#include <stdio.h>        // prototipo de printf
#include <math.h>         // prototipo de fabs
double f(double);
double g(double);
double newton_raphson( double, double = 1e-6 );

main(){
     double x = newton_raphson( 3.0 );
     double y = newton_raphson( 3.0, 1e-12 );
}

double f(double x){
     return 10 * x * x * x - 5 * x + 10;
}

double g(double x){
     return 40 * x * x - 5;
}

double newton_raphson(double x0, double precisión){
     double error;
     do {
          double x1 = x0 - f(x0) / g(x0);
          error = fabs( x0 - x1 );
          x0 = x1;
          printf("x0 = %lf\n",x0);
     } while ( error > precisión );
     return x0;
```

Figura 2.1. Argumentos por omisión.

Enfatizamos que los parámetros que reciban valores por omisión deben ser los últimos en la lista de parámetros de la función. Observe que el valor de los argumentos se debe especificar en la declaración de la función, mientras que es opcional en la definición.

2.3.5 Sobrecarga de funciones.

Las funciones de un programa en C++ pueden sobrecargarse; es decir, se pueden escribir varias funciones diferentes que se pueden invocar con el mismo nombre. Esta característica es útil cuando se debe aplicar la misma operación a objetos de tipos diferentes. Antes de sobrecargar una función es necesario advertir al compilador usando la palabra **overload** antes de la declaración de funciones sobrecargadas (en la versión 2.0 de C++ esto no es necesario). La sobrecarga de una función permite utilizar el mismo nombre de mensaje con diferentes argumentos para que el compilador sepa cómo manejarlo. Así, usted no tendrá que recordar tantos nombres de mensajes por lo trabajará menos y la computadora más. El lema podría ser: "Deje que el programador piense sobre el diseño y encargue a la computadora el manejo de la implementación". Para un ejemplo de esta característica del lenguaje revise la figura 2.3.

```
#include <stdio.h>

void print (char *);
void print (int);

main() {
     print(3);        // las dos funciones se invocan con el
     print("hola. \n"); // mismo nombre
}

void print(int n) {  // despliega el entero n
     char s[16];
     int i,n,sign;

     if ((sign = n) < 0)       //anota el signo
          n = -n
     i = 0;
     do {            // genera digitos en orden inverso
          s[i++] = n % 10 + '0';   // obtiene el sig. digito
     } while ( (n /= 10) > 0 );
     if ( sign < 0 )
          s[i++] = '-';

     while (  i >= 0  )
          putchar (s[i--]);
}

void print(char *s) {  // despliega la cadena s
     while (*s)
          putchar(*s++);
}
```

Figura 2.3 Sobrecarga de funciones

2.3.6 Funciones en línea.

Las macros de ANSI C y C++ ahorran tiempo de escritura, mejora la legibilidad, reduce errores y elimina la llamada a una función. Las funciones macro del preprocesador tienen la desventaja de que no son funciones "reales", por lo que la comprobación de errores no ocurre durante la compilación.

C++ fomenta el uso de pequeñas funciones. El programador

preocupado por la rapidez, sin embargo, podría utilizar las macros del preprocesador, más que las funciones para eliminar la llamada de una función. Para eliminar el costo de llamadas a funciones pequeñas, C++ tiene **funciones en línea** (*inline*). Estas funciones se especifican modificando la definición con la palabra reservada **inline**, de esta manera:

```
inline double sqr(double x) {
     return x * x;
}
```

Cuando el compilador encuentra la definición de una función en línea no genera código, sino que, cuando encuentra una llamada a esa función, la sustituye por su código. Las funciones en línea combinan la eficiencia de las macros del preprocesador de C con el chequeo de parámetros de una función convencional.

2.3.7 Alcance.

El *alcance* de una declaración es la región de un programa en donde esa declaración es válida. Un objeto que se declara dentro de un bloque de instrucciones, llamado *interno o local*, tiene como alcance el bloque en donde está declarado:

```
// ...
for (int i = 0; i < MAX; i++) {
     if ( s[i] == 'x' )
          int encontrado = 1;
}
if (encontrado)  // error, la variable 'encontrado' no
                 // está declarada en este bloque.
```

Un objeto *externo*, también llamado *global*, se declara fuera de cualquier función y su alcance va desde el punto en que se encuentra la declaración hasta el final del archivo. Todas las funciones deben ser externas, pues C++ no permite la definición de una función dentro de otra. Cualquier objeto externo es potencialmente utilizable por todas las funciones que se encuentren después de su declaración:

```
main() {
     // ...
     double suma = total_ventas; // error, 'total_ventas' aún
                                 // no se encuentra declarada
     // ...
}
```

```
// otras funciones
double total_ventas = 0;
double calcula_sueldos() {
     // ...
     double comisión = total_ventas * 0.15; // bien, ya está
                                            // declarada
     // ...
}
```

Las variables externas pueden resultar útiles cuando un conjunto de funciones comparten un gran número de variables, pues con ellas se puede evitar la sobrecarga de utilizar una lista de parámetros demasiado larga. Sin embargo el uso de este tipo de variables es peligroso, pues la interfase entre las funciones que las utilizan no se pueden notar a simple vista y una de ellas puede afectar a una variable en forma inesperada, produciendo un error muy difícil de descubrir. El uso de clases (capítulo 3) anula la necesidad de variables externas.

Cuando se necesita utilizar una variable antes de su definición es necesario declararla como externa (utilizando la palabra reservada **extern**):

```
extern double total_ventas; // declaración de 'total_ventas'
main(){
     // ...
     double suma = total_ventas; // ok, 'total_ventas' ya
                                 // está declarada
     // ...
}
double total_ventas = 0;    // definición de 'total_ventas'
double calcula_sueldos() {
     // ...
     double comisión = total_ventas * 0.15; // ok
     // ...
}
```

Más frecuentemente, la palabra extern se utiliza para accesar variables definidas en archivos diferentes:

Archivo #1

```
extern double total_ventas;
main() {
     // ...
     double suma=total_ventas;
     // ...
}
```

Archivo #2

```
double total_ventas = 0;
double calcula_sueldo() {
     // ...
     double comisión =
     total_ventas * 0.15;
     // ...
}
```

2.3.8 Tipos de almacenamiento.

El *tipo de almacenamiento* de un objeto determina el lugar de un programa en el que se almacena un objeto. En C++ hay cuatro tipos de almacenamiento: *automático, en registro, estático y libre*.

Todas las variables definidas dentro de un bloque y los parámetros que recibe una función son *automáticos*. Una variable automática se crea en el momento de su definición y se destruye al terminar la ejecución del bloque en el que está definida. La palabra reservada *auto* se puede utilizar para enfatizar que una variable es automática:

```
auto int i;  // equivale a int i;
```

Típicamente, una variable automática se almacena en el *segmento de stack* del programa; si se utiliza la palabra **register** en la definición de la variable, el compilador tratará de mantenerla en los *registros de la CPU*.

```
register int i;
```

Una variable registro tienen un tiempo de acceso considerablemente menor al de una variable que se almacena en memoria; sin embargo, el número y tipo de estas variables está limitado por el número y tamaño de los registros en el procesador, por lo que las variables registro únicamente pueden almacenar valores enteros o apuntadores.

Si la definición de una variable interna a un bloque se modifica con la palabra **static**, la variable adquiere un tipo de *almacenamiento estático*, por lo que no se destruye al terminar la ejecución de ese bloque; por ejemplo, el programa de la figura 2.4 produce la salida

```
10  11  12  13  14  15  16  17  18  19
```

```
#include <stdio.h>
void f(void);
main() {
     for (int i = 0; i < 10; i++)
          f();
}

void f() {
     static int k = 10;
     printf("%d ", k++);
}
```

Figura 1.4 Variables estáticas

Cuando una variable externa se califica como estática, su acceso queda restringido al archivo en que se declara. Este tipo de declaraciones se utilizan cuando es necesario compartir variables entre un conjunto de funciones, pero es necesario compartir variables entre un conjunto de funciones, pero es necesario esconder dichas variables al resto del programa:

Archivo #1

```
static double inc = 3.0;
int f() {
     // ...
     inc += 1.37;
     // ...
}

int g() {
     // ...
     double x = inc;
     // ...
```

Archivo #2

```
extern double inc;
// error, 'inc' es pri-
// vada al archivo 1


main() {
     // ...
}
```

Las variables externas tienen almacenamiento estático. Por omisión, las variables externas y estáticas se inicializan con cero; las variables automáticas no se inicializan.

Si una variable automática se inicializa explícitamente, esa inicialización se hace cada vez que se ejecuta la proposición de definición. Las variables estáticas y externas se inicializan una sola vez, antes de que se inicie la ejecución de main.

2.3.9 Inicialización.

Los arreglos se pueden inicializar precediendo su definición por una lista de inicializadores separados por comas y encerrados entre llaves:

```
int dígitos[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

No es válido especificar un número de inicializadores mayor a la dimensión del arreglo, aunque se pueden especificar menos. Los arreglos de caracteres pueden inicializarse con la notación

```
char hola[5] = "hola";
```

que es equivalente a

```
char hola[] = {'h',,'o', 'l', 'a', '\0'};
```

Cuando se omite la dimensión del arreglo, el compilador se encarga de calcularla:

```
char f16[] = "Falcon";
```

CAPITULO III

Uso de clases

Como se mencionó en el primer capítulo, la programación orientada a objetos se basa principalmente en la creación de tipos abstractos. La abstracción de datos permite al programador expresar la solución de un problema usando objetos internos a su programa que tienen una correspondencia directa con entidades en el dominio del problema. La descripción de un tipo de datos abstracto se hace utilizando el concepto de **clase**. En este capítulo se analizan las características básicas de C++ que permiten la definición de clases.

3.1 Definición de clases en C++.

Una clase se utiliza para describir un conjunto de objetos que tienen características comunes y que se comportan de manera similar. En C++ una clase se declara utilizando la palabra reservada **class**, que define los atributos y métodos de una clase (recordemos que los *atributos* son todos los datos que describen a los objetos de una clase, mientras que los *métodos* son las funciones que determinan el comportamiento de los objetos). Tomemos como ejemplo la clase *stack*, un tipo definido para manejar pilas estáticas de números reales.

```
class stack {  // Definición de la clase stack.
     float *p;  // Apuntador hacia los datos.
     int  max;  // Número máximo de elementos en el stack.
     int    n;  // Número de elementos. 0 <= tope <= max.
public:
     void crea(int nmax = 50); // Inicializa el stack.
     void destruye(); // Borra el stack.
     void push(float x); // Inserta un elemento.
     float pop(); // Obtiene y borra un elemento del stack.
     int vacio(); // Indica si el stack contiene datos.
};
```

En este caso, los métodos de la clase están dados por las funciones *push, pop, vacio, crea y destruye* y los atributos por las variables *n, max y p* (en realidad la definición de la clase aún está incompleta, pues los métodos no se definen todavía).

Una vez que se hace la definición de la clase, **stack** se convierte en un tipo que puede usarse de manera similar a los tipos internos del lenguaje; por ejemplo

```
stack s1, s2;
```

define las variables de tipo stack s1 y s2.

Los miembros de una clase se accesan con el operador punto:

```
s1.crea();           // inicializa al stack s1
s1.push(5.67);       // inserta 5.67 en el s1
s2.crea();           // inicializa el stack s2
const float pi = 3.14159;
s2.push(pi);         // inserta 3.14159 en s2
s1.push(s2.pop());   // toma un elemento de s2 y lo inserta
                     // en s1
s1.destruye();       // borra el stack s1
s2.destruye();       // borra el stack s2
```

Note que la declaración de la clase *stack* se encuentra dividida en dos partes por la palabra reservada public. Los elementos que se encuentran en la primer parte de la declaración son *privados* a la clase (sólo pueden ser usados por los métodos de la misma clase); los que se encuentran en la segunda parte son *públicos* a todo el programa (pueden ser usados por cualquier otra función). Cualquier atributo o método puede ser público, sin embargo, generalmente todos los atributos se declaran privados a la clase (recuerde el principio de ocultamiento de información). Si se desea tener acceso a alguno de estos atributos privados se debe añadir a la clase un método que realice esa tarea, como es el caso de vacio, que permite preguntar por el valor de *n* (número de elementos del stack).

```
stack s; // define un objeto 's'
s.crea(); // inicializa un objeto
// ...
if (s.n == 0) // error, 'n' es un miembro privado
     // ...
if (s.vacio()) // correcto, 'vacio' es un miembro público
     // ...
```

El propósito de organizar la declaración de una clase de esta

forma es separar la *implementación* de la clase de su *interface* (recuerde que ésta es una propiedad de los tipos de datos abstractos).

La definición de una clase generalmente se separa en dos archivos: un archivo de encabezados en donde se define la clase (el bloque class ... { ... }) y un archivo en el que se definen los métodos. El archivo de encabezados se debe incluir en aquellos archivos en los que se utilice la clase, mientras que el código objeto del segundo archivo deberá ligarse con todos los archivos del programa. Las figuras 3.1 y 3.2, al final de capítulo, ilustran este tipo de organización.

Observe en la segunda figura cómo se utiliza el operador :: para asociar un método con la clase a la que pertenece. De esta forma, varias clases diferentes pueden tener métodos con los mismos nombres.

Una clase puede tener varios métodos con el mismo nombre, siempre y cuando todos ellos reciban diferentes parámetros. Cuando se sobrecarga un método, no es necesario utilizar la palabra **overload**.

3.2 Definición de miembros en línea.

Cuando se realiza el diseño de una clase es muy común encontrar que muchos de los métodos son funciones de unas cuantas líneas. Si esos métodos se emplean constantemente en un programa la sobrecarga causada por las llamadas funciones puede resultar demasiado costosa. Las funciones en línea pueden ser útiles para minimizar ese problema:

```
class stack {
     // igual al ejemplo anterior
};
// mismas definiciones para push, pop, crea y destruye

inline int stack::vacio() { // ahora vacio es un miembro en
     return n == 0;         // línea.
}
```

Los métodos en línea deben colocarse en el archivo de encabezados en el que se define la clase. Otra forma de declarar un método en línea es definirlo dentro de la declaración de la clase a la que pertenece, por ejemplo:

```
class stack {
      float *p;
      int  max;
      int    n;
public:
      void crea(int nmax = 50);
      void destruye();
      void push(float x);
      float pop();

      int vacio() { // vacio es un método en línea
            return n == 0;
      }
};
// mismas definiciones para push, pop, crea y destruye.
```

3.3 Constructores y destructores.

La mayoría de los tipos de datos abstractos requieren de métodos de inicialización y destrucción (tales como *crea y destruye* en nuestro ejemplo). Sin embargo, este tipo de métodos favorece que, en un programa grande, aparezcan errores como olvidar llamar al método de inicialización, inicializar un objeto varias veces o invocar un método para un objeto que ya se haya destruído. El lenguaje C++ brinda facilidades que permiten evitar este tipo de errores.

Para cualquier clase *X*, se puede escribir un método llamado *X* (el mismo nombre que la clase) que realice la función de inicialización. Ese método es llamado *constructor* y se invoca automáticamente cada vez que se crea un nuevo objeto de la clase *X*. Similarmente, un método llamado ~*X* (tilde X) se convierte en el *destructor* de la clase y se invoca automáticamente cada vez que es necesario destruir un objeto de la clase *X*. Una clase puede tener varios constructores, pero sólo un destructor y éste no debe recibir parámetros. El ejemplo de la figura 3.3 utiliza un constructor y un destructor para la clase stack.

Cuando un constructor tiene parámetros, éstos se deben especificar en el momento de crear un objeto:

```
stack s1(30); // invoca al constructor con nmax = 30
stack s2;     // invoca al constructor con nmax = 50
stack *s3 = new stack(45); // crea un objeto en forma
                           // dinámica con nmax = 45
```

Si se desea construir un arreglo de objetos, la clase a la que pertenecen esos objetos debe tener un constructor que no reciba parámetros.,

Un objeto se puede inicializar también asignándole otro objeto de la misma clase; sin embargo, en este caso el constructor del objeto que se está inicializando no se invoca, sino que sólo se hace una copia bit a bit de cada uno de los atributos del objeto con el que se hace la inicialización:

```
stack s1(20); // crea s1 de 20 elementos
// ...
if ( ... ) {
     stack s2 = s1; // ¡cuidado!, s2 y s1 son ahora el mismo
                    // stack
     // ...
} // el Ambito de s2 termina, por lo que se invoca a su
  // destructor; sin embargo, también se destruye a s1
s1.push(1022); // error, s1 ya se destruyó.
```

El constructor de un objeto se invoca en el momento en el que se crea el objeto; el destructor se invoca al llegar a destruírlo. Por ejemplo, el constructor de un objeto automático se invoca al momento de definirlo y el destructor al llegar al fin del bloque en el que se encuentra la definición. Los constructores de objetos externos y estáticos se invocan antes de iniciar la ejecución de *main*; los destructores, después de terminar dicha función. El operador *new* invoca al constructor del objeto que está creando; el operador *delete* invoca al destructor.

El siguiente programa (figuras 3.4 a 3.8) implementa una calculadora de notación po[illegible]a utilizando una versión ligeramente modificada de la clase stack.

3.4 Atributos estáticos.

En algunas ocasiones un conjunto de objetos de la misma clase necesitan compartir cierta información para poder funcionar correctamente. Esa información se puede representar utilizando variables globales; sin embargo, una variable global puede ser modificada por cualquier función, lo cuál favorece la ocurrencia de errores (recuerde el principio de ocultamiento de información). Cuando un atributo se declara como estático, no se crea una copia de ese atributo para cada objeto de la clase, sino que se crea un sólo atributo que comparten todos los objetos de la clase. El compilador inicializa los atributos estáticos con ceros.

Como ejemplo, la siguiente clase (*raton*) no permite que se defina más de una instancia de ella:

```
class raton {
     static int ninst; // número de instancias
     // ...
public:
     raton();
     // ...
};
raton::raton() {
     ninst++; // se está creando una nueva instancia
     if (ninst > 1) {
          fprintf(stderr,"No se puede crear más de un"
                         " raton.");
          exit(1);
     }
     // ...
}
// ...
```

La versión 2.0 del lenguaje permite inicializar un atributo estático utilizando una proposición de inicialización externa:

```
raton::ninst = 0; // igual que el default
```

Los atributos estáticos disminuyen en gran medida la necesidad de variables globales.

3.5 Amigos.

Los miembros privados de una clase no pueden ser accesados por funciones ajenas a la clase; sin embargo, una clase puede exportar selectivamente sus miembros privados a una función determinada (llamada *amiga*). Una función amiga debe declararse en la definición de la clase utilizando la palabra **friend**, por ejemplo:

```
class procesoB; // declara la clase B por anticipado

class procesoA {
     int hora; // una medida de tiempo
     // ...
public:
     // ...
     friend void sincroniza(procesoA x, procesoB y);
     // sincroniza puede usar los miembros privados de
     // procesoA
};

class procesoB {
     int hora; // una medida de tiempo
     // ...
public:
     // ...
     friend void sincroniza(procesoA x, procesoB y);
     // sincroniza puede usar los miembros privados de
     // procesoB
};

void sincroniza(procesoA x, procesoB y) {
     x.hora = y.hora; // utiliza e latributo privado hora
}
```

Los miembros no sólo pueden ser funciones, sino también miembros de otras clases o clase completas:

```
class X { // cualquier clase
     // ...
public:
     friend class Y; // La clase Y es amiga de la clase X
     friend int Z::A(); // El método A de la clase Z es
                        // amigo   la clase X
     // ...
}
```

```
#ifdef STACK_H     // Para evitar declaraciones múltiples en
#define STACK_H    // 'includes' anidados.

class stack { // Definición de la clase stack.
              // Una pila de float.
     float *p; // Apuntador hacia los datos.
     int  max; // Número máximo de elementos en el stack.
     int    n; // Número de elementos. 0 <= n <= max.
public:
     void crea(int nmax = 50); // Inicializa el stack.
     void destruye(); // Borra el stack.
     void push(float x); // Inserta un elemento en el stack.
     float pop(); // Obtiene y borra un elemento del stack.
     int vacio(); // Indica si el stack contiene datos.
};

#endif STACK_H
```

Figura 3.1 Stack1.h: Declaración de la clase stack.

```
#include "stack1.h"  // Declaración de la clase stack.
#include <iostream.h>  // Declaraciones de cin, cout y cerr.

void stack::crea(int nmax) {
// Crea un stack vacio reservando espacio para'nmax'
// elementos.
// Observe que en la declaración de la clase 'nmax'
// tiene un valor por omisión de 50.
     max = nmax;
     p = new float[max]; // crear un arreglo p de max
                         // elementos.
     n = 0;
}

void stack::destruye() {
// Libera el espacio ocupado por el stack
     delete p;
     max = n = 0;
}

void stack::push(float x) {
// Inserta x en el stack.
     if (n == max) {
          cerr << "Error, stack lleno.\n";
     return;
     }
     p[n++] = x;
}

float stack::pop() {
// Regresa el valor que se encuentra al inicio del stack.
// Borra dicho valor.
     if (n == 0) }
          cerr << "Error, stack vacío.\n";
          return;
     }
     return p[--n];
}

int stack::vacio() {
// ¿Está vacío el stack?
     return n == 0;
}
```

Figura 3.2. Stack1.cpp Definición de los métodos de la clase stack.

```
class stack {
     float *p;
     int  max;
     int    n;
public:
     stack(int nmax = 50); //constructor
     ~stack(); // destructor

     void push(float x);
     float pop();
     int vacio() { return n == 0; }
};

stack::stack(int nmax) {
     max = nmax;
     p = new float[max];
     n = 0;
}

stack::~stack() {
     delete p;
     max = n = 0;
}

void stack::push(float x) {
     if (n == max) {
          cerr << "Error, stack lleno.\n";
          return;
     }
     p[n++] = x;
}

float stack::pop() {
     if (n == 0) {
          cerr << "Error, stack vacío.\n";
          return 0.0;
     }
     return p[--n];
}
```

Figura 3.3. Stack2.cpp: Constructores y destructores.

```
// CALC.CPP: Calculadora. Evalúa expresiones en notación
// polaca utilizando los operadores aritméticos +, -, / y *.
// El operador = muestra el resultado de una expresión; el
// operador c limpia el stack.

#include <math.h>
#include "stack.h"
#include "scanner.h"
const MAXOP = 20; // tamaño máximo de un operando

main() {
      int tipo;
      char s[MAXOP];
      stack st;
      double op2;
      while((tipo = getop(s,MAXOP)) != EOF)
            switch (tipo) {
                  case NUMERO:
                        st.push(atof(s));
                        break;
                  case '+':
                        st.push(st.pop() + st.pop());
                        break;
                  case '*':
                        st.push(st.pop() * st.pop());
                        break;
                  case '-':
                        op2 = st.pop();
                        st.push(st.pop() - op2);
                        break;
                  case '/':
                        op2 = st.pop();
                        if (op2 != 0.0)
                              st.push(st.pop() / op2);
                        else
                              cin << "División entre cero\n";
                        break;
                  case '=':
                        cin << "\t" << st.peep() << endl;
                        break;
                  case 'c':
                        st.limpia();
                        break;
                  case MUYGRANDE:
                        cin << "\t\t... es muy grande\n" << s << endl;
                        break;
                  default:
                        cin << "Comando desconocido " << tipo << endl;
                        break;
            }
}// fin main
```

Figura 3.4. Calc.cpp. Calculadora de notación polaca.

```
// SCANNER.H: Declaraciones para scanner.cpp

#include <stdio.h>
#ifndef SCANNER_H
#define SCANNER_H

#define NUMERO '0'  // indicador de número
#define MUYGRANDE '9' // la cadena es muy grande
int getop( char *s, int lim );

#endif
```

Figura 3.5. Scanner.h

```
// SCANNER.CPP: Reconoce operadores y operandos

#include "scanner.h"
#include <conio.h>

int getop( char *s, int lim ) { // Obtiene el próximo
     int c;                     // operador u operando.
     while((c = getche()) == ' ' || c == '\t' || c == '\n')
          ;
     if (c != '.' && (c < '0' || c > '9'))
          return c;
     s[0] = c;
     for (int i=1; (c = getche()) >= '0' && c <= '9'; i++)
          if (i < lim)
               s[i] = c;
     if ( c == '.' ) {
          if (i < lim)
               s[i] = c;
          for (i++; (c = getche()) >= '0' && c <= '9'; i++)
               if (i < lim)
                    s[i] = c;
     }
     if (i < lim) {
          ungetch(c);
          s[i] = '\0';
          return NUMERO;
     } else {
          while (c != '\n' && c != EOF)
               c = getchar();
          s[lim - 1] = '\0';
          return MUYGRANDE;
     }
}
```

Figura 3.6. Scanner.cpp: Reconocimiento de los tokens de la calculadora.

```
// STACK3.H: Encabezado para la clase stack.

#ifndef STACK_H
#define STACK_H

class stack {
     int n;          // Número de elementos
     int nmax;       // Número máxinmo de elementos
     double *d;      // Apuntador a los datos
public:
     stack (int max = 10);
     ~stack();

     double push(double); // inserta un número en el stack
     double pop(void); // obtiene y borra el primer elemento
     double peep(void)    // obtiene el primer elemento
     void limpia(void)    // inicializa el stack
};

#define STACK_H
```

Figura 3.7. Stack3.h: Declaración de la clase stack.

```
//STACK3.CPP: Métodos de la clase stack

#include <iostream.h>
#include "stack.h"

stack::stack (int i) {
        n = 0;
        nmax = i;
        a = new double[nmax];
}

stack::~stack() {
        delete a;
}

double stack::push(double f) {
        if (n == nmax) {
                cerr "Error: stack lleno.\n" << endl;
                return 0;
        } else {
                a[n++] = f;
                return f;
        }
}

double stack::pop(void) {
        if (n == 0) {
                cerr << "Error. stack vacio.\n";
                return 0;
        } else {
                return a[--n];
        }
}

double stack::peep(void) {
        if (n == 0) {
                cerr << "(peep) Error: stack vacio.\n";
                return 0;
        } else
                return a[n-1];
}
void stack::limpia(void) {
        n = 0;
}
```

Figura 3.3. Stack3.cpp. Metodos de la clase stack.

CAPITULO IV

Definición de operadores

Uno de los objetivos del diseño de C++ es brindar al programador las facilidades necesarias para definir tipos de datos que se puedan manipular de manera similar a los tipos básicos del lenguaje. Una de esas facilidades es la definición de clases con operadores; C++ permite redefinir el significado de sus operadores para que puedan ser aplicados a cualquier tipo de datos abstractos. Este aspecto de C++ se discute en las siguientes secciones.

4.1 Definición de operadores.

El ejemplo de la figura 4.1 muestra la definición de una clase (*cadena*) que permite manejar cadenas de caracteres de longitud arbitraria (el constructor *cadena(cadena &)* es necesario para poder pasar cadenas como parámetros de una función, su uso se explica hasta la sección 4.3).

El tipo *cadena* se puede utilizar en proposiciones como:

```
cadena t("Hola");              // t <- "Hola"
cadena s;                      // s <- cadena nula
s.asigna(t);                   // s <- t (s <- "Hola")
s.asigna(s,concatena(t));      // s <- s + t (s <- "HolaHola")
t.asigna(s.subcadena(4,2))l // t <- "Ho"
```

Cuando un constructor recibe un sólo parámetro, se puede utilizar también el símbolo = en su inicialización. Por ejemplo, la proposición

```
cadena t("Hola");
```

es equivalente a

```
cadena t = "Hola"
```

Como muestran las dos últimas proposiciones, esta notación para manejar cadenas puede resultar poco práctica en expresiones complicadas, como en

```
cadena a = "Uno";
cadena d = "Dos";
cadena c = "Tres";
cadena d;
d.asigna((a.concatena(b)).concatena(c)); // d <- a + b + c
```

El lenguaje C++ permite redefinir el significado de sus operadores de tal modo que se puedan utilizar con objetos de cualquier clase. En muchos casos, el uso de operadores permite simplificar la notación en operaciones complicadas, facilitando la lectura del programa y mejorando la comprensión del problema. Una función llamada **operator** *op*, en donde *op* es cualquier operador de C++ diferente a **.** (punto), **::** (alcance) y **?:** (operador condicional), se utiliza para redefinir el significado de *op*. La figura 4.2 repite el ejemplo de la clase *cadena* utilizando operadores en vez de llamadas a funciones. Observe cuidadosamente la redefinición de los operadores **()**, llamada a función, y **[]**, subíndice de un arreglo.

Utilizando operadores, las proposiciones del ejemplo anterior se convierten ahora en:

```
cadena t = "Hola";
cadena s;                    // s <- cadena nula
s = t;                       // s.operador=(t)
s = s + t;                   // s.operador=(s.operador+(t))
t = s(4,2);                  // t.operador+(t))
cadena a = "Uno";
cadena b = "Dos";
cadena c = "Tres";
cadena d;
d = a + b + c;  //d.operador=((a.operador+(b).operador+(c));
```

Cuando se redefine el significado de un operador, éste conserva su asociatividad, su precedencia y su aridad (número de operandos); en particular, no es posible definir un operador **++** binario, ni un operador **+** asociado por la derecha. El compilador no asume ningún significado predefinido para un operador; por ejemplo, si se define el operador **+=** para la clase *cadena*, la expresión

```
s += t;                  // s.operador+=(t)
```

no es equivalente a

```
s = s + t                // s.operador=(s.operador+(t))
```

Todos los operadores deben tener entre sus argumentos por lo menos un objeto de un tipo definido por el usuario; sin embargo no es necesario que los operadores sean miembros de una clase.

Debido a que todos los métodos reciben como primer parámetro una instancia de la clase a la que pertenecen, cualquier operador binario miembro de una clase únicamente debe tener un parámetro en su definición; si el operador es unario no debe tener parámetros.

4.2 Conversión de tipos.

Los constructores de una clase se pueden invocar directamente en un programa sin necesidad de asociarlos con un objeto; esta operación puede conceptualizarse como la creación de una constante:

```
cadena s = "Hola";
cadena t;
t = s + cadena("Adiós");  // t <- "Hola Adiós"
```

La expresión anterior, *cadena("Adiós")*, crea un objeto temporal que se inicializa con la cadena "Adiós"; al terminar de evaluar la expresión, ese objeto se destruye. Si el constructor sólo recibe un parámetro, se puede omitir su nombre:

```
cadena s = " Hola";
cadena t;
t = s + "Adiós";          // t <- "Hola Adiós"
```

La creación de una constante puede verse también como una conversión de tipo; en el ejemplo anterior, el apuntador a carácter *"Adiós"* se convierte al tipo *cadena*.

El diseñador de una clase puede también definir operadores de conversión. Para una clase **X**, el operador **X::operator T()**, en donde **T** es un tipo cualquiera, define una conversión del tipo de **T** a la clase **X**. La clase *cadena* puede tener operadores de conversión que permitan accesar sus atributos:

```
#include <string.h>
class cadena {
     // ...
public:
     // ...
     operator int () { return l; } // conversión a int
     operator const char *() { return v; } // conversión a
                                           // char *
};

main() {
     cadena s = "Hola";
     s = s + "Adiós";
     int longitud = s;  // s <- 10 (llama implícitamente a
                        // cadena::int ())
     printf("%s", (char *) s);  // imprime "Hola Adiós"
     // ...
}
```

Se debe tener especial cuidado para lograr que no existan reglas de conversión ambiguas. El siguiente fragmento de código produce un error de compilación:

```
class x {
     // ...
     x (int);  // conversión de int a x
};

class y {
     // ...
     y (int);  // conversión de int a y
};

int f(x);      // f puede recibir un x o un y
int f(y);
```

```
main() {
     // ...
     f(1);      // error, ambigüedad entre f(x(1)) y f(y(1))
     f(x(1));   // correcto
     // ...
}
```

El compilador nunca aplica reglas de conversión definidas por el usuario a más de un nivel de profundidad:

```
class x {
     // ...
     x (int); // conversión de int a x
};

class y {
     // ...
     y (x);   // conversión de x a y
};

int f(y);

main() {
     // ...
     f(1);    // error, nunca se intenta f(y(x(1)))
     f(x(1)); // correcto
     // ...
}
```

4.3 inicialización.

Como se mencionó en el capítulo anterior, un objeto puede ser inicializado al momento de su definición con otro objeto de la misma clase. Cuando se realiza esta operación, el constructor del objeto que se define no se invoca, sino que sólo se hace una copia bit a bit del contenido del objeto con el que se está haciendo la inicialización. Este comportamiento puede acarrear problemas cuando los objetos tienen miembros de tipo apuntador:

```
int f() {
     cadena s = "Hola";
     cadena t = s;    // no se invoca constructor para t. Las
                      // cadenas s y t apuntan al mismo valor.
     // ...
}
```

Al terminar la ejecución de la función se invoca el destructor para *s*; al invocar el destructor de t se trata de borrar un vector que ya no existe, causando un error de corrida.

Este tipo de problemas se puede evitar definiendo un constructor que reciba como parámetro una referencia hacia un objeto de la misma clase. Para cualquier clase **X**, un constructor que reciba una referencia hacia un objeto de la misma clase, **X::X(X&)**, se encarga de realizar las tareas de inicialización; ese constructor se utiliza también para copiar el valor de un objeto cuando se utiliza como argumento de una función. Para la clase *cadena*, ese método se puede escribir como:

```
cadena::cadena (const cadena &s) {
     l = s.l;
     v = new char[l +1];
     strcpy(v, s.v );
}
```

4.4 Operadores amigos.

Tal como se encuentra definida la clase *cadena* en este momento, es posible escribir expresiones como

```
cadena s = "Hola";
cadena t = s + "Adiós";  //t(s.operator+("Adiós"))
```

sin embargo, una expresión de la forma

```
cadena t = "Adiós" + s;
```

es un error, puesto que el operador de concatenación debe recibir como primer parámetro un objeto de tipo *cadena* (debido a que es un miembro de la clase). La solución a este problema consiste en definir **operator+** como una función externa a la clase *cadena*, haciéndola amiga de la misma para que pueda usar sus miembros privados. La versión de la clase con ésta última modificación se muestra en la figura 4.3.

El último ejemplo de este capítulo es una versión de la clase stack utilizando operadores y cambiando la representación del stack por una lista ligada.

```
# include <string.h>

class cadena {
        char *v;  // apuntador hacia los datos
        int l,    // longitud de la cadena
public:
        cadena() {  // crea una cadena nula
                v = 0;
                l = 0;
        }

        cadena(char *s) { // inicializa con una constante
                l = strlen(s);              // cadena
                v = new char[l + 1];
                strcpy( v, s );
        }

        cadena(cadena &s) {      // Este constructor se explica más tarde
                v = new char [l + 1];
                strcpy( v, s.v );
        }

        ~cadena() { // destructor
                if ( v )  delete v;
        }

        char elemento(int i); // regresa el i-esimo caracter
                                                        //  de la cadena
        cadena subcadena(int n, int m); // obtiene una
                                        // subcadena de longitud m a partir del n-ésimo caracter
        cadena asigna(cadena s), // copia el contenido de s
        cadena concatena(cadena s); // concatena con la cadena s
};

cadena cadena::asigna(cadena s) {
        if ( v )
                delete v;
        l = s.l;
        v = new char [l + 1];
        strcpy( v, s.v );
        return *this;
}
```

Figura 4.1.a. Definicion de la clase cadena, version 1

```
char cadena::elemento(int i) {
        if (i > l)
                return '\0';
        return v[i];
}

cadena cadena::subcadena(int m, int n) {
        cadena tmp;
        if ( v ) {
                tmp.l = n - m + 1;
                tmp.v = new char [tmp.l + 1];
                char *x = tmp.v, *y = v + m;
                int i = tmp.l;
                while ( i--  &&  (*x++ = *y++) )
                        ;
                if ( i == -1 )    *x = '\0';
        }
        return tmp;
}

cadena cadena::concatena(cadena s) {
        cadena tmp;
        tmp.l = l + s.l;
        cadena cadena::subcadena(int m, int n) {
        cadena tmp;
        if ( v ) {
                tmp.l = n - m + 1;
                tmp.v = new char[   tmp.l + 1],
                char *x = tmp.v, *y = v + m;
                int i = tmp.l;
                while ( i--  &&  (*x++ = *y++) )
                        ;
                if ( i == -1 )    *x = '\0';
        }
        return tmp;
}
cadena cadena::concatena(cadena s) {
        cadena tmp;
        tmp.l = l + s.l;
        tmp.v = new char[tmp.l + 1];
        strcpy( tmp.v , v );
        strcat( tmp.v , s.v );
        return tmp;
}
```

Figura 4.1.b. Definicion de la clase cadena, version 1 (continuacion)

```
#include <string.h>
class cadena {
     char *v;
     int l;
public:
     cadena () {
          v = 0;
          l = 0;
     }

     cadena (char *s) {
          l = strlen(s);
          v = new char[l + 1];
          strcpy( v, s );
     }

     cadena(cadena &s) {   // Este constructor se explica
          l = s.l;                    // más tarde
          v = new char[l + 1];
          strcpy( v, s.v );
     }

     ~cadena() {          // Destructor
          if ( v )  delete v;
     }

     cadena operator = (cadena s); // asignación
     cadena operator + (cadena s); // concatenación
     cadena operator [] (int i);   // elemento i-ésimo de
                                   // la cadena
     cadena operator () (int n, int m);   // m carácteres
                                   // después del n-ésimo
};


cadena cadena::operator = (cadena s) {
     if ( v )
          delete v;
     l = s.l;
     v = new char [ strlen( s.v ) + 1 ];
     strcpy( v, s.v );
     return *this;
}
```

Figura 4.1.a Definicion de la clase cadena, version 2.

```
char cadena::operator [] (int n) {
     if (i > l)
          return '\0';
     return v[i];
}

cadena cadena::operator () ( int m, int n ) {
     cadena tmp;
     if ( v ) {
          tmp.l = n - m +1;
          tmp.v = new char[ tmp.l + 1 ];
          char *x = tmp.v, *y = v + m;
          int i = tmp.l;
          while( i-- && ( *x++ = *y++ ) != 0 )
               ;
          if ( i == -1 )   *x = '\0';
     }
     return tmp;
}

cadena cadena::operator + (cadena s) {
     cadena tmp;
     tmp.l = l + s.l;
     tmp.v = new char [ tmp.l + 1 ];
     strcpy( tmp.v, v );
     strcat( tmp.v, s.v );
     return tmp;
```

Figura 4.2.e. Definicion de la clase cadena, version 2. (Continuación).

```
#include <string.h>

class cadena {
     char *v;
     int l;
public:
     cadena () {
          v = 0;
          l = 0;
     }

     cadena (char *s) {
          l = strlen(s);
          v = new char[l+1];
          strcpy( v, s );
     }

     cadena(const cadena &s) {
          l = s.l;
          v = new char[l + 1];
          strcpy( v, s.v );
     }

     ~cadena() {          // Destructor
          if ( v )  delete v;
     }

     cadena operator = (cadena);
     cadena operator += (cadena);
     cadena operator [] (int);
     cadena operator () ( int, int );
     operator int ();
     operator const char *();

     friend cadena operator + (cadena, cadena);
};

cadena::operator int () {
     return l;
}

cadena::operator const char *() {
     return v;
}
```

Figura 4.3.a. Definicion de la clase cadena, version 3.

```
cadena cadena::operator += (cadena s) {
     l += s.l;
     char *t = new char [l+1];
     strcpy( t, v );
     strcat( t, s.v );
     if ( v )  delete v;
     v = t;
     return *this;
}

cadena cadena::operator = (cadena s) {
     if ( v )  delete v;
     l = s.l;
     v = new char [ strlen( s.v ) + 1 ];
     strcpy( v, s.v );
     return *this;
}

char cadena::operator [] (int n) {
     if (i > l)
          return '\0';
     return v[i];
}
cadena cadena::operator () ( int m, int n ) {
     cadena tmp;
     if ( v ) {
          tmp.l = n - m +1;
          tmp.v = new char[ tmp.l + 1 ];
          char *x = tmp.v, *y = v + m;
          int i = tmp.l;
          while( i-- && ( *x++ = *y++ ) != 0 )
               ;
          if ( i == -1 )   *x = '\0';
     }
     return tmp;
}
cadena operator + (cadena s1, cadena s2) {
     cadena tmp;
     tmp.l = s1.l + s2.l;
     tmp.v = new char [ tmp.l + 1 ];
     strcpy( tmp.v, s1.v );
     strcat( tmp.v, s2.v );
     return tmp;
}
```

Figura 4.3.b. Definicion de la clase cadena, version 3. (Continuacion).

```
#include <stdió.h>
struct nodo {
     double d;
     nodo *sig;
};
class stack {
     nodo *tope;
public:
     stack();
     stack (stack &);
     ~stack();
     stack operator = (stack);          // copia de stacks
     void operator >> (double &);       // pop
     stack &operator << (double);       // push
     int operator() ();                 // verdadero si el
                                        // stack está vacío
     operator double ();                // peep
     void operator ! ();                // borra el stack
     void print();                      // despliega el
                                        // contenido del stack
};
inline stack::stack () {
     tope = 0;
}
inline int stack::operator() () {
     return tope == 0;
}
inline double stack::operator double() {
     if (tope)
          return tope->d;
     return 0;
}
stack::stack (stack &s) {
     if ( !s() ) {
          tope = new nodo;
          tope->d = s;
          for ( nodo *s1 = tope, *s = s.tope->sig; s2 ;
                s2 = s2->sig ) {
               nodo *tmp = new nodo;
               tmp->d = s2->d;
               s1->sig = tmp;
               s1 = tmp;
          }
          s1->sig = 0;    } }
```

Figura 4.4.4. Definicion de la clase stack.

```
stack::~stack () {
     while (tope) {
          nodo *tmp = tope;
          tope = tope->sig;
          delete tmp;
     }
}

void stack::operator ! () {
     while (tope) {
          nodo *tmp = tope;
          tope = tope->sig;
          delete tmp;
     }
}

stack *stack::operator << (double x) {
     nodo *tmp = new nodo;
     tmp->d = sig;
     tmp->sig = tope;
     tope = tmp;
     return *this;
}

void stack::operator >> (double &x) {
     if (tope) {
          x = tope->d;
          nodo *tmp = tope;
          tope = tope->sig;
          delete tmp;
     } else  x = 0;
}
```

Figura 4.4.b. Definicion de la clase stack. (Continuacion).

```
stack stack::operator = (stack s) {
     while (tope) {
          nodo *tmp = tope;
          tope = tope->sig;
          delete tmp;
     }
     if ( !s() ) {
          tope = new nodo;
          tope->d = s;
          for ( nodo *s1 = tope, *s = s.tope->sig; s2 ;
                s2 = s2->sig ) {
               nodo *tmp = new nodo;
               tmp->d = s2->d;
               s1->sig = tmp;
               s1 = tmp;
          }
          s1->sig = 0;
     }
     return *this;
}

void stack::print() {
     for ( nodo *t = tope; t ; t->sig )
          printf("%f ", t->d);
     printf("(NULL)\n");
}
```

Figura 4.4.c. Definicion de la clase stack. (Continuacion).

CAPITULO V

Herencia

La herencia es la base de la reutilización de código; este mecanismo permite definir una clase a partir de un conjunto de clases definidas con anterioridad; la herencia permite también diseñar clases con interfaces comunes para objetos de tipos diferentes. En este capítulo se presentan los aspectos sintácticos de C++ relacionados con la herencia.

5.1 Clases derivadas.

Supongamos que nos encontramos diseñando un conjunto de clases que serán utilizadas para desarrollar un programa de dibujo. Las clases se utilizarán para describir objetos como líneas, círculos, polígonos, curvas y otros más. Un primer intento para describir un *polígono* podría ser:

```
struct punto { // Punto de dos dimensiones
     unsigned x,y;
};

class poligono {
     // ...
     punto *v;       // Arreglo de vértices del polígono
     short n;        // Número de vértices
     punto centro;   // Coordenadas del centro
     short color;    // Código de color del polígono
public:
     // ...
     // Método de despliegue
     void dibujo();
     // Traslada x horizontalmente, y verticalmente
     void traslacion( int x, int y );
     // Rotación de x grados alrededor de c
     void rotacion( float x, punto c );
     // Cálculo del perimetro
     unsigned perimetro();
};
```

```
// Definición de los métodos para dibujo, rotación, etc.

unsigned poligono::perimetro() {
// El perímetro de un polígono es la suma de las longitudes
// de sus lados
     int p = 0;
     for ( int i = 1; i < n; i++ )
          p += sqrt( (v[i].x - v[i-1].x) * (v[i].x - v[i-1].x)
               + (v[i].y - v[i-1].y) * (v[i].y - v[i-1].y) );
     p+= sqrt( (v[n-1].x - v[0].x) * (v[n-1].x - v[0].x)
          + (v[n-1].y - v[0].y) * (v[n-1].y - v[0].y) );
     return p;
}
```

Consideremos ahora una clase para describir cuadrados. Es claro que un cuadrado es un caso especial de un polígono; seguramente esta clase puede usar los mismos métodos de dibujo, traslación y rotación que la clase *polígono*; sin embargo, un cuadrado tiene atributos especiales (como la longitud de su diagonal), propiedades especiales (tiene cuatro vértices, la longitud de sus lados es igual) y versiones especiales de algunos de sus métodos (por ejemplo, es más sencillo calcular el perímetro de un cuadrado que de un polígono). En vez de iniciar desde cero la construcción de la clase *cuadrado*, podemos aprovechar todo el código de la clase *polígono* mediante el proceso de herencia o derivación (la clase *polígono* es llamada *clase base*; la clase *cuadrado* es llamada *clase derivada*):

```
// ¡Cuidado, aún hay dos errores!
class cuadrado : poligono {  // cuadrado se deriva de polígono
     unsigned diagonal;      // longitud de la diagonal
public:
     unsigned perimetro(); //Redefinición del método perimetro
};

unsigned cuadrado::perimetro() {
// El perímetro de un cuadrado es cuatro veces la longitud de
// uno de sus lados
     return 4 * sqrt( (v[1].x - v[0].x) * (v[1].x - v[0].x)
                    + (v[1].y - v[0].y) * (v[1].y - v[0].y)
                    );
}
```

Mediante el proceso de herencia, la clase *cuadrado* adquiere todos los métodos y atributos de la clase *polígono*, además de agregar un nuevo atributo (*diagonal*) y redefinir uno de los métodos (*perimetro*). Sin embargo, los métodos privados de la clase base siguen siendo privados a ella (de otra forma, la herencia rompería el principio de ocultamiento de información). De esta manera, el método *cuadrado::perimetro* no puede accesar al atributo privado *polígono::v* (los vértices del polígono). Este problema de acceso es muy común en la programación orientada a objetos, por ello C++ incluye la palabra **protected** que puede ser usada en la declaración de la clase base para dar permiso de acceso a sus miembros a todas las clases derivadas de ella:

```
class poligono {
protected:      // Ahora todas las clases derivadas de polígono
     punto *v; // tienen acceso a v, n, centro y color
     short n;
     punto centro;
     short color;
     // ...
public:
     void dibujo();
     void traslacion( int x, int y );
     void rotacion( float x, punto c );
     unsigned perimetro();
     // ...
};
```

Sin embargo, aún subsiste un problema: por default, todos los miembros públicos de la clase base (en este caso *polígono*) son privados a la clase derivada (en este caso *cuadrado*). Entonces, ¡ningún cliente de *cuadrado* puede usar los métodos de dibujo, traslación ni rotación!:

```
main() {
     cuadrado c;
     // ...
     c.dibujo(); // error, dibujo() es privado a cuadrado
          // 'poligono::dibujo(float,punto)' is not accessible
     // ...
}
```

La solución de este problema es hacer públicos en la clase derivada todos los elementos públicos de la clase base:

```
class cuadrado : public poligono {
     unsigned diagonal;
public:
     unsigned perimetro();
};
```

También es posible utilizar **protected o private** para hacer protegidos o privados a los miembros públicos de la clase base. La figura 5.1 muestra la declaración de las dos clases anteriores.

Una característica muy importante en el proceso de herencia es que los constructores, el destructor y el operador de asignación (=) no son heredados por la clase derivada, aunque pueden ser invocados por ella. Esto se debe a que generalmente, estos métodos son más complicados en la clase base que en la clase derivada.

Retomando.....

Una clase representa un conjunto de objetos que comparten una estructura común y un mismo fin.

La *interface* de una clase enfatiza la abstracción mientras que oculta su estructura y los secretos de su propósito. Esta interface primeramente consiste de las declaraciones de todas las operaciones aplicables a las instancias de esta clase, constanes, variables y excepciones como se necesiten para completar la abstracción. Por el contrario, la *implementacion* de la clase muestra ampliamente los secretos de su propósito. La implementación de la clase primeramente consiste de la implementación de todas las operaciones definidas en la interface de la clase.

Podemos dividir la interface de la clase en tres partes:

-Public — Una declaración que forma parte de la interface de la clase y es visible a todos los clientes que son visibles a él.

-Protected — Una declaración que forma parte de la interface de la clase, pero no es visible a cualquier otra clase excepto sus subclases.

-Private — Una declaración que forma parte de la interface de una clase, pero no es visible a cualquiera otras clases.

C++ hace el mejor trabajo permitiendo a los programadores hacer distinciones explícitas entre las diferentes partes de la interface de una clase. Si es necesario, se pueden usar estructuras en C++ para definir completamente una clase encapsulada. Ada permite declaraciones públicas o privadas, pero no protegidas. En Smalltalk, Object Pascal, y CLOS, estás distinciones se logran con un convenio de programación.

5.2 Constructores.

Como se mencionó anteriormente, una clase derivada no hereda los constructores de su clase base. Sin embargo, si la clase base tiene un constructor, el constructor de la clase derivada debe invocarlo, pasándole parámetros si los necesita:

```
class x {
     // ...
public:
     x(int, char *);
     x();
};

class y : public x {
     // ...
public:
     y(int n, char *s, float x) : ( n / 2, s ) {
          // invoca a x::x(int, char *) con los argumentos
          // n/2 y s.
          // ...
     }
};

class z : public x {
     // ...
public:
     z(int n) {
          // invoca al constructor x::x()
          // ...
     }
};

main() {
     x uno(4, "Hola");   // crea un x
     y dos(5, "Adiós", 7.1);  // crea un y
     z tres(8);     // crea un z
}
```

5.3 Polimorfismo y métodos virtuales.

El polimorfismo permite manipular objetos de diferentes clases (relacionadas por herencia) como si todos pertenecieran a la misma clase. En C++, el polimorfismo sólo puede hacerse con apuntadores; los apuntadores a objetos de una clase derivada se pueden manejar como apuntadores a la clase base:

```
poligono p;
cuadrado c:
// ...
poligono *ap;  //apuntador a la clase base
ap = &p;
ap = dibujo(); // dibuja el polígono
ap = &c;
ap->dibujo();  // dibuja el cuadrado
```

Sin embargo, al tratar de invocar un método redefinido en la clase derivada se obtienen resultados inesperados:

```
ap = &c;
unsigned x = ap->perimetro(); // error, se invoca a
                              // polígono::perimetro
```

Lo que sucede aquí es que se invoca al método de la clase base, puesto que *ap* es un apuntador a esa clase. Para lograr que el compilador escoja la función adecuada, se deben calificar con la palabra **virtual** todos los métodos de la clase base que vayan a ser redefinidos en clases derivadas de ella:

```
class poligono {
     // ...
     punto *v;
     short n;
     punto centro;
     short color;
public:
```

```
        // ...
        void dibujo();
        void traslacion(int x, int y);
        void rotacion(float x, punto c);
        virtual unsigned perimetro(); // virtual permite
             // redefinir este método en las clases derivadas
             // de polígono, de tal forma que el compilador
             // pueda escoger el método adecuado al invocar a
             // perimetro con un apuntador a polígono.
};
```

Al utilizar **virtual** en la declaración de un método de la clase base, éste se puede redefinir en las clases derivadas de tal forma que, cuando se invoque utilizando un apuntador hacia la clase base, el compilador se encargue de invocar el método correcto.

Los métodos virtuales permiten definir las mismas interfaces para manipular objetos de clases diferentes (polimorfismo).

```
/*   poligono.h
*/

struct punto { // Punto de dos dimensiones
          unsigned x,y;
};

class poligono {
          // ...
protected:
          punto *v;       // Arreglo de vértices del polígono
          short n;        // Número de vértices
          punto centro;   // Coordenadas del centro
          short color;    // Código de color del polígono
public:
          // ...
          // Método de despliegue
          void dibujo();
          // Traslada x horizontalmente, y verticalmente
          void traslacion( int x, int y );
          // Rotación de x grados alrededor de c
          void rotacion( float x, punto c );
          // Cálculo del perimetro
          unsigned perimetro();
};
```

Figura 5.1. Derivación.

```
/*        poligono.cpp
*/
#include <math.h>
#include "poligono.h"

     // Definición de los métodos para dibujo, rotación,
etc.

unsigned poligono::perimetro() {
     // El perímetro de un polígono es la suma de las
longitudes
     // de sus lados
          int p = 0;
          for ( int i = 1; i < n; i++ )
               p += sqrt( (v[i].x - v[i-1].x) * (v[i].x - v[i-1].x)
                    + (v[i].y - v[i-1].y) * (v[i].y - v[i-1].y) );
          p+= sqrt( (v[n-1].x - v[0].x) * (v[n-1].x - v[0].x)
               + (v[n-1].y - v[0].y) * (v[n-1].y - v[0].y) );
          return p;
}
```

```
/*        poligonp.cpp
*/
#include "poligono.h"
#include <math.h>

class cuadrado : public poligono {
     unsigned diagonal;
public:
     unsigned perimetro();
};

unsigned cuadrado::perimetro (){

     return 4 * sqrt( (v[1].x - v[0].x) * (v[1].x - v[0].x)
                    + (v[1].y - v[0].y) * (v[1].y - v[0].y)
                    );
}

void main(){
     cuadrado d;
     d.dibujo();
```

Figura 5.1 (continuación).

CAPITULO VI

Herencia Múltiple

Este capítulo describe el concepto de herencia múltiple. La derivación de clases ofrecen un simple, flexible y eficiente mecanismo para definir una clase agregando facilidades a una clase existente sin reprogramación o recompilación. El uso de herencia múltiple ofrece una interfase común para varias clases diferentes así que los objetos de esas clases pueden manipulados por otras partes del programa. El concepto de función virtual permite a los objetos ser usados apropiadamente en contexto en los cuales su tipo no puede ser conocido en tiempo de compilación. Fundamentalmente la herencia existe para hacer al programador más fácil expresar cómodamente las relaciones entre las clases.

6.1 Herencia Múltiple

Uno de los cambios más importantes y fundamentales de la versión 2.0 de C++ es la posibilidad de crear una nueva clase derivada a partir de más de una clase base. Esto se denomina *herencia múltiple*. La herencia múltiple es especialmente útil cuando se requieren añadir características que no están relacionadas con una jerarquía a una clase que forma parte de esa jerarquía. Por ejemplo, se pude crear una clase que sea parte de una jerarquía de "transporte", y que tenga la capacidad de almacenarse así misma heredándose desde la **class storable**. La única forma de llevar a cabo esto es a través de la herencia múltiple.

La sintaxis para una herencia múltiple es muy sencilla. Con herencia simple sería:

```
class bc {
     int i;
public:
     bc();
};

class dc : public bc {
     // ...
};
```

Con herencia múltiple sería:

```
class bc1 {
     // ...
};

class bc2 {
     // ...
};
```

```
class bc3 {
     // ...
};

class midc : public bc1, public bc2, public bc3 {
     // ...
};
```

Al igual que con la herencia simple, si no se especifica el acceso a los elementos de la clase base con la palabra reservada **public**, toma por omisión **private**. No se puede utilizar **protected** para especificar el acceso a la clase base.

No se puede especificar una clase en la lista de las clases base más de una vez. El orden de las clases en la lista determina el orden de la inicialización si no se especifica ésta en el constructor (es decir, si es el compilador el que se encarga de llamar a los constructores). El orden de destrucción de las clases base es siempre el inverso al de su declaración.

6.2 Constructores

Cuando se utilizaba una herencia simple, era redundante dar el nombre de la clase base en la lista del inicializador del constructor, por lo que no se hacía. Por ejemplo:

```
class bc {
     int i;
public:
     bc (int x = 0) { i = x };
};

class dc : public  bc {

public:
     dc (int y = 0) : ( y ) {}      // llamada al constructor
                                    // de la clase base
};
```

Aunque esta sintaxis para la herencia simple se acepta en la versión 2.0, no es recomendable. Sin embargo, cuando se utiliza la herencia múltiple, esta sintaxis es ambigua, ya que el compilador no sabe de que clase base se está hablando. Para resolver este problema, se debe especificar el nombre de la clase base en la lista del inicializador del constructor.

```
class bc1 {

public:
     bc1(int x);
};

class bc2 {

public:
```

```
        bc2(int x);
    };


    class mdc : public bc1, public bc2 {

    public:
        // Los nombres de las clases base se dan en la lista de
        // inicialización.
        mdc(int a, int b) : bc1( a ), bc2( b ){}
    };
```

En otros compiladores también se puede dar el nombre de la clase base en la lista del inicializador del constructor para una herencia simple.

6.3 Clases base virtuales

Al mismo tiempo que la herencia múltiple aumenta en gran medida las posibilidades de C++, también puede introducir algunas ambigüedades. En concreto, ¿qué ocurriría si dos clases comparten la misma clase base y ambas se heredan en una clase derivada simple? Para comprender este problema es necesaria una nueva terminología.

Una *clase base directa* es aquella que está en la lista de las clases base para una clase derivada en particular. Por ejemplo:

```
class dc : public bc1, public bc2, public bc3 { // ... };
```

las clases **bc1**, **bc2** y **bc3** son clases base directas. Una *clase base indirecta* es una clase que no aparece en la lista anterior de clases base, peor está heredada en una o más clases en la lista. Por ejemplo:

```
class ibc {

public:

     void print();
};

class dbc1 : public ibc {};
class dbc2 : public ibc {};
class der : public dbc1, public dbc2 {

     public:
          void print_der () { print(); } // ambiguo
};
```

Aquí, **dbc1** y **dbc2** son clases base directas y **ibc** es una clase base indirecta. Observe que debido a que **ibc** se utiliza tanto en

dbc1 como **dbc2**, la llamada a la función **ibc::print();** es ambigua en der::print_der(). El problema se produce cuando se crea un objeto **class der**, contiene dos *subobjetos* de **class ibc**. El compilador no sabe cuál utilizar y se genera un mensaje de error.

Este problema puede resolverse de dos formas distintas. La primera forma es especificar el subobjeto de la clase base utilizando un operador de resolución de ámbito. Por ejemplo:

```
void der::print_der(){ dbc1::print(); } // sin ambigüedad
```

Sin embargo, el usuario de una clase derivada quizás no tenga suficientes conocimientos sobre las clases base para hacer esto, además de resultar muy confuso. La otra forma de hacerlo, es que el diseñador de clase debe declarar la clase base común **virtual** cuando se está derivando en las clases base directas. Por ejemplo:

```
class dbc1 : virtual public ibc {};

class dbc2 : virtual public ibc {};

class der : public dbc1, public dbc2 {

     public:
          void print_der(){ print(); }  // no ambiguo
};
```

Aquí sólo hay un subobjeto de **class ibc**, por lo que llamar a **print()** no es ambiguo, **dbc1** y **dbc2** comparten el subobjeto de **class ibc**.

Se pueden tener subobjetos virtuales y no virtuales en la misma clase. Si se crea otra clase base directa *sin* utilizar la palabra reservada **virtual** cuando se hereda de **class ibc;**

```
class sbc3 : public ibc {}
```

Y esta clase base también se utiliza en **class der,**

```
class der : public debc1, public debc2, public debc3 {

public:
     void print_der() { print(); }  // ambiguo otra vez!
};
```

se tiene otra ambigüedad debido a que hay dos subobjetos de **class ibc;** uno en **dbc3** y otro compartido por **dbc1** y **dbc2.**

El compilador comprueba si hay ambigüedades antes de ver si el acceso es correcto y de comprobar los tipos. Si se descubre alguna ambigüedad, se genera un mensaje de error. Se deben eliminar todas las ambigüedades, ya sea calificando un miembro con su nombre de clase o utilizando clases base **virtual.**

CAPITULO VII

Entrada/Salida en C++

What you see is all you get.
- Brian Kernighan

El lenguaje C++ no ofrece facilidades para la entrada/salida. No lo necesita; tales facilidades pueden ser simple y elegantemente creadas usando el lenguaje mismo. Las librerías estándar de entrada/salida descritas aquí ofrecen tipización segura, flexible y un método eficiente para el manejo de caracteres de entrada y enteros de salida, números de punto flotante, y cadenas de caracteres, un modelo que permite manejar tipos definidos por el usuario. Las interfases pueden ser encontradas en <iostream.h>.

7.1 Salida

Una tipización segura y tratamiento uniforme de los tipos incorporados y los definidos por el usuario pueden ser realizados usando una sencilla función sobrecargada para un conjunto de funciones de salida.

Por ejemplo:

```
put(cerr,"x = ");    // cerr es el objeto para la salida de
                     // error
put(cerr,x);
put(cerr,'\n');
```

El tipo de argumento determina que función put será invocada para cada argumento. Esta solución ha sido usada en diferentes lenguajes. Sin embargo, depende del contexto. La sobrecarga del operador << para que diga "colócalo en" da una mejor notación y permite al programador imprimir una secuencia de objetos en un simple enunciado, por ejemplo:

```
cerr << " x = " << x << '\n';
```

donde cerr es el objeto para la salida de error. Así, sí **x** es un entero con valor 123, este enunciado imprimirá

```
x = 123
```

y un carácter nueva línea en la salida de error estándar. Similarmente, sí **x** es un tipo defino por el usuario, como complejo con el valor (1,2.4), el enunciado de arriba imprimirá

```
x = (1,2.4)
```

en cerr. Este estilo de programación puede ser usado tanto como **x** sea de un tipo para el cual el operador << esta redefinido, un usuario puede redefinir en forma trivial el operador << para el nuevo tipo.

Los operadores << y >> fueron redefinidos también para la entrada y salida estándar. Ellos son asimétricos en el sentido de que pueden ser usados para sugerir "hacia" y "de", además ellos no son lo operadores más utilizados en los tipos de datos incorporados; y la precedencia de << es más baja de formar que permite operaciones aritméticas y operandos sin el uso de paréntesis. Por ejemplo:

```
cout << "a*b+c=" << a*b+c << '\n';
```

Los paréntesis deben ser usados para escribir expresiones que contengan operadores con una precedencia menor. Por ejemplo:

```
cout << "a^b|c = " << (a^b|c) << '\n';
```

El operador de corrimiento de bits puede ser usado en el contexto anterior, por supuesto, usando paréntesis.

```
cout << "a<<b=" << (a<<b) << '\n';
```

7.2 Salida de tipos incorporados

La clase **ostream** esta definida con el operador << para manipular la salida de tipos incorporados:

```
class ostream : public virtual ios {
     // ...
public:
     ostream & operator<<(const char *); // cadenas
     ostream & operator<<(const char);
     ostream & operator<<(short i)
               { return *this << int(i) };
     ostream & operator<<(int);
     ostream & operator<<(double);
     ostream & operator<<(const void *);
     ostream & operator<<(const char *);  // apuntadores
     // ...
};
```

Naturalmente, ostream también tiene un conjunto de funciones operator <<() para tipos unsigned.

Una función operador << regresa una referencia al ostream para el cual fue llamado, de tal forma que otro operador << pueda aplicársele, por ejemplo:

```
cerr << " x = " << x;
```

donde x es un entero, será interpretado como:

```
(cerr.operator<<("x = ")).operator<<(x);
```

Esto quiere decir que los argumentos se imprimen de izquierda a derecha.

La función **ostream::operator<<(int)** imprime valores enteros y **ostream::operator<<(char)** imprime caracteres. Por ejemplo:

```
void val(char c)
{
     cout << "int('" << c << "') = " << int(c) << '\n';
}
```

imprimiendo valores enteros de los caracteres:

```
main()
{
     val('A');
     val('Z');
}
```

imprimirá

```
int('A') = 65
int('Z') = 90
```

Todo esto, si su máquina utiliza el código ASCII. Note que un carácter literal tiene tipo char así que cout << 'Z' imprimirá la letra Z y no el valor entero 90.

6.2 Salida de tipos definidos por el usuario.

Consideremos la clase complejo vista con anterioridad:

```
class complejo {
     double re,im;
public:
     complejo(double r = 0, double i = 0) {
               re = r; im = i }
     friend complejo operator +(complejo, complejo);
     friend complejo operator +(complejo, complejo);
     friend complejo operator +(complejo, complejo);
     friend complejo operator +(complejo, complejo);
     // ...
     friend ostream & operator << ( ostream &, complejo)
};
```

El operador << puede redefinirse para el nuevo tipo complejo como sigue:

```
ostream & operator << ( ostream &s, complejo z)
{
     return s << '(' << re << ',' << im << ')' ;
}
```

Y usado como un tipo incorporado:

```
main() {
     complejo x(1,2);
     cout << "x = " << x << '\n';
}
```

producirá

```
x = (1,2)
```

6.4 Salida

La salida es similar a la entrada. Existe una clase **istream** que ofrece un pequeño conjunto de funciones sobrecargadas con el operador >>. Una función operator >> puede entonces ser definida para un tipo definido por el usuario.

La clase istream está definida como sigue:

```
class istream : public virtual ios {
     // ...
public:
     istream& operator>>(char *);
     istream& operator>>(char &);
     istream& operator>>(short &);
     istream& operator>>(int &);
     istream& operator>>(long &);
     istream& operator>>(float &);
     istream& operator>>(double &);
     // ...
};
```

Las funciones operator >> están definidas con el siguiente estilo:

```
istream& istream::operator>>(T& tvar)

{
     // salta espacios en blanco
     // alguna forma de leer a T en la variable 'tvar'

     return *this;
}
```

Alternativamente, se pueden usar las funciones get():

```
class istream : public virtual ios {
     // ...
     istream get(char & c);              // char
     istream& get(char* p, int n, char='\n');  // cadenas
};
```

Se pueden usar de la siguiente forma:

```
main()
{
     char c;
     while(cin.get(c))
          cout << c;
}
```

Así como cin tiene la función get, cout tiene la función put la cual trabaja de la siguiente forma:

```
main()
{
     char c;
     while(cin.get(c))
          cout.put(c);
}
```

La segunda función get() lee al menos *n* caracteres en el arreglo de caracteres apuntado por *p*. Una llamada a get() siempre pondrá un 0 al final de los caracteres (si existen) colocados en el búfer, así que al menos n-1 caracteres son leídos dando n como segundo argumento. El tercer argumento especifica el terminador. Por ejemplo:

```
void f()
{
     char buf[100];

     cin >> buf;              // sospechoso
```

```
        cin.get(buf,100,'\n');   // seguro
        // ...
}
```

La instrucción cin>>buf es sospechosa por que una cadena de más de 99 caracteres causa un overflow del búfer. Si el terminador es hallado, este se deja como el primer carácter no leído en la entrada. Esto permite checar por búfer overflow.

```
void f()
{
        char buf[100];
        cin.get(buf,100,'\n');   // seguro
        char c;
        if ( cin.get(c) && c!;'\n'){
                // la cadena de entrada es más grande de lo esperado
        }
        // ...
}
```

Naturalmente, existen versiones de get() para unsigned chars.

El archivo de encabezado <ctype.h> define varias funciones que pueden ser usadas con éxito cuando se procesa la entrada. Por ejemplo, una función que se "coma" los espacios en blanco de la entrada estándar podría ser definida como:

```
istream& eatwhite(istream& is)
{
        char c;
        while ( is.get(c) ){
                if ( isspace(c)==0) {
                        is.putback(c);
                        break;
                }
        }
        return is;
}
```

6.5 Estado de los Streams

Cada *stream* (istream ó ostream) tienen un *estado* asociado. Los errores y las condiciones no estándares son manejadas poniendo y probando este estado apropiadamente.

El estado del stream puede ser examinado por operaciones en la clase ios:

```
class ios { // ios es la base de ostream e istream
     // ...
public:
     int eof() const;    // se alcanzo EOF
     int fail() const;   // sig. operación fallará
     int bad() const;    // stream corrupto
     int good() const;   // sig. operación tendrá éxito
     // ...
};
```

Si el estado es good() o eof(), la operación previa de entrada tuvo éxito. Sí el estado es good(), la próxima operación tendrá éxito, de otra forma tendrá falla. Aplicando una operación de entrada a un stream que no esta en el estado good(), es una operación nula. Sí se intenta leer en una variable v y la operación falla, el valor de v debe permanecer sin cambios (no se altera, si v es uno de los tipos manejados por las funciones miembro de istream o ostream).

Los valores usados para representar estos estados están también definidos en la clase ios:

```
class ios {
     // ...
public:
     enum io_state {
          goodbit = 0,
          eofbit = 1,
```

```
            failbit = 2,
            badbit = 4,
        };
        // ...
};
```

6.6 Entrada de tipos definidos por el usuario.

Una operación de entrada puede ser definida para tipos definidos por el usuario de la misma forma que la operación de salida fue definida, pero en una operación de entrada es esencial que el segundo argumento sea una referencia. Por ejemplo:

```
istream& operator >>(istream& s, complejo& a)
/*
     Lee un complejo en el siguiente formato; donde "f" indica
     un float.

     f
     ( f )
     ( f , f )
*/

{
     double re = 0, im = 0;
     char c = 0;

     s >> c;
     if ( c=='(' ){
          s >> re >> c;
          if ( c== ',' )
               s >> im >> c;
          if ( c!= ')' )
               s.clear(ios::badbit);    // cambia el estado
     }
     else {
          s.putback(c);
          s >> re;
     }

     if ( s ) a = complejo( re,im );
     return s;
}
```

La variable local **c** es inicializada para evitar tener un valor accidental '(' después de una operación fallida. El último if es para asegurárse que el valor de **a** será modificado solamente si la lectura tuvo éxito.

La operación para cambiar el estado del stream es llamada clear() por que es más común usarla para restaurar el estado del stream a good()(; ios:goodbit es el valor de default para ios::clear().

Los siguientes programas ilustran los conceptos anteriores.

```
// Uso de cin.
#include <iostream.h>

void main () {
     int i;
     float f;

     cout << "Escribe dos números: ";
     cin >> i >> f; // Escribir 22 b 13.57

     cout << "Los número leídos son " << i << " y " << f << '\n';

     char s[12];

     cin.getline (s, 12);     // Extrae el resto del buffer hasta
'\n'

     cout << "El resto del buffer es: " << s << ".\n";
}
```

```
#include <iostream.h>

class Point {

      float x, y, z;

public:

      Point(float i, float j, float k) {
        x = i; y = j; z = k;
      }

      friend ostream& operator<< (ostream& os, Point& p);
      friend istream& operator>> (istream& os, Point& p);
};

ostream& operator<< (ostream& os, Point& p)
{
      return os << '('
                << p.x << " "
                << p.y << " "
                << p.z << ")";
}

istream& operator>> (istream& is, Point& p)
{
      return is >> p.x >> p.y >> p.z;
}


void main()
{
      Point p(2, 3, 5);
      cout << "\n\nLas coordenadas de p son " << p;

      // obtener nuevas coordenadas
      cout << "\n¿Nuevas coordenadas? ";
      cin >> p;

      // Escribe las nuevas coordenadas
      cout << "\nLas coordenadas de p son" << p;
}
```

```
#include <iostream.h>
#include <strstrea.h>
#include <iomanip.h>

void main()
{
      char numbers [] = "\n  10 010 0x10";

      // asocia la cadena con un stream
      istrstream is(numbers, sizeof(numbers) );

      // extrae del stream usando diferentes bases
      int v1, v2, v3;
      is >> v1 >> v2 >> v3;

      // despliega el resultado
      cout << "\n" << v1 << " " << v2 << " " << v3;
}
```

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

void main()
{
      float v1 = 1300.23;
      float v2 = 320.99;
      float v3 = 54430.00;

      cout << setiosflags(ios::showpoint | ios::fixed)
        << setprecision(2)
        << setfill('*')
        << setiosflags(ios::right);
      cout << "\n$" << setw(10) << v1;
      cout << "\n$" << setw(10) << v2;
      cout << "\n$" << setw(10) << v3;
}
```

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

void main()
{
      // abre un archivo existente
      ifstream help_file("C:\\autoexec.bat");
      // se fija si ha ocurrido un error
      if (!help_file) {
     cout << "\nNo pude abrir el archivo AUTOEXEC.BAT";
     return;
      }

      // Despliega el archivo
      while (help_file) {
     char buffer [100];
     help_file.getline(buffer, sizeof(buffer) );
     cout << "\n" << buffer;
      }
}
```

```
#include <iostream.h>
#include <fstream.h>
void main()
{
      ifstream help_file("\\BORLANDC\\README.    ");

      // abre un archivo y lo trunca
      ofstream copy_file("COPY");

      // se fija si han ocurrido errores
      if (!help_file) {
     cout << "\nNo pude abrir \\BORLANDC\\README";
     return;
      }
      if (!copy_file) {
     cout << "\nNo pude abrir el archivo COPY";
     return;
      }

      int line_count = 0;
      while (help_file && copy_file) {
     char buffer [80];
     help_file.getline(buffer, sizeof(buffer) );
     copy_file << buffer << "\n";
     if (++line_count == 4) break;
      }
}
```

```
#include <strstrea.h>
#include <iomanip.h>

void main()
{
      char buffer [80];
      ostrstream text(buffer, sizeof(buffer) );
      int i = 10;
      char* code = "No sé";

      text << "\nHay "
        << i
        << " objectos"
        << ends;

      cout << buffer;

      // regresa el apuntador de inserción del stream
      // antes de usarlo nuevamente
      text.seekp(0);
      text << "\nEl error ocurrido es: "
        << code
        << ends;
      cout << buffer;
}
```

eso, es todo.

CAPITULO VIII

Templates

Este capítulo introduce el concepto de *template*. Un template es una función o clase que permite manipular los distintos tipos de datos incorporados y abstractos. Un template permite crear funciones genéricas, tales como *sort()*, la cual se define una sola vez para una familia de tipos.

8.1 Un template simple.

La definición de una *clase template* especifica como clases individuales pueden ser construidas mucho mejor, como una declaración de clase que especifica como los objetos individuales pueden ser construidos. Definamos una clase template sencilla:

```
/*              _temple.cpp
*/

#include <iostream.h>

template  class T>
class A {
        T x;
public:
        void read(){
                cout << "\n      Dame un valor para x ";
                cin >> x;
        }
        void print(){
                cout << "\n-- es : " << x << endl;
        }
};

void main(){
        {
        A<char> 1;
        1.read();
        1.print();
        }

        {
        A<int  1;
        1.read();
        1.print();
        }
}
```

En este ejemplo, podemos observar dos cosas; la forma de declarar una clase template, y la forma de crear una instancia de la clase.

La sintaxis para la declaración es:

template <class Nombre> función|clase {...};

Donde *Nombre* es el nombre real del template.

La sintaxis para crear una instancia es:

Nom_clase_temple**<tipo>** objeto

Donde *tipo* es un tipo incorporado o abstracto.

Es importante escribir *templates* que tengan pocas dependencias globales, tanto como sea posible. La razón es que un template será usado para generar funciones y clases basadas en **tipos** y **contextos** desconocidos.

Para crear un template, es conveniente que primero se defina la clase y se prueben sus métodos; de tal manera que la clase quede bien definida y depurada. Una vez creada esta clase se procede a generalizarla, es decir, a convertirla en una **clase template**.

Siguiendo la idea anterior, definamos a la clase stack (vista en el capítulo III) como un template.

```
// STACKT.H: Encabezado para la clase stack template.

#ifndef STACKT_H
#define STACKT_H

template <class T>
class stac  {
     int n;          // Número de elementos
     int nmax;       // Número máximo de elementos
     T *d;      // Apuntador a los datos
public:
     stack (int max = 10);
     ~stack();

     T push(T);      // inserta un número en el stack
     T pop(void);         // obtiene y borra el primer elemento
     T peep(void);        // obtiene el primer elemento
     void limpia(void);        // inicializa el stack
};

template<class T> inline void stack<T>::limpia(void) {
     n = 0;
}

#endif STACKT_H
```

```
//STACKT.CPP: Métodos de la clase stack template

#include <iostream.h>
#include "stackt.h"

template <class T> stack<T>::stack (int i) {
        n = 0;
        nmax = i;
        s = new T[nmax];
}

template <class T> stack<T>::~stack () {
        delete s;
}

template <class T> T stack<T>::push(T f) {
        if (n == nmax) {
                cerr << "Error: stack lleno.\n" << endl;
                return 0;
        } else {
                s[n++] = f;
                return f;
        }
}

template <class T> T stack<T>::pop(void) {
        if (n == 0) {
                cerr << "Error: stack vacio.\n";
                return 0;
        } else {
                return s[--n];
        }
}

template <class T> T stack<T>::peep(void) {
        if (n == 0) {
                cerr << "(peep) Error: stack vacio.\n";
                return 0;
        } else
                return s[n-1];
}


void main(){

        stack<int> i(30);

        stack<char> cp;

        i.push(5);
        cp.push('a');

}
```

El prefijo *template <class T>* especifica que un template está siendo declarado y que un argumento T de tipo *type* será usada en la declaración. Después de esta instrucción, T es usada exactamente como cualquier otro tipo. El ámbito de T se extiende al final de la declaración del prefijo *template <class T>*.

Note que template <class T> dice que T es el nombre de un *tipo;* este no necesita ser actualmente el nombre de una clase. Para el objeto *i* del ejemplo anterior, T se vuelve *int*.

Las funciones *template* no necesitan ser en linea, como en el primer ejemplo. Se puede observar en la clase template <class T> que los métodos están declarados fuera de la clase y que se asocian con su respectiva clase con el operador de ámbito :: y la definición del template. Como en:

```
template <class T> T stack<T>::push(T f) {
// ...
}
template <class T> stack<T>::stack (int i) {
// ...
}
```

Una vez dada la definición de la clase template, stack puede ser ahora definida y usada como:

```
     stack<shape*> ssp(200);  // Stack de apuntadores a shape
     stack<Point> sp(400);    // Stack de Points

     void f(stack<complejo>& sc ) // argumento 'referencia a un
                              // stack de complejos'
     {
          sc.push(complejo(1,2));
          complejo z = 2.5 * sc.pop();

          stack<int>*p = 0;    // ap. a un stack de ints
          p = new stack<int>(800); // stack de ints en almace-   //
namiento libre
          for (int i = 0; i < 400; ++i) {
               p->push(i);
               sp.push(Point (i,i+400));
          }
          // ...
     }
```

8.2 Funciones Template.

No se aplican las reglas de conversión en los argumentos de la función template. En su lugar, nuevas versiones son generadas tanto como sea posible. Por ejemplo:

```
template<class T> T sqrt(T);

void f(int i, double d, complejo z)
{
     complejo z1 = sqrt(i);   // sqrt( int )
     complejo z2 = sqrt(d);   // sqrt( double )
     complejo z3 = sqrt(z);   // sqrt( complejo )
     // ...
}
```

Esto generará una función *sqrt* del template para cada uno de los tres tipos de argumentos. Si el usuario quiere algo distinto, digamos una llamada a sqrt(double) dando un argumento int, una conversión de tipo explícita será usada.

```
template<class T> T sqrt(T);

void f(int i, double d, complejo z)
{
     complejo z1 = sqrt(double(i)); // sqrt( double )
     complejo z2 = sqrt(d);   // sqrt( double )
     complejo z3 = sqrt(z);   // sqrt( complejo )
     // ...
}
```

Aquí, solamente se genera código para sqrt(double) y sqrt(complejo) del template.

Una función template puede ser sobrecargada tanto por otras

funciones con el mismo nombre o por otras funciones templates con el mismo nombre. La resolución de las funciones template sobrecargadas y otras funciones del mismo nombre es hecha en tres pasos:

[1] Se busca una concordancia exacta en las funciones; si se encuentra, se llama.

[2] Se busca una función template de la cual, una función que puede ser llamada con una concordancia exacta puede ser generada; si se encuentra, se llama.

[3] Se intenta una sobrecarga ordinaria para las funciones; si una función es encontrada, se llama. Si ninguna concordancia es encontrada la llamada es un error.

En cada caso, si existe más de una alternativa, la llamada ambigua y es un error. Por ejemplo:

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

void f(int a, int b, char c, char d)
{
     int  m1 = max(a,b);       // max(int ,int)
     char m2 = max(c,d);       // max(char,char)
     int  m3 = max(a,c);       // error, no se puede
                               // generar max(int ,char)
}
```

Debido a que las conversiones no son aplicadas antes de seleccionar una función template para generar y llamar (regla 2), la última llamada no puede ser resuelta a max(a,int(c)). El programador puede resolver el problema declarando una función max(int,int). Esto trae a la regla 3 en acción:

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


int max(int, int);


void f(int a, int b, char c, char d)
{
     int  m1 = max(a,b);       // max(int ,int)
     char m2 = max(c,d);       // max(char,char)
     int  m3 = max(a,c);       // max(int ,int)
}
```

No existe la necesidad de declarar max(int,int); ya que será generada del template, por default.

Finalmente, la función max() conbinada con la clase cadena quedaría así:

```
/*          maxt.cpp
*/

#include <iostream.h>
#include "cadena.h"

template <class T1,class T2>
T1 max(T1 a, T2 b){
     return ( a>b ? a : b );
};

void f(int a, int b, char c, char d)
{
     int  m1 = max(a,b);       // int    max(int ,int)
     char m2 = max(c,d);       // char max(char,char)
     int  m3 = max(a,c);       // int    max(int ,char)

     cout << "\n\tm1 = " << m1;
     cout << "\n\tm2 = " << m2;
     cout << "\n\tm3 = " << m3;
}

void main () {

     f( 5,6,'a','b' );

     cadena a="ala",b="alameda";
     cout << "\n";
     cout << (a > b ? a : b );     // operator > (a,b)
     cout << "\n";
     cout << max(a,b);

}
//   Recuerde abrir un Project e incluir los archivos maxt.cpp y
cadena.cpp
```

CAPITULO X

METODOLOGIAS DE DISEÑO

Actualmente las empresas se enfrentan a un profundo dilema. Cada vez más se están convirtiendo en organizaciones basadas en información, dependiendo de un flujo continuo de datos para, virtualmente, cualquier aspecto de sus operaciones. Sin embargo, su habilidad para manejar esos datos se ve disminuida porque el volumen de información se expande más rápido que su capacidad de procesarla. El resultado de ello es que las empresas prácticamente se están ahogando en sus propios datos, y el problema no está en el hardware; las computadoras continúan aumentando su capacidad y potencia a una velocidad impresionante. La falla se debe al software. *Desarrollar software que iguale el potencial de las computadoras resulta ser un reto mucho mayor que el construir máquinas más rápidas.*

10.1 Introducción

Este capítulo discute los tópicos relacionados con la construcción del Software. La discusión cubre los aspectos técnicos y sociológicos del desarrollo de Software. Un programa es visto como un modelo de la realidad, donde cada clase representa un concepto. La llave de la tarea del desarrollo es especificar las interfaces públicas y protegidas que definen las diferentes partes del programa. Definiendo estas interfaces, se vuelve un proceso iterativo que típicamente requiere experiencia.

10.2 La Crisis del Software

La diferencia de potencial entre el hardware y el rendimiento del software se está ampliando continuamente. Este potencial gastado afecta a cualquiera que usa una computadora, sin embargo, lo hace con especial énfasis en las grandes organizaciones, las que dependen ampliamente de su habilidad para construir sistemas de información confiables de gran escala. Hoy en día son contados los proyectos de integración que se terminan a tiempo y mucho menos aquellos que lo hacen dentro de lo previsto. Peor aún, es típico que los sistemas creados con estos esfuerzos estén plagados de defectos, y han sido tan rígidamente estructurados que es casi imposible realizar cambios significativos sin tener que rediseñarlos. Combinemos estos problemas con la incremental propensión al cambio en las condiciones de las organizaciones y *tendremos una receta para el desastre.*

La mayoría del software corporativo es obsoleto antes de liberarse y frecuentemente es incapaz de evolucionar para satisfacer necesidades futuras. Estudios al respecto indican que en algunos casos tan sólo el cinco por ciento de los proyectos de software terminan en sistemas funcionales; el resto se regresa para su reconstrucción, se abandona después de liberarse, o nunca es

completado.

Esta situación se conoce en la industria como "La Crisis del Software", y es un problema de tamaño considerable que amenaza la viabilidad de las organizaciones actuales basadas en la información. Resolver esta crisis se ha convertido rápidamente en una preocupación de las empresas a nivel mundial.

10.3 Cómo se construye el Software

Al revisar lo que se ha intentado anteriormente entenderemos en qué es diferente el enfoque Orientado a Objetos (OO) y por qué debe ser exitoso donde los demás han fallado.

10.3.1 Construyendo Programas

Un programa no es más que una serie de instrucciones que le dicen a la computadora que lleve a cabo acciones específicas. Los programas pequeños pueden construirlos un programador como una sola secuencia de instrucciones que realizan la tarea deseada. Los programas de mayor tamaño no pueden construirse así y, en principio, la solución a este problema es evidente: descomponer los programas grandes en componentes pequeños que puedan construirse independientemente y después combinarlos para formar el sistema completo. Esta estrategia general es conocida como "Programación Modular", y forma el principio en que están basados la mayoría de los avances en la contrucción del software en los últimos 40 años.

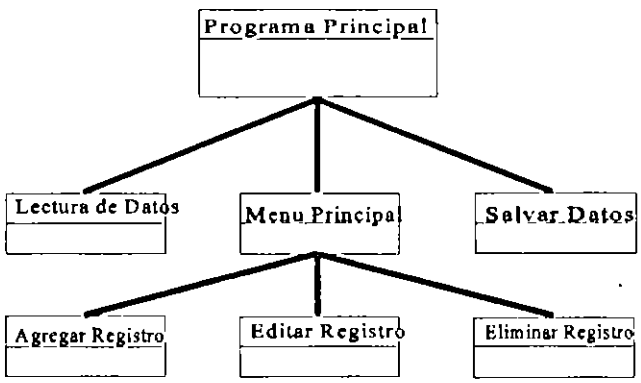
10.3.2 Programación Modular

El soporte más elemental para la programación modular se dió con la invención de la "Subrutina" a principios de los 50's. Una subrutina se crea al sacar una secuencia de instrucciones del programa principal y darle un nombre separado. Mientras que las subrutinas proporcionan el mecanismo básico para la programación modular, se requiere mucha disciplina para crear software bien

estructurado. Sin dicha disciplina es muy fácil escribir programas complicados que se resistan al cambio, difíciles de entender y prácticamente imposibles de mantener. Eso es lo que con frecuencia pasó en los primeros años de la industria.

10.3.3 Programación Estructurada

A finales de los 60's, el pobre estado del software disparó un esfuerzo concertado entre los científicos de la computación para desarrollar un estilo de programación consistente y disciplinado. El resultado de ese esfuerzo fué el refinamiento de la programación modular en el enfoque conocido como "Programación Estructurada", que se basa en la descomposición funcional al descomponer sistemáticamente un programa en componentes, cada uno de los cuales se descompone en subcomponentes y así sucesivamente hasta el nivel de subrutinas individuales. De esta manera, grupos de programadores separados escriben diferentes componentes, los que ensamblan posteriormente para formar el sistema completo.



Programación Estructurada

La programación estructurada ha producido mejoras significativas en la calidad del software en los últimos veinte años, pero sus limitaciones son penosamente aparentes hoy en día. Uno de los problemas más serios es que rara vez es posible anticipar el diseño completo de un sistema antes de que sea

implantado. Entre mayor sea el sistema, es más frecuente que se requiera reestructurarlo.

10.3.4 Ingeniería de Software Asistida por Computadora(CASE)

La última innovación en la programación estructurada es la "Ingeniería de Software Asistida por Computadora" (CASE). Con CASE las computadoras administran el proceso de la descomposición funcional verificando que todas las interacciones entre subrutinas sigan una forma correcta y específica. De hecho, los sistemas avanzados de CASE pueden construir programas completos a partir de diagramas en los que se expresa toda la información del diseño. Sin embargo, la experiencia a la fecha ha demostrado que desarrollar un diseño gráfico para un programa puede ser tan demandante y consumir tanto tiempo como el escribir dicho programa.

10.3.5 Lenguajes de Cuarta Generación

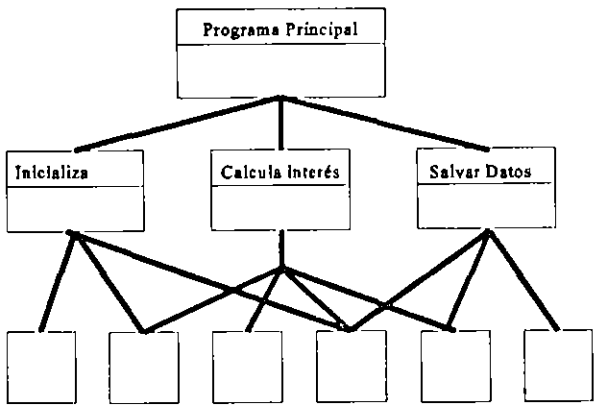
Otro enfoque de programación automática lo representan los "Lenguajes de Cuarta Generación" (4GL's). Los 4GL's incluyen una amplia variedad de herramientas que ayudan a automatizar la generación de aplicaciones típicas de negocios, incluyendo la creación de formas, reportes y menús. Los 4GL's ofrecen muchas ventajas, incluyendo el hecho de que la gente que no sea programadora los puede utilizar. Utiles, como son los 4GL's, son dejados de lado una vez que se atraviesa el umbral de las aplicaciones complejas.

10.3.6 Administrando la Información

La mayoría de los esfuerzos para mejorar el desarrollo del Software se han enfocado en la modularización de los procedimientos. Pero hay otro componente del Software que no por ser menos obvio es menos importante: los datos, es decir, la colección de información sobre la que operan los procedimientos.

Cuando la cantidad de unidades de datos va más allá de los cientos o miles, el hecho de permitir que diferentes rutinas los accesen, frecuentemente conduce a errores misteriosos y comportamiento impredecible, dado que siempre existe la posibilidad de que una persona cambie la información que otros están utilizando actualmente. Prevenir esta confusión resulta ser un problema técnico que no es fácilmente resuelto en los sistemas de archivos simples, para ello se utilizan programas especializados llamados "Sistemas Administradores de Bases de Datos" (DBMS), los cuales están diseñados para administrar el acceso simultáneo a datos compartidos.

Múltiples Subrutinas compartiendo datos



Administrando la Información

Los primeros modelos de bases de datos fueron el "jerárquico" y el de "red", los cuales facilitan la representación de relaciones complejas entre las unidades de datos almacenadas, pero había un costo: accesar los datos en una forma diferente a la soportada por las relaciones predefinidas era lento e ineficiente. Peor aún, las estructuras de datos eran difíciles de modificar, y cambiar esas estructuras requería que el administrador del sistema apagara la base de datos y la reconstruyera.

La forma de administración de bases de datos más aceptada actualmente es el modelo "relacional", que se avoca a la solución de estos problemas removiendo la información de relaciones complejas de la base de datos. Todas la unidades de datos se almacenan como tablas simples y las relaciones entre ellas también se definen como tablas. Sin embargo, y a pesar de que el modelo relacional es mucho más flexible que sus predecesores, pagan precio por esta flexibilidad: La información sobre relaciones complejas tiene que ser expresada como procedimientos en cada programa que accesada la base de datos, y a mayor complejidad corresponde mayor penalización en los tiempos de acceso, ya que las estructuras de datos deseadas tienen que ensamblarse cada vez que se accesen los datos.

10.4 El enfoque de la Orientación a Objetos

A pesar de los esfuerzos para encontrar la mejor forma de construir programas, la crisis del Software empeora cada año, y cuarenta años después de la invención de la subrutina seguimos construyendo sistemas a mano, una instrucción a la vez. Hemos desarrollado mejores métodos para este proceso de construcción, los que no funcionan bien en sistemas grandes y complejos. Adicionalmente, estos métodos producen Software plagado de defectos que es difícil de modificar y mantener. Necesitamos un nuevo enfoque para construir Software, uno que deje atrás los métodos de la programación convencional y ofrezca una mejor forma de construir sistemas pequeños y de gran escala que sean confiables, flexibles, mantenibles y capaces de evolucionar para satisfacer los requerimientos del cambio.

A continuación veremos como funciona la técnología de programación orientada a objetos (POO) y por qué tiene el potencial de tener éxito donde los otros métodos han fallado.

10.4.1 Elementos Claves para Entender la Tecnología OO

A pesar de la que la tecnología OO ha recibido atención recientemente, tiene más de veinte años de edad. Virtualmente todos los conceptos básicos del enfoque Orientado a Objetos fueron introducidos en el Lenguaje de Programación **Simula**, desarrollado en Noruega a finales de los 60's, con el fin de construir modelos funcionales de sistemas físicos complejos que pudiesen contener varios miles de componentes.

Era evidente en esas fechas que la programación modular era esencial para construir sistemas complejos y lo que es especial en Simula es la forma en que se definen los módulos: que están basados en los objetos físicos que se desea simular.

Esta elección tiene bastane sentido por que los objetos en una simulación ofrecen una forma natural de descomponer el problema a resolver. Cada objeto tiene un cierto comportamiento a ser modelado, y cada uno tiene que mantener información sobre su propio *status*. Para qué buscar otra forma de definir procedimientos y datos cuando el problema mismo nos ayuda a organizarlos.

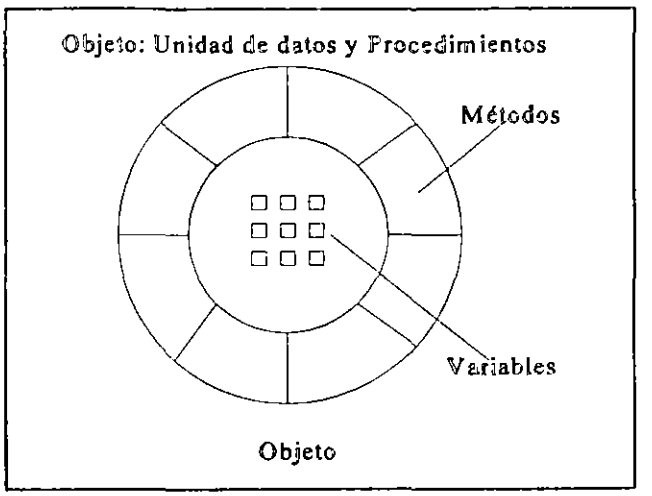
10.4.2 Dentro de los Objetos

El concepto de objetos surgió de la necesidad de modelar objetos del mundo real en simulaciones por computadora. Un objeto se define entonces como una unidad de Software que contiene una colección de datos y procedimientos interrrelacionados. Los datos se denominan como *Variables* porque definen el estado del objeto en cualquier momento. Los procedimientos reciben el nombre de *Métodos* y ellos definen todo el comportamiento de un objeto.

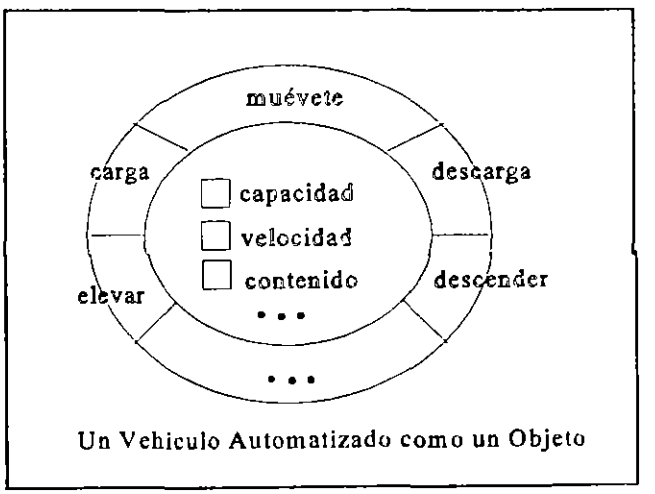
Por ejemplo, para representar un vehículo automatizado en la simulación de una fábrica debemos considerar que el vehículo puede realizar una variedad de acciones, tales como moverse de una

posición a otra, elevar su carga y descargar su contenido. También tiene que mantener información sobre sus características inherentes: capacidad de carga, velocidad máxima, etc., así como de su estado actual: contenido, posición, orientación y velocidad.

Para representar el vehículo como un objeto tendríamos que describir sus acciones posibles como métodos y sus características como variables. Durante la simulación, el objeto efectuaría sus diferentes métodos, cambiando sus variables conforme sea necesario para reflejar así los efectos de dichas acciones.



El concepto de un objeto es simple y a la vez poderoso. Los objetos forman módulos de software ideales debido a que pueden definirse y mantenerse independientemente, formando cada uno un universo autocontenido. *Todo lo que un objeto conoce está expresado en sus variables. Todo lo que puede hacer está expresado en sus métodos.*

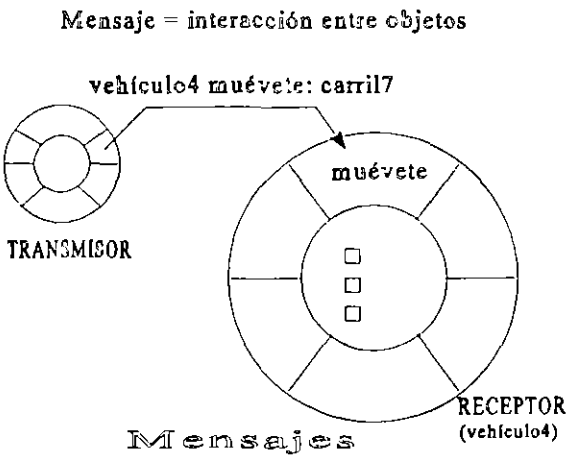


10.4.3 Mensajes

Los objetos del mundo real se pueden afectar en infinita variedad entre sí: crear, agregar, mover, enviar, doblar, etc. Esta tremenda variedad genera un problema interesante: cómo es posible representar todas estas interacciones en software. Una solución elegante a este problema es el *mensaje*.

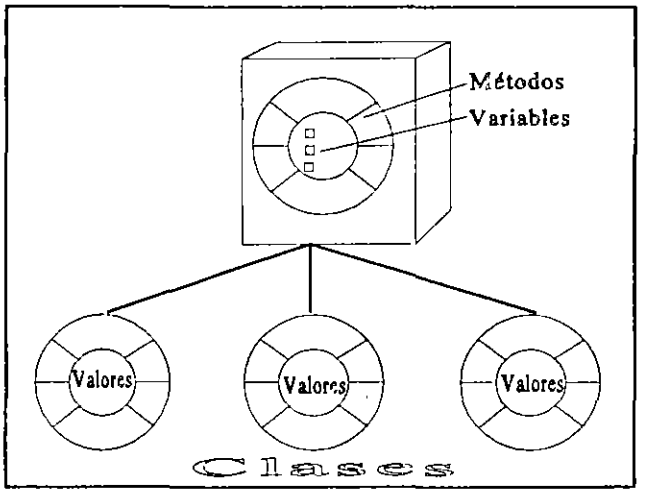
La forma en que los objetos interactúan unos con otros es enviándose mensajes pidiendo que se ejecute un método específico. Un mensaje consiste simplemente del nombre del objeto a quien va dirigido seguido del nombre del método que el objeto receptor sabe como ejecutar. Si el método requiere información adicional, el mensaje incluye esa información como parámetros.

El objeto que inicia el mensaje se conoce como transmisor y el que lo recibe como receptor. Un sistema OO consiste de varios objetos interactuando unos con otros enviándose mensajes entre sí. Debido a que todo lo que un objeto puede hacer está expresado en sus métodos, este simple mecanismo soporta todas las posibles interacciones entre objetos.



10.4.4 Clases

En raras ocasiones un sistema involucra un sólo objeto de cada tipo. Es mucho más común requerir más de uno. Por ejemplo, una fábrica automatizada puede tener cualquier número de vehículos. Sería extremadamente ineficiente el redefinir los mismos métodos para cada objeto que representa un vehículo.

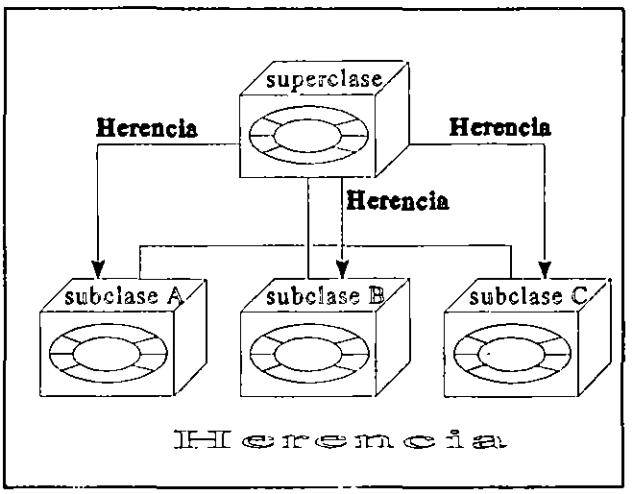


Aquí los autores de Simula ofrecieron otra solución elegante: La *Clase*. Una clase es un prototipo que define los métodos y variables que serán incluídas en un tipo de objeto en particular. Las descripciones de los métodos y variables que soportan se describen sólo una vez, en la definición de la clase.

Los objetos que pertenecen a una clase se denominan instancias de la clase, y contienen tan sólo los valores particulares para las variables, compartiendo el código de los métodos.

Para continuar con el ejemplo de un vehículo automatizado, la colección de vehículos podría representarse por la clase llamada *VehículoAutomatizado*, y esa clase contendría las definiciones de sus métodos y variables. Los vehículos en sí estarían representados por instancias de esta clase, cada uno con su nombre único: vehículo 11, vehículo 12, etc. Cuando uno de estos vehículos reciba un mensaje para ejecutar, iría a la clase por la definición del método y después aplicaría el método en sus propios valores locales.

10.4.5 Herencia



Herencia

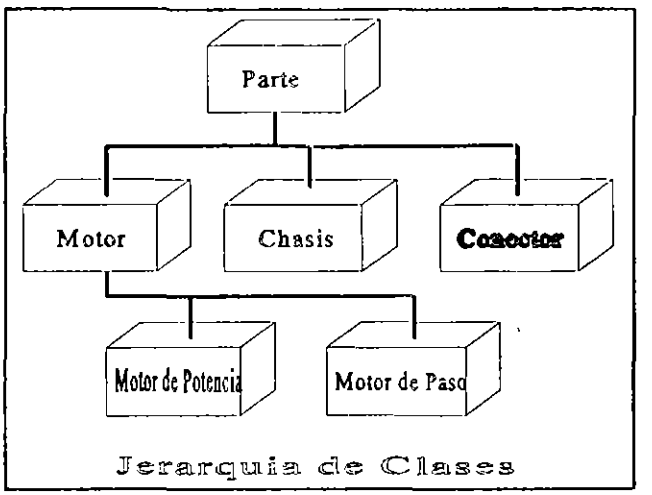
Simula dió un paso adicional con el concepto de clases al permitir que las clases fueran definidas en términos de otras. Si se requiere representar dos tipos de vehículos automatizados, es posible definir la clase de un vehículo a detalle y después definir la otra clase como la primera, agregándole algunos métodos y variables adicionales. Esta estrategia fué la primera forma de herencia y hoy en día es un elemento central de la tecnología de la POO.

La herencia es un mecanismo por el cuál una clase de objetos puede definirse como un caso especial de una clase más general, con lo cuál automáticamente incluye toda la definición de métodos y variables de la clase general. Casos especiales de una clase se conocen como subclases de esa clase, la clase más general de las subclases se conoce como la *superclase* de sus clases especializadas.

Continuando con nuestro ejemplo, utilizando una clase básica *VehículoAutomatizado*, se pueden crear nuevas clases que complementen con nuevas funciones dicha superclase sin necesidad de reescribir todo nuevamente, únicamente bastará con escribir la nueva funcionalidad.

10.4.6 Jerarquía de Clases

La herencia entre clases puede extenderse a cualquier grado. El resultado es una estructura arborescente conocida como *jerarquía de clases*. La invención de la jerarquía de clases es el verdadero genio de la tecnología OO. El conocimiento humano se estructura de esa manera, descansando en conceptos generales y refinando en casos cada vez más especializados.



Jerarquía de Clases

10.4.7 Programar con objetos

La POO frecuentemente se considera más natural que la programación tradicional, y esto es cierto en varios niveles. En un primer nivel, la POO, como hemos visto, es más natural porque permite organizar la información en formas que nos resultan más familiares. En un nivel más profundo, es más natural porque refleja las mismas técnicas que la naturaleza usa para manejar la complejidad.

El bloque básico a partir del cuál todos los seres vivos se componen es la célula. Las células son, también, una unidad que combina información y comportamientos. La mayoría de la información se almacena en el núcleo y el comportamiento se lleva a cabo por estructuras fuera del núcleo.

Las células están rodeadas de una membrana que permite sólo ciertos tipos de intercambios químicos con otras células. Esta

membrana no sólo proteje sus procesos internos de otras células, también oculta su complejidad y presenta una interfaz relativamente simple para el resto del organismo. Todas las interacciones entre células toman lugar por medio de mensajes químicos reconocidos por la membrana de la célula y pasados a su interior. Esta comunicación basada en mensajes simplifica grandemente la función de las células porque no tienen que leer las moléculas proteínicas o controlar las estructuras de otras células para obtener lo que requieren de ellas. Lo único que tienen que hacer es vocear su mensaje químico apropiado y la célula receptora actuará como le corresponde.

A partir de esta estructura básica hay una infinita variedad la cual no es caótica sino que está perfectamente organizada, o clasificada, en una jerarquía especializada de clases y subclases.

10.4.8 Lenguajes

Smalltalk se desarrolló a principios de los 70's en el Centro de Investigaciones de Xerox, en Palo Alto, California, y refleja la estrategia de diseñar un lenguaje totalmente nuevo para soportar el enfoque Orientado a Objetos, y se conoce como una de las más populares implantaciones de esa metodología a la fecha.

C++ fué desarrollado a principios de los 80's en los laboratorios Bell, de AT&T, y representa la estrategia de incluír los conceptos de la tecnología OO en un lenguaje existente "C". C++ se considera por esta razón un lenguaje híbrido, y junto con Smalltalk son los lenguajes más utilizados actualmente para el desarrollo de aplicaciones OO.

10.5 Análisis y Diseño Orientado a Objetos (Introducción al método de Booch)

La programación Orientada a Objetos, que últimamente se ha puesto de moda gracias a la disponibilidad comercial de varios lenguajes de programación como **Smalltalk**, **C++** o **Eiffel** requiere no sólo de entender el modelo de objetos sino también de apoyarse en algún método de análisis y diseño basado en este modelo. Diversas propuestas de este método empezaron a publicarse apenas hace unos cinco años. Aqui solo se presenta una introducción al método de Grady Booch, uno de los más completos en este momento, del cual ya existen las primeras herramientas disponibles para las computadoras personales y para las estaciones de trabajo.

10.5.1 Proceso de Análisis y Diseño

Un método de análisis y diseño orientado a objetos (ADOO) debe proponer los pasos a seguir para construir el modelo del sistema en términos de clases, objetos y relaciones entre ellos. Esta "receta" se conoce como el método de análisis y diseño. Durante el proceso de análisis se busca definir las clases semánticas del dominio del problema y, luego, durante el diseño, se trata de extenderlo al modelo del dominio de la solución. Este proceso, en el caso del método de Booch, no se presenta como una secuencia de pasos a seguir sino que se plantea como una serie de preguntas que debe contestar el diseñador para llevar a cabo la construcción del modelo.

Las preguntas son las siguientes:
¿Qué clases conforman el sistema y cómo se relaciona entre sí?
¿Cómo están estructurados los objetos individuales de estas clases y cómo colaboran entre sí?
¿Dónde estarán definidas las clases y creados los objetos ?
¿A qué procesador se asociarán los objetos activos y cómo se

organizará el manejo de los hilos de control, la comunicación y la sincronización en el caso de los sistemas concurrentes y/o distribuidos?

Las dos primeras preguntas están relacionadas con la estructura lógica global del sistema, mientras que las dos últimas se refieren a las decisiones de diseño que hay que tomar con respecto al mapeo físico del sistema de módulos de programas y a la arquitectura particular de máquinas(s).

Construir el modelo del sistema es ir contestando las preguntas en el orden que uno quiera. Claro que al principio uno empieza por las primeras dos, pero a medida de que avanza el diseño se pueden ir tomando decisiones que implican modificaciones de cualquier nivel, afectando la estructura lógica o física del sistema.

El proceso de análisis consiste en *descubrir* las clases de objetos que modelan el dominio del problema, mientras que el diseño requiere más de *invención* de clases adicionales y de *adaptación* de lo previamente modelado. El proceso de análisis y diseño tranquiliza mucho, por que normalmente trabajamos así. Uno va a ir identificando poco a poco las clases y sus relaciones, y al descubrir nuevas asociaciones tiende a modificar lo que ya no le gusta.

10.5.2 Notación para el método de Booch

El proceso de ADOO debe venir apoyado por una notación que permita representar el modelo que estamos construyendo. Por la complejidad y el tamaño de sistemas es imposible tener una sola notación para captar el modelo completo. Como remedio se proponen varias notaciones que reflejan diversos aspectos del sistema, conocidos como *vistas*(view). Las vistas representan la estructura estática y dinámica a diversos niveles de detalle, que funcionan como un *'zoom'* que nos permite *observar* el sistema con diversa

granularidad, desde la vista más abstracta hasta la más detallada.

En el caso del método de Booch se proponen cuatro notaciones distintas:

Diagramas de Clases	Diagrama de módulos
Diagramas de objetos	Diagrama de procesos

Cada uno de estos diagramas corresponde a la necesidad de ir denotando las respuestas a las preguntas que uno vaya contestando en el proceso de análisis y diseño, que mencionamos anteriormente.

Para ejemplificar el método de Booch, decidimos tomar un pequeño caso de estudio, que por supuesto, por su tamaño, nos podría reflejar todas las ventajas del método. Sin embargo, esperamos que por lo menos sirva como una primera aproximación al proceso de análisis y diseño, y a la notación de diagramas de clases del método de Booch.

10.5.3 Caso de estudio: Calendario

Para aclarar cuál es el problema a solucionar empezaremos por una descripción informal del problema.

Un calendario es una libreta de hojas correspondientes a un año. Cada hoja tiene asociada una fecha y unas listas de asuntos. Los asuntos se clasifican en citas, telefonemas y actividades.
El sistema a desarrollar debe proporcionar al usuario las siguientes funcionalidades:

crear un calendario del año específico
dar la hoja con fecha de hoy
dar la hoja con fecha específica
dar la hoja del día siguiente
dar la hoja del día anterior
en cada hoja dar de alta un asunto, modificar un asunto, dar de baja un asunto.

10.5.3.1 Análisis

El primer paso de análisis es descubrir en la descripción del problema los candidatos para las clases y los candidatos para las operaciones o métodos. Como lo sugiere Booch y otros autores, los candidatos para las abstracciones de clases se buscan entre los sustantivos significativos en la descripción, mientras que los candidatos para los métodos deben escogerse a partir de los verbos. Estas sugerencias no deberían sorprendernos, pues las clases del modelo de objetos son abstracciones para representar los objetos, no necesariamente tangibles, del mundo real, y los métodos mapean las acciones que estos objetos realizan. En el caso de nuestro ejemplo, los candidatos para las clases y los métodos son los siguientes:

Candidatos para las clases	Candidatos para los métodos
Calendario	CrearCalendario
LibretaDeHojas	DarHojaDeHoy
Hora	DarHoraDeLaFecha
Fecha	DarHojaDiaSig
ListasDeAsuntos	DarHojaDiaAnt
Asunto	DarDeAltaAsunto
Cita	ModificarAsunto
Telefonema	DarDeBajaAsunto
Actividad	

10.5.3.2 Diseño de Relación Entre Clases

Una vez identificados o descubiertos los candidatos iniciales para las clases y los métodos pasamos a la fase del diseño, que en el caso del método de Booch abarca todo el proceso de modelado del dominio del problema y del dominio de la solución.

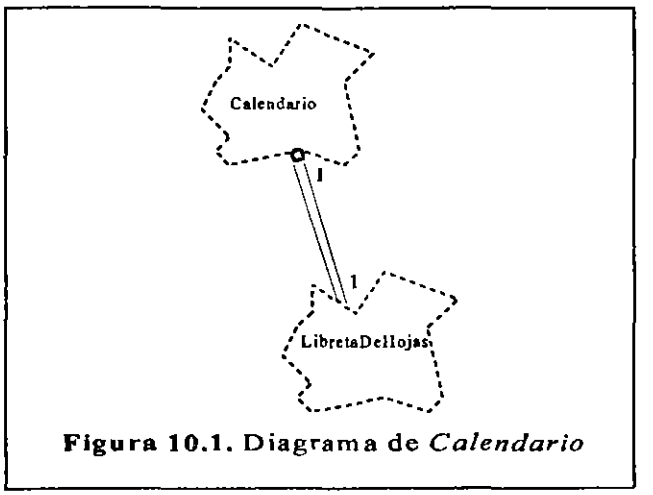


Figura 10.1. Diagrama de *Calendario*

El primer problema a descubrir en el diseño es definir las relaciones entre las clases basándonos en el conocimiento sobre el problema. Las relaciones básicamente en dos tipos:

la relación de Uso
la relación de Herencia

La relación de uso entre dos clases se establece cuando descubrirmos que una clase necesita de objetos de otra clase para realizar sus actividades. La relación de herencia entre dos clases se establece cuando una clase comparte el estado y el comportamiento con otra clase más general, añadiendo, tal vez, o modificando algunas cosas.

En el caso de nuestro ejemplo, fácilmente nos damos cuenta que un objeto de la clase *Calendario* necesariamente usa a un objeto de la clase *LibretaDeHojas*, lo que esquemáticamente se representa mediante el diagrama de la Figura 10.1. Las 'nubes' con líneas punteadas representan, en forma gráfica a las clases, mientras que las líneas paralelas con una 'bolita' del lado de la clase que es usuario de los objetos de la otra, representa la relación de uso. Una 'bolita' negra significa que la clase usuario necesita de otra clase para su implementación. Los números que aparecen sobre las

líneas paralelas nos dicen que un objeto de la clase *Calendario* requiere solamente de un objeto de la clase *LibretaDeHojas*.

Análogamente, descubrimos que la clase *LibretaDeHojas* requiere de varios objetos de la clase *Hoja*. La Figura 10.2 representa el diagrama correspondiente en el cual la letra 'n' del lado de la clase *Hoja* nos dice que se utilizará varios objetos de esta clase por un objeto de la clase *LibretaDeHojas*.

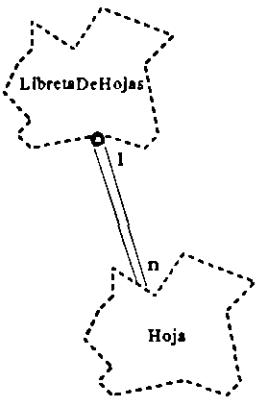


Figura 10.2. Diagrama de *LibretaDeHojas*

De la descripción informal sabemos que una hoja del calendario tiene asociada una fecha y las listas de asuntos, lo que en el diagrama de las clases se puede reflejar con doble relación de uso (Figura 10.3). Las listas de asuntos son tres: lista de citas, de telefonemas y de actividades, lo que en forma abstracta representa el diagrama de la figura 10.4.

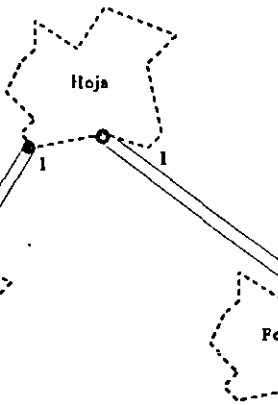


Figura 10.3. Diagrama de *Hoja*

La cosa se vuelve distinta cuando pensamos en la clase *Asuntos*. Utilizamos el concepto 'asunto' para abarcar con el mismo término asuntos más especializados como son citas, telefonemas y actividades. Un asunto no 'usa' citas, telefonemas ni actividades, un asunto 'es' una cita o un telefonema o una actividad. En este caso, la relación que se establece entre las clases es la relación de herencia (figura 10.5). Las flechas en el diagrama van desde la

clase heredera a la superclase (la que deja la herencia).

De esta forma hemos definido en forma abstracta las relaciones básicas entre los candidatos a clases. En el transcurso del diseño descubrimos la necesidad de tener clases auxiliares como *ListaDeCitas, ListaDeTelefonemas y ListaDeActividades,* las cuáles no se deducían en forma explícita de la descripción del problema. El diagrama completo de las clases descubiertas hasta el momento y sus relaciones se presenta en la figura 10.6.

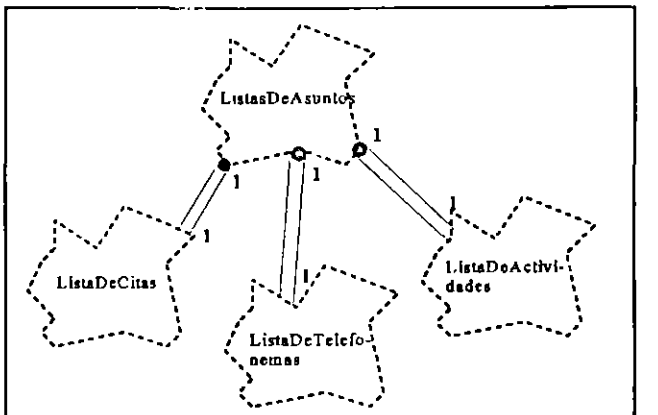


Figura 10.4. Diagrama de *ListasDeAsuntos*

El paso siguiente es tratar de asociar los candidatos a métodos a las clases mismas. Son las clases que con su comportamiento nos van a ofrecer las operaciones que deseamos que realice el sistema.

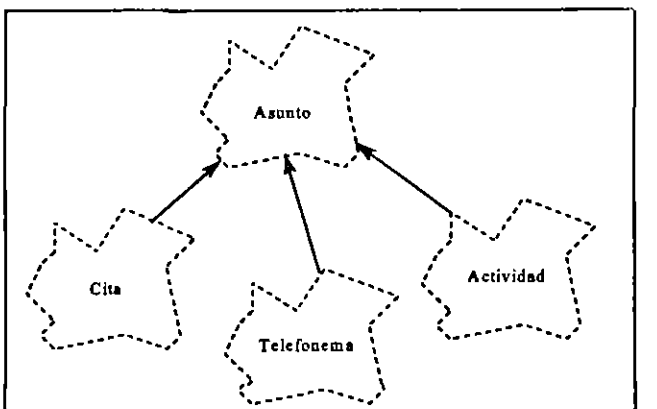


Figura 10.5. Diagrama de *Asunto*

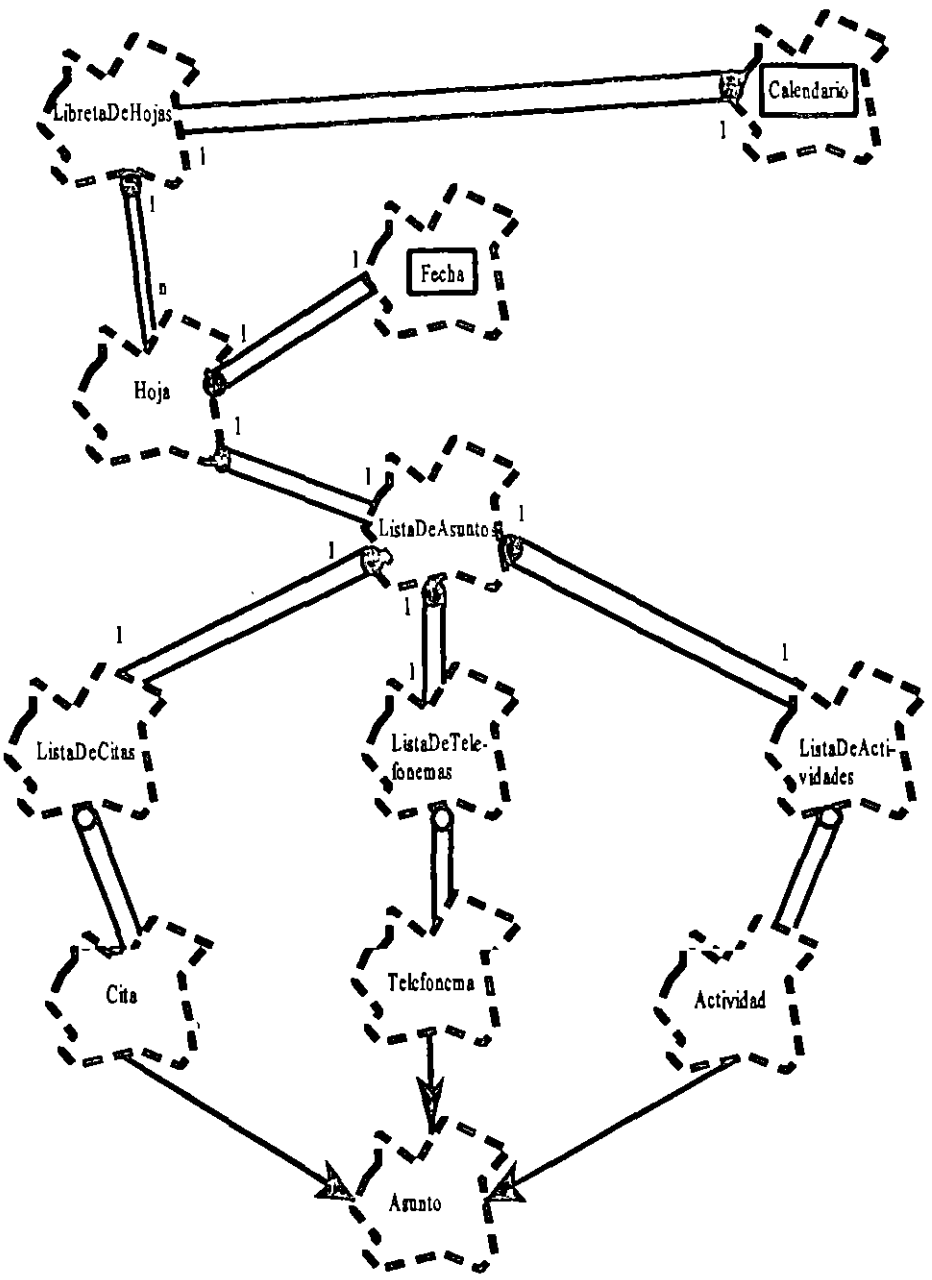


Figura 10.6 Diagrama Completo

10.5.3.3 Asociación de Métodos a Clases

A cada 'nube' del diagrama de clases, el método de Booch le asocia una descripción más precisa, conocida como esquema (template). En los esquemas, entre otras cosass, se enlistan los métodos que ofrece la clase como públicas. Regresando a nuestro ejemplo, los primeros cinco candidatos a métodos:

crearCalendario
darHojaDeHoy
darHojaDeLaFecha
darHojaDíaSig
darHojaDíaAnterior

son servicios que debe proporcionar la clase *Calendario*.. El esquema de la figura 10.7 presenta la descripción más precisa de esta clase donde todas las operaciones mencionadas aparecen como parte de la interfaz pública de la clase. El esquema contiene varios elementos que podemos ir detallando poco a poco mientras vayamos refinando el diseño. Algunos, como el nombre de la clase y la documentación, son obvios. La visibilidad se refiere a la situación de esta clase con respecto a la categoría a la cuál pertenece, de eso hablaremos más adelante. La cardinalidad indica cuántos objetos de la clase se permite crear, en este caso 'n' significa varios. La jerarquía (Hierarchy) y el uso (Uses) especifican las relaciones de herencia y uso con otras clases. La parte privada de la interfaz introduce, en este caso, la información sobre los campos del estado local de los objetos de la clase *Calendario*. Finalmente, los tres últimos descriptores dicen que la clase no tiene todavía definido el diagrama de estados, que sus objetos son secuenciales y persistentes.

Class name:

Calendario

Documentation:

Representa un calendario como una libreta de hojas del año específico. El calendario se inicializa con la hoja 1 de enero del año, esta hoja se vuelve su hoja actual.

Visibitily:	Exported
Cardinality:	n
Hierarchy:	
Superclasses: none	
Uses for implementation:	
	LibretaDeHojas
Public Interface:	
Operations:	CrearCalendario
	DarHojaDeHoy
	DarHojaDeLaFecha
	DarHojaDíaSig
	DarHojaDíaAnt
Private Interface:	
Fields:	Libreta:LibretaDeHojas
	HojaActual:Hoja
	Tiene la referencia a la hoja actual del calendario-hoja de hoy.
State Machine:	No
Concurrency:	Sequential
Persistence:	Persistent

Figura 10.7 Esquema de la Clase *Calendario*

Los métodos que nos hace falta asociar son lso que corresponden al manejo de asuntos sobre una hoja del calendario. Los colocaremos como servicios públicos de la propia clase *Hoja*. La figura 10.8 presenta el esquema correspondiente a esta clase.

Class name:
Hoja

Documentation:
Una hoja tiene asociada una fecha y un objeto de Listas de Asuntos.

Visibitily:	Privated
Cardinality:	n
Hierarchy:	
Superclasses: none	
Uses for implementation:	
	Fecha
	ListasDeAsuntos
Public Interface:	
Operations:	CrearHoja
	DesplegarAsuntos
	DarDeAltaAsunto
	DarDeBajaAsunto
	ModificarAsunto
	DarFecha
Private Interface:	
Fields:	Fecha:Fecha
	Asuntos:ListasDeAsuntos
State Machine:	No
Concurrency:	Sequential
Persistence:	Persistent

Figura 10.8 Esquema de la clase *Hoja*

10.5.3.4 Esquemas de Operaciones (métodos)

Los esquemas de clases dan una mejor aproximación de la estructura de clases que las 'nubes' del diagrama, pero sigue haciendo falta una mayor precisión en la descripción del comportamiento de sus métodos. Para tal fín Booch nos ofrece los esquemas de operaciones. La figura 10.9 presenta los distintos niveles de granularidad del diseño que se logran gracias al 'zoom' que permite observar clases con poco o mucho detalle.

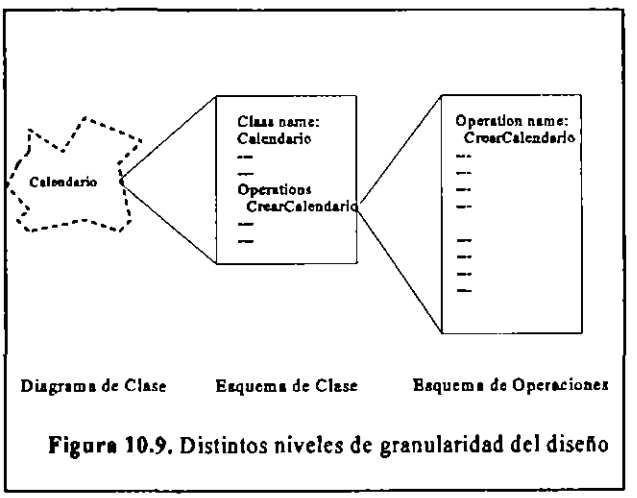


Figura 10.9. Distintos niveles de granularidad del diseño

Los esquemas de operaciones, aparte del nombre y la documentación (véase la figura 10.10), contienen la descripción de los parámetros y otros tres descriptores: *precondición, acción y postcondición*. La precondición define las restricciones que deben de cumplirse para que la operación se lleve a cabo exitosamente, la acción describe su comportamiento y la postcondición describe los resultados de la operación. La forma en que se describen estas partes se deja abierta, podemos usar el lenguaje natural, alguna especificación formal o hasta el código del lenguaje que se usará para la implementación. En este sentido el método de Booch es muy flexible, pero toda la responsabilidad por la claridad y consistencia de estos descriptores queda en nuestras manos.

Definiendo las acciones de las operaciones, por lo general nos damos cuenta que se requieren servicios de otras clases. Por ejemplo, para realizar *crearCalendario* descubrimos que necesitamos crear *libreta*, crear una hoja con fecha específica e insertar esa hoja en la libreta. Las tres operaciones hay que asignarlas entonces a las clases de *LibretaDeHojas* y *Hoja*, respectivamente. En este momento, el proceso de diseño nos lleva de regreso a modificar los esquemas de dichas clases, y tal vez hasta a modificar los diagrams de clases, si descubrimos la necesidad de introducir clases nuevas o modificar las relaciones entre la clases.

Operation name:

CrearCalendario

Member off: Calendario

Documentation:

Crea un calendario nuevo con la libreta inicializada, año definido y la fecha de hoy asignada al 1 de enero.

Formal Parameters:

a:1900..2100

Preconditions:

1900 < a < 2100

Actions:

Crea una libreta nueva y la asigna a la variable *Libreta*, se crea la *HojaActual* y se inicializa con el 1 de enero del año a, se inserta la *HojaActual* a la libreta.

Postconditions:

HojaActual.Fecha = 1/01/a y Libreta no es nula con *HojaActual* insertada

Concurrency: Sequential

Figura 10.10. Esquema de la operación *CrearCalendario*

10.5.4 Categorías de Clases

Cuando durante el diseño de un sistema el número de clases que se vayan descubriendo crece, se requiere de algún apoyo para manejar esta situación. En el caso del método de Booch se propone agrupar clases que son lógicamente afines en una categoría de clases. Gráficamente una categoría se dibuja como un rectángulo, y sustituye (otro movimiento de zoom) todos los diagramas de clases que le pertenecen. En el caso de nuestro ejemplo, es razonable ocupar todas las clases que hemos diseñado en una categoría, la cuál llamaremos *Calendarios* (figura 10.11). Una categoría exporta

a otras categorías algunas de las clases que agrupa, en nuestro ejemplo serían las clases *Calendario* y *Fecha*, y las demás clases quedan ocultas, porque su papel es relmente secundario para la definición del calendario. El criterio que podemos aplicar para escoger a las clases exportadas es su supuesta utilidad (reutilización) para varias aplicaciones.

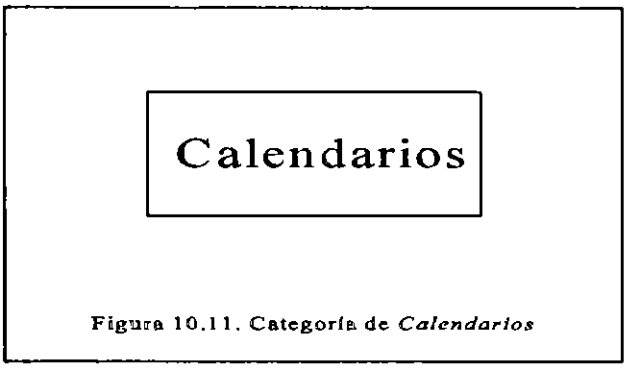


Figura 10.11. Categoría de *Calendarios*

10.5.5 Herramientas de diseño

Como se puede observar, este proceso de diseño es realmente un ir y venir entre diferentes niveles de diagramas y esquemas, afinando, modificando y añadiendo detalles. Para que este trabajo se nos facilite es indispensable contar con una herramienta computacional que nos ayude a *"mover el zoom"* automáticamente y guardar toda la información de distintas fases del diseño.

Afortunadamente ya hay en el mercado productos para las computadoras personales y para las estaciones de trabajo que apoyan el método de diseño de Booch. Particularmente, para desarrollar el ejemplo del calendario se utilizó el sistema **Rational Rose**, para PC's bajo Windows. Antes de usarlo es recomendable estudiar los primeros 9 capítulos del libro "Object-Oriented Design with Applications" de Booch (del cuál está disponible en español un resúmen de esta parte del libro: "Diseño Orientado a Objetos: Método de Booch" de Oktaba), para entender el significado de los distintos diagramas que se pueden construír.

En 1992 Grady Booch publicó en la revista **Computer Language** dos artículos proponiendo ciertas modificaciones a su notación con respecto a la propuesta del libro.

Actividades de análisis
Establecimiento de estándares (formatos de documentos, codificación, etc.).
Especificar procedimientos de control de calidad.
Identificar probables mejoras del producto.
Estimar recursos y costos de mantenimiento.
Actividades de diseño
Establecer la claridad y modularidad como criterios de diseño.
Diseñar para facilitar probables mejoras.
Usar notaciones estandarizadas para la documentación, algoritmos, etc.
Seguir los principios de ocultamiento de información, abstracción de datos y descomposición jerárquica de arriba hacia abajo.
Especificar efectos colaterales de cada módulo.
Actividades de implementación
Usar estructuras de una sola entrada y una sola salida.
Usar sangrado estándar en las diferentes estructuras.
Usar un estilo de codificación simple y claro.
Usar constantes simbólicas para asignar parámetros a las rutinas.
Proporcionar prólogos estándares de documentación en cada módulo.
Otras actividades
Desarrollar una guía de mantenimiento.
Desarrollar un juego de pruebas.
Proporcionar la documentación del juego de pruebas.

Tabla 1. Actividades que facilitan el mantenimiento de un sistema.

1.2 Reutilización de código.

La *reutilización de código* es otro de los factores que no se toma en cuenta en los métodos de diseño tradicionales. Cualquier programador que desarrolla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, entre otras, se repiten continuamente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente por él mismo

```
//STACK3.CPP: Métodos de la clase stack

#include <iostrema.h>
#include "stack.h"

stack::stack (int i) {
        n = 0;
        nmax = i;
        d = new double[nmax];
}

stack::~stack() {
        delete d;
}

double stack::push(double f) {
        if (n == nmax) {
                cerr "Error: stack lleno.\n" << endl;
                return 0;
        } else {
                d[n++] = f;
                return f;
        }
}

double stack::pop(void) {
        if (n == 0) {
                cerr << "Error: stack vacio.\n";
                return 0;
        } else {
                return d[--n];
        }
}

double stack::peep(void) {
        if (n == 0) {
                cerr << "(peep) Error: stack vacio.\n";
                return 0;
        } else
                return d[n-1];
}
void stack::limpia(void) {
        n = 0;
}
```

Figura 3.8. Stack3.cpp. Métodos de la clase stack.

```
# include <string.h>

class cadena {
        char *v;  // apuntador hacia los datos
        int l;    // longitud de la cadena
public:
        cadena() {  // crea una cadena nula
                v = 0;
                l = 0;
        }

        cadena(char *s) { // inicializa con una constante
                l = strlen(s);              // cadena
                v = new char[l + 1];
                strcpy( v, s );
        }

        cadena(cadena &s) {      // Este constructor se explica más tarde
                v = new char [l + 1];
                strcpy( v, s.v );
        }

        ~cadena() { // destructor
                if ( v )  delete v;
        }

        char elemento(int i); // regresa el i-ésimo carácter
                                                        //  de la cadena
        cadena subcadena(int n, int m); // obtiene una
                                        // subcadena de longitud m a partir del n-ésimo carácter
        cadena asigna(cadena s); // copia el contenido de s
        cadena concatena(cadena s); // concatena con la cadena s
};

cadena cadena::asigna(cadena s) {
        if ( v )
                delete v;
        l = s.l;
        v = new char [l + 1];
        strcpy( v, s.v );
        return *this;
}
```

Figura 4.1.a. Definición de la clase cadena, versión 1

```
char cadena::elemento(int i) {
        if (i > l)
                return '\0';
        return v[i];
}

cadena cadena::subcadena(int m, int n) {
        cadena tmp;
        if ( v ) {
                tmp.l = n - m + 1;
                tmp.v = new char [tmp.l + 1];
                char *v = tmp.v, *y = v + m;
                int i = tmp.l;
                while ( i--  &&  (*x++ = *y++) )
                        ;
                if ( i == -1 )    *x = '\0';
        }
        return tmp;
}

cadena cadena::concatena(cadena s) {
        cadena tmp;
        tmp.l = l + s.l;
        cadena cadena::subcadena(int m, int n) {
        cadena tmp;
        if ( v ) {
                tmp.l = n - m + 1;
                tmp.v = new char [tmp.l + 1];
                char *v = tmp.v, *y = v + m;
                int i = tmp.l;
                while ( i--  &&  (*x++ = *y++) )
                        ;
                if ( i == -1 )    *x = '\0';
        }
        return tmp;
}
cadena cadena::concatena(cadena s) {
        cadena tmp;
        tmp.l = l + s.l;
        tmp.v = new char[tmp.l + 1];
        strcpy( tmp.v , v );
        strcat( tmp.v , s.v );
        return tmp;
}
```

Figura 4.1.b. Definición de la clase cadena, versión 1 (continuación)

8.1 Un template simple.

La definición de una *clase template* especifica como clases individuales pueden ser construidas mucho mejor, como una declaración de clase que especifica como los objetos individuales pueden ser construidos. Definamos una clase template sencilla:

```
/*              s_temple.cpp
*/

#include <iostream.h>

template <class T>
class A {
        T x;
public:
        void read(){
                cout << "\n      Dame un valor para x ";
                cin >> x;
        }
        void print(){
                cout << "x++ es : " << ++x << endl;
        }
};

void main(){
        {
        A<char> i;
        i.read();
        i.print();
        }

        {
        A<int> i;
        i.read();
        i.print();
        }
}
```

En este ejemplo, podemos observar dos cosas; la forma de declarar una clase template, y la forma de crear una instancia de la clase.

La sintaxis para la declaración es:

template <class Nombre> función|clase {...};

Donde *Nombre* es el nombre real del template.

La sintaxis para crear una instancia es:

Nom_clase_temple**<tipo>** objeto

```
//STACKT.CPP: Métodos de la clase stack template

#include <iostream.h>
#include "stackt.h"

template <class T> stack<T>::stack (int i) {
        n = 0;
        nmax = i;
        d = new T[nmax];
}

template <class T> stack<T>::~stack () {
        delete d;
}

template <class T> T stack<T>::push(T f) {
        if (n == nmax) {
                cerr << "Error: stack lleno.\n" << endl;
                return 0;
        } else {
                d[n++] = f;
                return f;
        }
}

template <class T> T stack<T>::pop(void) {
        if (n == 0) {
                cerr << "Error: stack vacio.\n";
                return 0;
        } else {
                return d[--n];
        }
}

template <class T> T stack<T>::peep(void) {
        if (n == 0) {
                cerr << "(peep) Error: stack vacio.\n";
                return 0;
        } else
                return d[n-1];
}


void main(){

        stack<int> i(30);

        stack<char> cp;

        i.push(5);
        cp.push('h');

}
```

El prefijo *template <class T>* especifica que un template está siendo declarado y que un argumento T de tipo *type* será usada en la declaración. Después de esta instrucción, T es usada exactamente como cualquier otro tipo. El ámbito de T se extiende al final de la declaración del prefijo *template <class T>*.

Note que template <class T> dice que T es el nombre de un *tipo;* este no necesita ser actualmente el nombre de una clase. Para el objeto *i* del ejemplo anterior, T se vuelve *int*.

```
/*          maxt.cpp
*/

#include <iostream.h>
#include "cadena.h"

template <class T1,class T2>
T1 max(T1 a, T2 b){
     return ( a>b ? a : b );
};

void f(int a, int b, char c, char d)
{
     int  m1 = max(a,b);       // int    max(int ,int)
     char m2 = max(c,d);       // char max(char,char)
     int  m3 = max(a,c);       // int    max(int ,char)

     cout << "\n\tm1 = " << m1;
     cout << "\n\tm2 = " << m2;
     cout << "\n\tm3 = " << m3;
}

void main () {

     f( 5,6,'a','b' );

     cadena a="ala",b="alameda";
     cout << "\n";
     cout << (a > b ? a : b );     // operator > (a,b)
     cout << "\n";
     cout << max(a,b);

}
//   Recuerde abrir un Project e incluir los archivos maxt.cpp y
cadena.cpp
```