E V A L U A C I O N D E L P E R S O N A L 1

D O C E N T E

CURSO: PROGRAMACION ORIENTADA A OBJETOS USANDO C++, 1992.

FECHA: 16 de marzo al 6 de abril lunes, miercoles y viernes de 17 a 21 hrs.

	C O N F E R E N C I S T A					
1.-	ING. NORBERTO ARRIETA MARQUEZ					
2.-	ING. HECTOR M. BADILLO ROJAS					
3.-	JOSE A. CHAVEZ FLORES					
4.-	ING. AARON ARCOS TAPIA					
5.-						

EVALUACION TOTAL

ESCALA DE EVALUACION: 1 A 10

E V A L U A C I O N D E L A E N S E Ñ A N Z A 2

CURSO: PROGRAMACION ORIENTADA A OBJETOS USANDO C++, 1992.

FECHA: 16 de julio al 13 de agosto lunes, miercoles y viernes de 17 a 21 hrs.

	T E M A					
1.-	CONCEPTOS DE DISEÑO ORIENTADO A					
	OBJETOS.					
2.	PROGRAMACION FUNCIONAL EN C++					
3.-	USO DE CLASES					
4.	DEFINICION DE OPERADORES					
5.	HERENCIA					
6.	HERENCIA MULTIPLE					
7.	MANEJO DE MEMORIA					
8.	ENTRADA/SALIDA EN C++					
9.	METODOLOGIAS DE DISEÑO					
	EVALUACION TOTAL					

ESCALA DE EVALUACION: 1 A 10

EVALUACION DEL CURSO

	C O N C E P T O	
1.-	APLICACION INMEDIATA DE LOS CONCEPTOS EXPUESTOS	
2.-	CLARIDAD CON QUE SE EXPUSIERON LOS TEMAS	
3.-	GRADO DE ACTUALIZACION LOGRADO EN EL CURSO	
4.-	CUMPLIMIENTO DE LOS OBJETIVOS DEL CURSO	
5.-	CONTINUIDAD EN LOS TEMAS DEL CURSO	
6.-	CALIDAD DE LAS NOTAS DEL CURSO	
7.-	GRADO DE MOTIVACION LOGRADO EN EL CURSO	
	EVALUACION TOTAL	

ESCALA DE EVALUACION: 1 A 10

1.- ¿Qué le pareció el ambiente en la División de Educación Continua?

MUY AGRADABLE ☐ AGRADABLE ☐ DESAGRADABLE ☐

2.- Medio de comunicación por el que se enteró del curso:

PERIODICO EXCELSIOR ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA ☐

PERIODICO NOVEDADES ANUNCIO TITULADO DIVISION DE EDUCACION CONTINUA ☐

FOLLETO DEL CURSO ☐

CARTEL MENSUAL ☐

RADIO UNIVERSIDAD ☐

COMUNICACION CARTA, TELEFONO, VERBAL, ETC. ☐

REVISTAS TECNICAS ☐

FOLLETO ANUAL ☐

CARTELERA UNAM "LOS UNIVERSITARIOS HOY" ☐

GACETA UNAM ☐

3.- Medio de transporte utilizado para venir al Palacio de Minería:

AUTOMOVIL PARTICULAR ☐ METRO ☐ OTRO MEDIO ☐

4.- ¿Qué cambios haría en el programa para tratar de perfeccionar el curso?

_____

_____

_____

5.- ¿Recomendaría el curso a otras personas? ☐ SI ☐ NO

5.a.¿Qué periódico lee con mayor frecuencia?

_____

_____

6.- ¿Qué cursos le gustaría que ofreciera la División de Educación Continua?

_____

_____

_____

7.- La coordinación académica fué:

EXCELENTE ☐ BUENA ☐ REGULAR ☐ MALA ☐

8.- Si está interesado en tomar algún curso INTENSIVO ¿Cuál es el horario más conveniente para usted?

LUNES A VIERNES DE 9 a 13 H. Y DE 14 A 18 H. (CON COMIDAD) ☐

LUNES A VIERNES DE 17 a 21 H. ☐

LUNES A MIERCOLES Y VIERNES DE 18 A 21 H. ☐

MARTES Y JUEVES DE 18 A 21 H. ☐

VIERNES DE 17 A 21 H. SABADOS DE 9 A 14 H. ☐

VIERNES DE 17 A 21 H. SABADOS DE 9 A 13 H. DE 14 A 18 H. ☐

OTRO ☐

9.- ¿Qué servicios adicionales desearía que tuviese la División de Educación Continua, para los asistentes?

_____

_____

_____

10.- Otras sugerencias:

_____

_____

_____

DIRECTORIO DE ASISTENTES AL CURSO: PROGRAMACION ORIENTADA A OBJETOS EN C++

DEL 16 DE MARZO AL 6 DE ABRIL DE 1992

1.-JOSE GUADALUPE ALVARADO SANTANA
SUBDIRECTOR
BANCO INTERNACIONAL, S.N.C.

REFORMA NO. 45
TABACALERA CUAUHTEMOC
06030 MEXICO, D.F.
TEL. 721-5984

CALLE 14 NO. 78
MARAVILLAS
EDO. DE MEXICO
TEL. 721-5987

2.-ROBERTO I. CASTAÑEDA ALVARADO
JEFE DE DEPTO.
S. C. T.

ALTADENA NO. 23
NAPOLES BENITO JUAREZ
03810 MEXICO, D.F.
TEL. 687-6199

CALLE CARRIL NO. 19 EDIF-22-301
SAN JUAN XALPA IZTAPALAPA
09850 MEXICO, D.F.
TEL. 685-4414

3.-ANDRES LEOBARDO CHAVEZ OTERO
PROFESOR
INSTITUTO TECNOLOGICO DE TOLUCA

AV. INSTITUTO TECNOLOGICO S\N.
EX-RANCHO LA VIRGEN
METEPEC, EDO. DE MEXICO
TEL. 160344

CA;ADA 68 PLAZAS DE LA COLINA
MUNICIPIO TLANEPANTLA
EDO. DE MEXICO
TEL. 397-3241

4.-MARTHA GUTIERREZ GONZALEZ
ANALISTA PROGRAMADOR
U.N.A.M.
DIREC. GRAL. SERVICIOS COMPUTO LA ADMON.
MATIAS ROMERO NO. 1220
DEL VALLE BENITO JUAREZ
03100 MEXICO, D.F.
TEL. 554-5575

MARISOL NO. 25
STA. MA. LA RIBERA
MEXICO, D.F.
TEL. 547-9124

5.-ESTEBAN ISIDRO PIOQUINTO

U.N.A.M.
CHAPINGO, EDO. DE MEXICO

6.-JORGE LEROUX ROMERO
ANALISTA
PETROLEOS MEXICANOS

MARIAN NACIONAL NO. 350
ANAHUAC MIGUEL HIDALGO
11320 MEXICO, D.F.

QUETZAL NO. 4
LAS ARBOLEDAS
ATIZAPAN, EDO. DE MEXICO
TEL. 379-7195

7.-MARTIN ROBERTO MAQUEDA ZAMORA
INVESTIGADOR
INSTITUTO DE INVESTIGACIONES ELECTRICAS

LEIBNITZ NO. 20, 6o. P.
ANZURES MIGUEL HIDALGO
11590 MEXICO, D.F.
TEL. 254-6939

LAURELES NO. 39
AMPL. FORESTAL
MEXICO, D.F.

8.-G. FILIBERTO NAVARRETE NAVARRETE
ASESOR TECNICO
S. C. T. (METRO)

DELICIAS NO. 67
CENTRO CUAUHTEMOC
06000 MEXICO, D.F.
TEL. 512-5808

NOCHE TRISTE NO. 15-301
POPOTLA MIGUEL HIDALGO
11400 MEXICO, D.F.
TEL. 341-6915

9.-ERNESTO OLVERA ARANZOLO

U.N.A.M.
CHAPINGO

10.-MARCOS RODRIGUEZ BERNAL
INVETIGADOR B
INSTITUTO DE INVESTIGACIONES ELECTRICAS

DANTE NO. 36, 2o. P.
NUEVA ANZURES MIGUEL HIDALGO
11590 MEXICO, D.F.
TEL. 254-6959

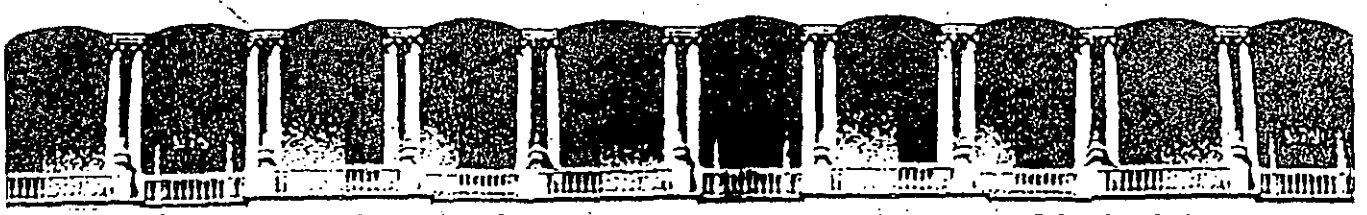
LEIBNITZ NO. 20, 6o. P.
NUEVA ANZURES MIGUEL HIDALGO
11590 MEXICO, D.F.
TEL. 254-8177

11.-MAURICIO APOLONIO RODRIGUEZ GARCIA
JEFE DE DEPTO.
S. C. T.

AV. CUAUHTEMOC NO. 614

MEXICO, D.F.
TEL. 519-2636

CIRCUITO RIO DEL ORO NO. 72
PASEOS DE CHURUBUSCO IZTAPALAPA
09030 MEXICO, D.F.
TEL. 654-4784



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

CENTRO DE INFORMACION Y DOCUMENTACION

"ING. BRUNO MASCANZONI"

EL CENTRO DE INFORMACION Y DOCUMENTACION "ING. BRUNO MASCANZONI" TIENE POR OBJETIVO SATISFACER LAS NECESIDADES DE ACTUALIZACION AL PROPORCIONAR LA ADECUADA INFORMACION QUE PERMITA A LOS PROFESIONALES INGENIEROS PROFESORES Y ALUMNOS, ESTAR AL TANTO DEL ESTADO ACTUAL DEL CONOCIMIENTO SOBRE TEMAS ESPECIFICOS ENFATIZANDO LAS INVESTIGACIONES DE VANGUARDIA DE LOS CAMPOS DE LA INGENIERIA TANTO NACIONALES COMO EXTRANJERAS.

POR LO QUE SE PONE A DISPOSICION DE LOS ASISTENTES DE LOS CURSOS DE LA D.E.C.F.I.; ASI COMO AL PUBLICO EN GENERAL.

EN DICHO CENTRO USTED TENDRA LOS SIGUIENTES SERVICOS:

* PRESTAMO INTERNO
* PRESTAMO EXTERNO
* PRESTAMO INTERBIBLIOTECARIO
* SERVICIO DE FOTOCOPIADO
* CONSULTA TELEFONICA
* CONSULTA A LOS BANCOS DE DATOS: LIBRUNAM EN CD-ROM Y EN LINEA

LOS MATERIALES A SU DISPOSICION SON:

* LIBROS
* TESIS DE POSGRADO
* NOTICIAS TECNICAS
* PUBLICACIONES PERIODICAS
* PUBLICACIONES DE LA ACADEMIA MEXICANA DE INGENIERIA
* NOTAS DE LOS CURSOS QUE SE HAN IMPARTIDO DE 1971 A LA FECHA

EN LAS AREAS DE INGENIERIA INDUSTRIAL, CIVIL, ELECTRONICA, CIENCIAS DE LA TIERRA, MECANICA Y ELECTRICA Y COMPUTACION.

EL C.I.D. SE ENCUENTRA UBICADO EN EL MEZZANINE DEL PALACIO DE MINERIA LADO ORIENTE. EN HORARIO DE SERVICIO DE 10:00 A 19:30 HORAS DE LUNES A VIERNES.

Palacio de Minería Calle de Tacuba 5 Primer piso Deleg. Cuauhtémoc 06000 México, D.F. Tel.: 521-40-20 Apdo. Postal M-2285

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

A LOS ASISTENTES A LOS CURSOS DE LA DIVISION DE EDUCACION CONTINUA

Las autoridades de la Facultad de Ingeniería, por conducto del Jefe de la División de Educación Continua, otorgan una constancia de asistencia a quienes cumplan con los requisitos establecidos para cada curso.

El control de asistencia se llevará a cabo a través de la persona que le entregó las notas. Las inasistencias serán computadas por las autoridades de la División, con el fin de entregarle constancia solamente a los alumnos que tengan un mínimo del 80% de asistencias.

Pedimos a los asistentes recoger su constancia el día de la clausura. Estas se retendrán por el período de un año, pasado este tiempo la DECFI no se hará responsable de este documento.

Se recomienda a los asistentes participar activamente con sus ideas y experiencias, pues los cursos que ofrece la División están planeados para que los profesores expongan una tesis, pero sobre todo, para que coordinen las opiniones de todos los interesados, constituyendo verdaderos seminarios.

Es muy importante que todos los asistentes llenen y entreguen su hoja de inscripción al inicio del curso, información que servirá para integrar un directorio de asistentes, que se entregará oportunamente.

Con el objeto de mejorar los servicios que la División de Educación Continua ofrece, al final del curso deberán entregar la evaluación a través de un cuestionario diseñado para emitir juicios anónimos.

Se recomienda llenar dicha evaluación conforme los profesores impartan sus clases; a efecto de no llenar en la última sesión las evaluaciones y con esto sean más fehacientes sus apreciaciones.

¡ G R A C I A S !

UNO DE LOS PROYECTOS QUE ACTUALMENTE ESTA LLEVANDO A CABO LA DECFI, ES LA ORGANIZACIÓN DE CURSOS DE ACTUALIZACIÓN EN TEMAS DE INGENIERÍA, DENTRO DE LOS CUALES SE INCLUYEN - PROGRAMAS DE COMPUTADORA RELACIONADOS CON EL TEMA DEL CURSO, LOS CUALES SE DISTRIBUIRÁN EN SUS VERSIONES FUENTE.

CON EL OBJETO DE CONOCER LOS TEMAS DE MAYOR INTERÉS PARA ESTE TIPO DE CURSOS, ASÍ COMO PARA DEFINIR LOS REQUISITOS TÉCNICOS QUE DEBEN REUNIR LOS PROGRAMAS A DISTRIBUIR, MUCHO AGRADECEREMOS A USTED SE SIRVA LLENAR EL SIGUIENTE CUESTIONARIO, EL CUAL SERÁ DE - UNA GRAN AYUDA PARA LA DECFI.

1.- CALIFIQUE CON ESCALA DE CERO A DIEZ LOS SIGUIENTES CURSOS UTILIZANDO LAS LÍNEAS EN BLANCO PARA AQUELLOS QUE USTED PROPONGA (0=NO INTERESA, 10 = INTERESA MUCHO)

ANÁLISIS ESTRUCTURAL	( )	ESTADÍSTICA	( )	CONTROL DE PERSONAL	( )
CONTROL DE OBRAS	( )	DISEÑO MECÁNICO	( )	ALMACENES	( )
RUTA CRÍTICA	( )	PROGRAMACIÓN ESTRUC.	( )	INV. DE OPERACIONES	( )
PROGRAMACIÓN LINEAL	( )	ESTRUCTURA DE DATOS	( )	CONTROL DE CALIDAD	( )
MATEMÁTICAS	( )	CONTABILIDAD	( )	ADMON. PROGRAMACIÓN DE LA PRODUCCIÓN	( )
_____	( )	_____	( )	_____	( )
_____	( )	_____	( )	_____	( )
_____	( )	_____	( )	_____	( )

DEBIDO A QUE LA PRINCIPAL CARACTERÍSTICA DE LOS CURSOS SERÍA LA DE DISTRIBUIR PROGRAMAS DE COMPUTADORA QUE PUEDAN SER USADAS POR LOS ASISTENTES EN SUS DIFERENTES EMPRESAS CON EL MENOR ESFUERZO DE ADAPTACIÓN.

2.- ¿PARA QUE TIPO DE COMPUTADORA DESEARÍA QUE SE ESCRIBIERAN LOS PROGRAMAS?

PRIMERA OPCIÓN MARCA _____ MODELO _____ LENGUAJE _____

SEGUNDA OPCIÓN MARCA _____ MODELO _____ LENGUAJE _____

TERCERA OPCIÓN MARCA _____ MODELO _____ LENGUAJE _____

SI USTED CONOCE ALGUNAS OTRAS PERSONAS INTERESADAS EN ESTE TIPO DE CURSOS, MUCHO LE - AGRADECEREMOS HACERLE LLEGAR UNA COPIA DE ESTA HOJA Y ENVIARLA POSTERIORMENTE A:

DIVISIÓN DE EDUCACIÓN CONTINUA
PALACIO DE MINERÍA
CALLE DE TACUBA NO. 5
DELEGACIÓN CUAUHTEMOC
06000 MÉXICO, D.F.

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

CUESTIONARIO PARA EL DIRECTORIO GENERAL

PARA USO EXCLUSIVO DE LA D.E.C.

ALTA [SI] MODIFICACION [SI]

1.– INSTRUCCIONES GENERALES.

A.– Escriba con letra de molde.

B.– Escriba un solo caracter por cuadro ó semicuadro.

C.– Para contestar las preguntas en las líneas ó en los semicuadros escriba, letras y en los cuadros pondrá números según sea la respuesta que se pide.

NOTA : No se haga ninguna anotación en los cuadros sombreados.

.NOMBRE DEL CURSO :_____ FOLIO : DIG. VER. (1 – 5 – 6)

1.– NOMBRE :

8 TITULO (PROFESIONAL) * 13 — 14 NOMBRE (S) APELLIDO PATERNO APELLIDO MATERNO. 41

NOTA : Separe mediante una coma (,) el (los) nombre (s) de los apellidos.

2.– REGISTRO FEDERAL DE CAUS.: 42 AÑO MES DIA 51

3.– CEDULA PROFESIONAL: 52 NUMERO 58

4.– TELEFONO PARTICULAR: 59 NUMERO 65

5.– TELEFONO OFICINA: 66 NUMERO 72 — 73 EXT. 76

6.– ASISTENTE AL CURSO O PROFESOR: _____ 77 — A 1 (80) — B 5 (80)

7.– DIRECCION PARTICULAR :

8 CALLE, NUMERO EXTERIOR E INTERIOR 41

42 COLONIA 71 — A 2 (80) — M 6 (80)

8 DELEGACION O CIUDAD 27 — _____ ESTADO (28 29)

CODIGO POSTAL 30 34

8.– _____ TITULO PROFESIONAL Y AREA (EJEMPLO: INGENIERO CIVIL) (35 36)

_____ ESPECIALIDAD (37 38)

10.– DIRECCION DE OFICINA:

39 CALLE, NUMERO EXTERIOR E INTERIOR 72 A 3 (80) M 7 (80)

8 COLONIA 37

38 DELEGACION O CIUDAD 57 _____ ESTADO 58 59

CODIGO POSTAL 60 64

11.– ASOCIACIONES A LAS QUE PERTENECE :

PRINCIPAL : _____ 65 66

OTRAS : _____ 67 68

_____ 69 70

_____ 71 72

_____ 73 74 A 4 (80) M 8 (80)

FECHA DE ELABORACION

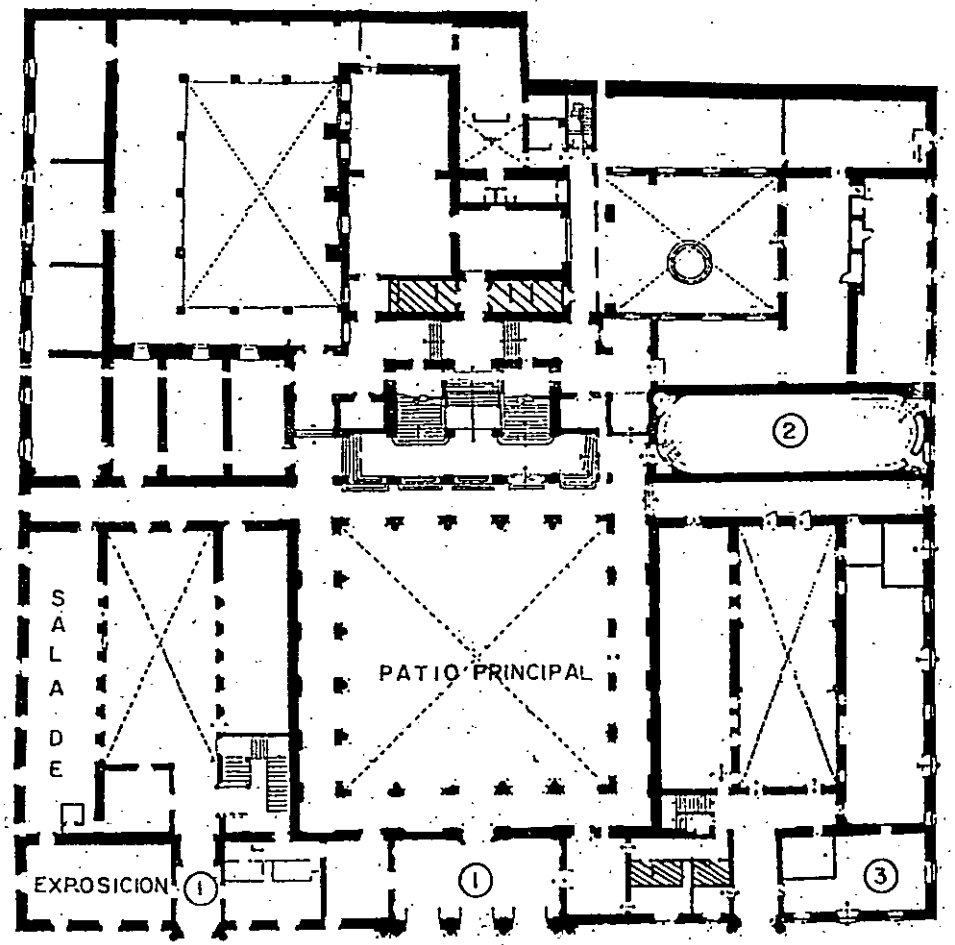
_____ A ____ DE _____ DE 19 ____

_____
FIRMA

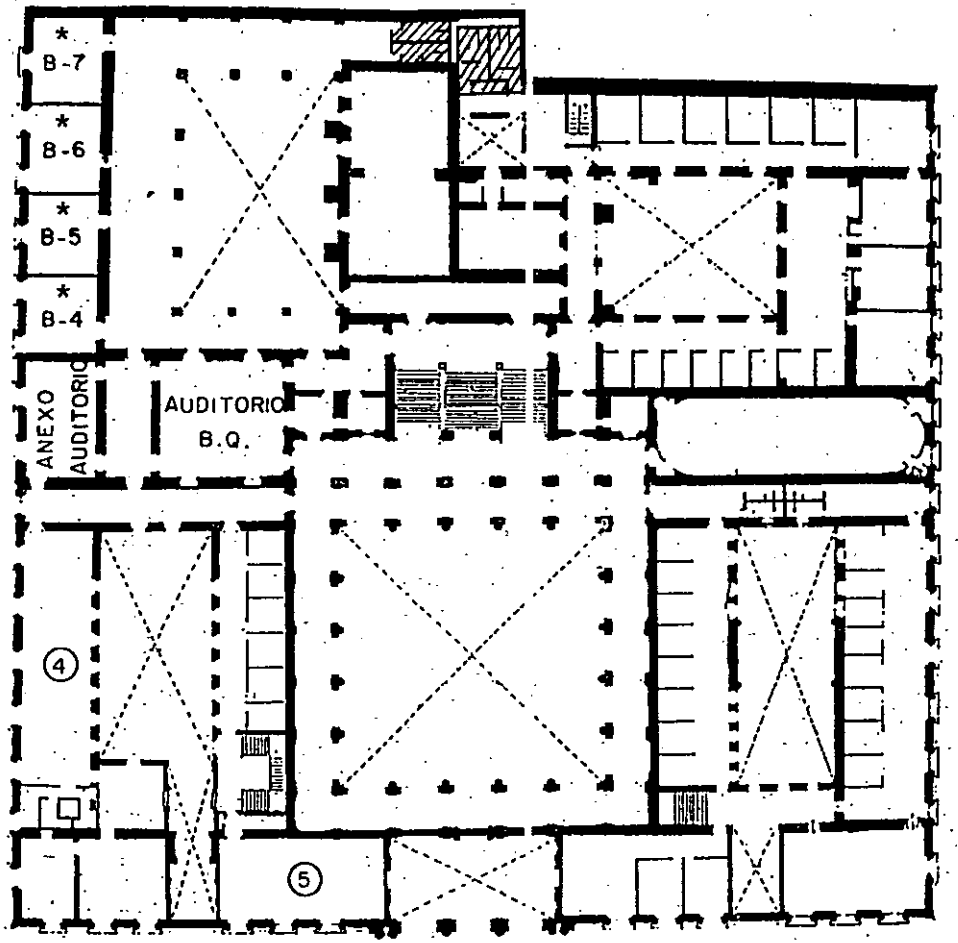
PARA USO EXCLUSIVO DE LA DIVISION DE EDUCACION CONTINUA

CODIFICO:	REVISO:	OBSERVACIONES:

PALACIO DE MINERIA

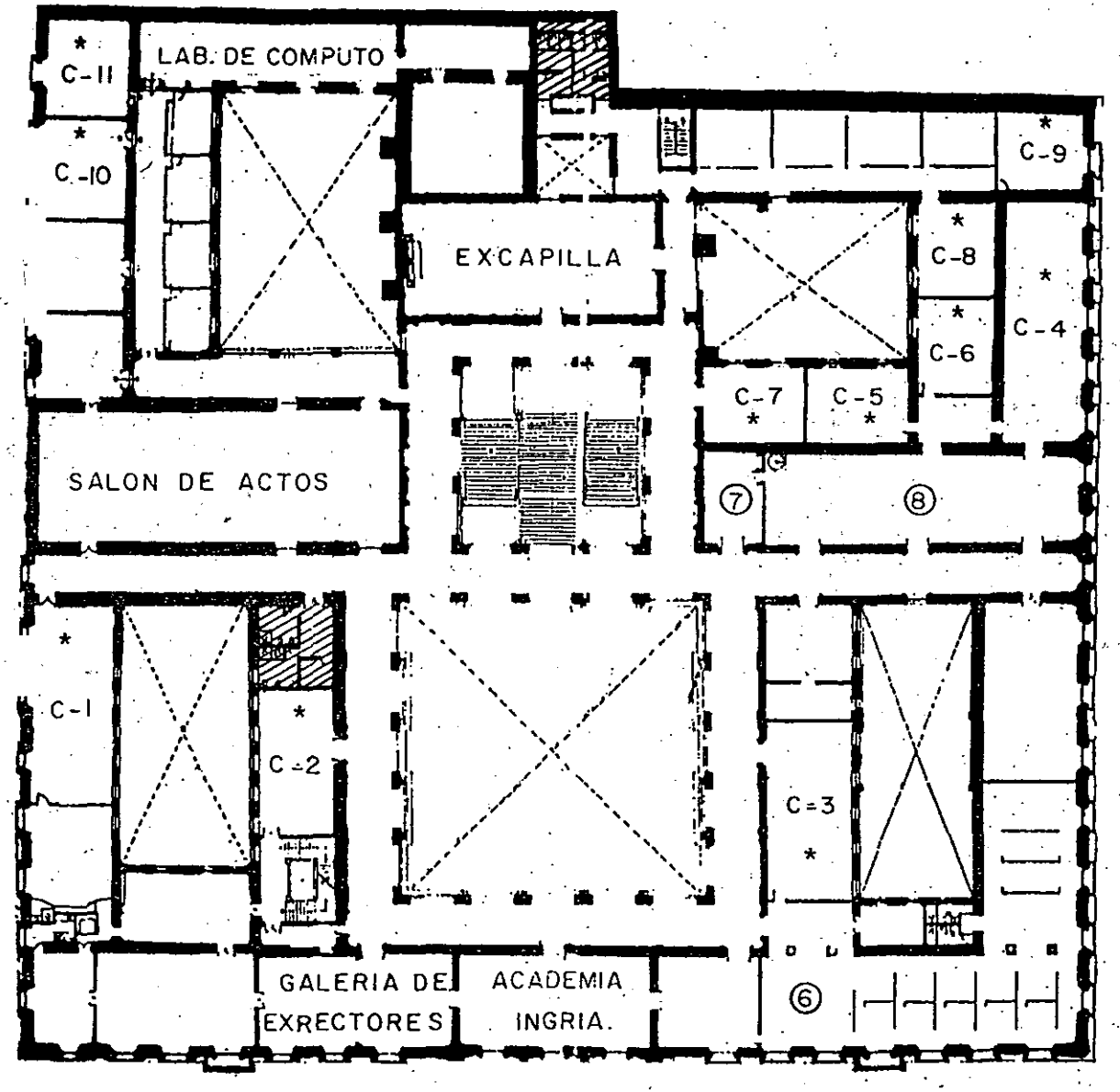


PLANTA BAJA



MEZZANINNE

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



1er. PISO

GUIA DE LOCALIZACION

1 – ACCESO

2 – BIBLIOTECA HISTORICA

3 – LIBRERIA U N A M

4 – CENTRO DE INFORMACION Y DOCUMENTACION "ING. BRUNO MASCANZONI"

5 – PROGRAMA DE APOYO A LA TITULACION

* AULAS

6 – OFICINAS GENERALES

7 – ENTREGA DE MATERIAL Y CONTROL DE ASISTENCIA.

8 – SALA DE DESCANSO

▨ SANITARIOS

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

NOTAS PARA EL CURSO

PROGRAMACION ORIENTADA A OBJETOS

USANDO C++

Norberto Arrieta Márquez
Marzo, 1991.

Capítulo I

La programación orientada a objetos

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

1.1 El problema del mantenimiento de software.

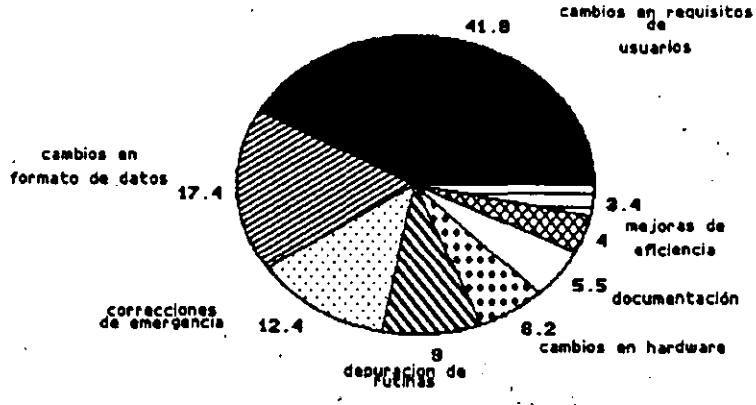
La construcción de software ha recibido la atención de los expertos desde hace mucho tiempo; en la década de los 70's se consiguieron avances significativos hacia el desarrollo de metodologías para construir programas en forma sistemática y a bajo

costo. Como resultado de esos esfuerzos surgieron técnicas que, como el diseño estructurado y el desarrollo descendente (top-down), durante mucho tiempo han 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:

- los productos que resultan al emplear estas técnicas son poco flexibles.
- los programadores que las usan tienden a concentrarse en el diseño y la implementación inicial del sistema, sin tomar en cuenta su vida posterior.
- 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 ó seis veces más grande que el lapso en que se desarrolla; por ejemplo, un sistema que se desarrolla en uno ó dos años puede mantenerse trabajando durante un período que va de cinco a quince años. Los gastos que se hacen durante este último período (*gastos de mantenimiento*) representan alrededor del 70% del costo total del sistema. La figura 1 ilustra la forma en que se distribuyen estos

gastos.



La gráfica muestra que cerca del 60% de los costos de mantenimiento de un sistema (alrededor del 42% del costo total) se tienen que hacer por cambios en las especificaciones del usuario ó en los formatos de los datos. Es por ello que la *extensibilidad* (la facilidad con que se modifica un sistema para que 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 faciliten su propio mantenimiento. Pero, ¿cómo alcanzar este objetivo? Los expertos recomiendan una serie de actividades y heurísticas que pueden servir como guía durante el desarrollo del sistema. La tabla 1 agrupa tales actividades de acuerdo a la etapa de desarrollo en que se deben realizar; más tarde nos ocuparemos

con mayor cuidado de la *modularidad*, el *ocultamiento de 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 estandard en las diferentes estructuras.
Usar un estilo de codificación simple y claro.
Usar constantes simbólicas para asignar parámetros a las rutinas.
Proporcionar prólogos estandard 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 última forma de reutilización se da cuando un programador utiliza parte un programa escrito con anterioridad para crear un nuevo programa. Sin embargo, estas 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 ó 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 cuenta con un conjunto de módulos que realizan una función específica, se debe alentar al programador a usar esos módulos 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 interfaces (medios de comunicación con otros módulos), y éstas deben ser pequeñas. Un número pequeño de interfaces 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 una 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 interfaces explícitas y bien definidas (esto favorece la comprensibilidad y la protección modular).

1.4 La programación orientada a objetos.

La *programación orientada a 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 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 se simplifica en gran medida si cada una de las entidades (u *objetos*) del problema que se está 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 se 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 (como el nombre, sueldo y edad de un empleado) y en las operaciones que se pueden realizar sobre ese

objeto (como calcular el sueldo de un empleado ó los impuestos que debe pagar). De esta forma, un *tipo de dato abstracto* se puede 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 modernos. 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, en la mayoría de los programas, los objetos involucrados son **mucho** más complejos que un número de punto flotante. La programación orientada a objetos 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 ó para implementar el recorrido de un árbol) únicamente necesita tomar en cuenta la propiedad LIFO (último en entrar primero en salir) 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 instrumenta el stack (p.e. un arreglo ó 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 está 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 y manipulación 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 (llamadas *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 *interface* 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 una 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 clases *stack*, *ventana* y *editor*. Los procesos de composición y de herencia múltiple son muy parecidos, más adelante se tratará este aspecto.

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

con lo cual es posible definir interfaces uniformes para diferentes tipos de objetos.

1.5 Lenguajes orientados a objetos.

Las técnicas en las que se basa la programación orientada a objetos (ocultamiento de 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 de objetos de diferentes clases de la misma forma en que manejan 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 interfaces de usuario gráficas y Eiffel, que 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 (como el Turbo Pascal 5.5 de Borland).

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 inicial 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 origenes 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 porque el autor deseaba escribir ciertas 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 tiempo y espacio 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 desperdiciadas. 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 rendimientos 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 sus 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 embargo, 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 la AT&T; sin embargo, se espera que para fines de 1992, la ANSI publique un estandard del lenguaje.

C++ se ha empleado en gran medida en ambiente Unix; sin embargo, también existen compiladores que corren bajo otros sistemas operativos. En el ambiente MSDOS, los compiladores más populares son los por 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 TurboC++ de Borland; aunque en ellos sólo se utilizaron características estándares, el lector debe estar consciente de que tal vez se necesiten pequeños cambios para correrlos utilizando otro compilador. Lo mismo se aplica para la descripción del lenguaje que se hace aquí.

Capítulo 2:

El léxico de C++

Todos los lenguajes se construyen en base a pequeñas unidades llamadas *símbolos*; en un lenguaje de programación esas unidades son llamados *símbolos terminales* ó *palabras* (en inglés reciben el nombre de *tokens*). La primer etapa de un compilador debe dividir el programa fuente en palabras, para después formar las *frases* u *oraciones* del lenguaje (por ejemplo expresiones y proposiciones) y analizar su significado. Las palabras de C++ incluyen a los *identificadores*, las *palabras reservadas*, las *constantes* y los *operadores*. En este capítulo se presentan las reglas para construir identificadores, palabras reservadas y constantes; los operadores y las constantes simbólicas se analizan en el siguiente capítulo.

2.1 Identificadores.

Un identificador en C++ se forma con una secuencia de dígitos y letras que empieza con una letra; el subguión (_) se considera como una letra. Al igual que en C, las letras mayúsculas son diferentes a las minúsculas. Un identificador puede tener cualquier número de caracteres, aunque programas externos al compilador (como

los ligadores, por ejemplo) pueden imponer restricciones de longitud. Ejemplos de identificadores bien formados son los siguientes:

```
CostoInicial    costo_final    origen    total1990
_AX             b52            identificador_muy_largo
```

Algunos identificadores incorrectos son:

```
67ABC           costo$         -precio   Precio.final
```

2.2 Palabras reservadas.

Algunos identificadores se encuentran reservados para construir las frases propias del lenguaje; estos identificadores, llamados palabras reservadas, no pueden ser usados por el programador para otros fines.

La lista de palabras reservadas de C++ 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
```

Algunas de esas palabras no se usan en las versiones actuales del lenguaje pero se encuentran reservadas para versiones

posteriores.

2.3 Comentarios y espacios en blanco.

El carácter blanco, el tabulador, el fin de línea y los comentarios son considerados por el compilador únicamente como separadores entre palabras. Excepto en unos cuantos contextos especiales (dentro de una cadena de caracteres, por ejemplo), el compilador ignora todas estas secuencias de caracteres; sin embargo, el programador debe emplearlas para facilitar la lectura y comprensión de sus programas.

C++ sigue conservando la notación de C para los comentarios: cualquier secuencia de caracteres delimitada por /* y */. Además, C++ permite utilizar comentarios de línea, que inician con la secuencia // y terminan en el fin de línea. Por ejemplo:

```
#include <iostream.h>
main() { // un programa muy sencillo
     cout << "Hola, mundo.\n";
}
```

2.4 Constantes.

C++ cuenta con tres clases de constantes para expresar valores de los tipos básicos: caracteres, cadenas, números enteros y números de punto flotante. El lenguaje también permite construir constantes para tipos de datos definidos por el usuario; sin embargo, ese tipo de constantes se analizarán en un capítulo

posterior.

Una constante carácter se debe escribir entre comillas sencillas, por ejemplo: 'a', '+', '%', etc.. C++ provee de secuencias especiales para expresar caracteres no visibles, por ejemplo: '\n', '\t', etc. La lista completa de estas secuencias es la siguiente:

'\n' - fin de línea	'\t' - tabulador
'\r' - retorno de carro	'\v' - tabulador vertical
'\b' - backspace	'\f' - salto de hoja
'\a' - campana	'\\' - antidiagonal
'\'' - comillas sencilla	'\"' - comillas
'\0' - nulo	

Cualquier carácter se puede escribir utilizando las notaciones '\0*ooo*' ó '\x*hh*', en donde *ooo* y *hh* representan el código numérico del carácter en octal y hexadecimal, respectivamente. Por ejemplo, si la computadora en que se está trabajando utiliza código ASCII, el carácter 'a' (ASCII 96) se puede expresar también como '\0140' ó '\x60. Todas estas secuencias de caracteres también se pueden usar dentro de una cadena.

Una constante de tipo cadena consiste de una secuencia de caracteres delimitados por comillas. Por ejemplo "Hola", "adiós", "%s %d", etc. Al final de una constante cadena se incluye automáticamente el carácter nulo ('\0'); en particular, la cadena

"a" consta de dos caracteres: 'a' y '\0'. Dentro de una cadena, una antidiagonal seguida de un fin de línea se ignora; además, dos cadenas constantes contiguas se concatenan automáticamente (de estas dos formas se pueden escribir cadenas muy largas en varios renglones).

Las constantes enteras se construyen con una secuencia de dígitos precedidos por un signo opcional. Si la constante empieza con cero (0), se asume que el número está escrito en octal; si empieza con la secuencia 0x ó 0X, se asume que está escrito en hexadecimal (no hay una notación para escribir números en binario). Como ejemplo, las constantes 96, 0140 y 0x60 son equivalentes. Si una constante entera se precede con la letra l ó L, se tomará como una constante entera larga (los enteros largos se presentan en el siguiente capítulo).

Una constante de punto flotante tiene una parte entera, un punto decimal y una parte fraccionaria, por ejemplo 3.14159, 7.21, 0.567, etc. C++ permite usar notación científica, como en 123.456e-7. Toda constante real es de doble precisión, excepto si se precede con una letra f ó F.

Capítulo 3

Variables y expresiones

Un programa es un modelo de un pequeño subconjunto del mundo real; en ese modelo los objetos externos se representan con entidades 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 este capítulo 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 7.

3.1 Tipos de datos.

Básicamente C++ sólo cuenta con dos tipos de datos: enteros y números de punto flotante; sin embargo, a partir de ellos es posible construir tipos más complejos como apuntadores, arreglos, funciones, estructuras y clases. Los tipos básicos de C++ se designan con las palabras **char**, **int**, **float** y **double**, además de los modificadores **short**, **long**, **signed** y **unsigned**.

Las variables de tipo carácter se designan con la palabra **char**. 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 caracteres pueden ser calificados como **signed** o **unsigned**; en el primer caso el bit más significativo se interpreta como el signo de ese valor, en el segundo todos los bits se utilizan para el código del carácter (este último tipo es útil cuando se usan códigos extendidos).

La palabra reservada **int** especifica una variable de tipo entero. Los enteros pueden ser calificados con las palabras reservadas **short**, **long**, **signed** y **unsigned**. Las dos primeras sirven para diferenciar el tamaño de los enteros: generalmente, un **int** se ajusta al tamaño de la palabra de la computadora que se está usando; los enteros cortos (**short int**) y los enteros largos (**long**

int) se incluyeron con el propósito de disminuir o ampliar el rango de valores de los enteros simples (**int**). Sin embargo, dependiendo del compilador que se esté usando **short** o **long** pueden ser equivalentes a **int**.

Los enteros sin signo (**unsigned int**) sólo pueden ser positivos; todos los bits de un entero de este tipo se utilizan para la magnitud, sin reservar bits para representar el signo del número. Por supuesto, **signed int** es el default para **int**.

Cuando la declaración de un entero se califica con uno de los modificadores anteriores, la palabra **int** puede omitirse. Por ejemplo, **unsigned int** equivale a **unsigned** y **long int** equivale a **long**.

Por último, **float** se utiliza para definir una variable de punto flotante de precisión sencilla; double se utiliza para variables de punto flotante de doble precisión. El modificador **long** se puede aplicar a **double** para especificar un tipo de punto flotante de alta precisión (**long double**).

La cantidad de memoria que se reserva para las variables de cada tipo depende de la implementación del lenguaje; en el compilador TurboC++ 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 programa de la figura 3.1 (cout es un objeto que despliega un mensaje en la pantalla; su declaración se encuentra en iostream.h).

```
#include <iostream.h>
int main() {
     cout << "Tamaño de char:         " << sizeof(char) << "\n";
     cout << "Tamaño de short:        " << sizeof(short) << "\n";
     cout << "Tamaño de int:          " << sizeof(int) << "\n";
     cout << "Tamaño de long:         " << sizeof(long) << "\n";
     cout << "Tamaño de float:        " << sizeof(float) << "\n";
     cout << "Tamaño de double:       " << sizeof(double) << "\n";
     cout << "Tamaño de long double: " << sizeof(long double) <<
                                                          "\n";
}
```

Fig. 3.1. Tamaños en bytes de los tipos básicos

3.2 Declaraciones.

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 distancia, modulo;
```

Las variables pueden ser inicializadas en el momento de su declaración:

```
int i = 3;
float distancia = 345.89;
```

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, lo que en algunas ocasiones obliga al programador a declarar las variables mucho antes de 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() {
     // ...
     int c;
     for (int cars = 0; (c = getchar()) != EOF; cars++)
          ;
     // ...
}
```

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

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

3.3 Constantes simbólicas.

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';
      // ...
}
```

Los *enumerados* son otro tipo de constantes simbólicas. Un enumerado consiste de una serie de identificadores asociados con valores enteros:

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

El primer identificador se asocia con el valor cero, el segundo con el valor uno, etc.. 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];
     // ...
}
```

Si se necesita que algunos de los identificadores tomen valores específicos, se pueden asignar esos valores en el momento de declarar el enumerado; los identificadores a los que no se les asigne valor tomarán valores secuenciales a partir de la última asignación:

```
enum mes = { enero = 1, febrero, marzo, /*...*/ diciembre };
/* enero = 1, febrero = 2, ..., diciembre = 12 */
```

3.4 Conversiones de tipos.

Cuando en una expresión intervienen operandos de diferentes tipos, éstos se convierten a un mismo tipo de acuerdo a las siguientes reglas:

- Antes de evaluar cualquier expresión, los caracteres (**char**) y los enteros cortos (**short**) se convierten siempre a enteros (**int**). Los reales de precisión sencilla (**float**) se convierten a reales de doble precisión (**double**).

- Si algún operando es **long double**, el otro se convierte a **long double**.
- Si algún operando es **double**, el otro se convierte a **double**.
- Si algún operando es **long**, el otro se convierte a **long**.
- Si algún operando es **unsigned**, el otro se convierte a **unsigned**.

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:

```
(int) (i * 3.1459)          (char *) p
int (i * 3.14158)           float(i) / float(3)
```

La conversión de entero a carácter se realiza truncando la parte más significativa del entero; la conversión de punto flotante a entero se hace mediante truncamiento; la conversión de doble precisión a precisión sencilla, mediante redondeo ó truncamiento, dependiendo del compilador.

3.5 Operadores aritméticos.

Los operadores aritméticos de C++ son + (suma), - (resta), / (división), * (multiplicación) y % (módulo). El - unario indica cambio de signo; existe también un + unario. La división de enteros

da como resultado un entero. Por supuesto, el operador de módulo sólo puede aplicarse a enteros.

3.6 Operadores relacionales y lógicos.

C++ cuenta con los mismos operadores relacionales y lógicos de C. Los operadores relacionales son > (mayor), >= (mayor o igual), < (menor), <= (menor o igual), == (igual a) y != (diferente de). Estos operadores devuelven un valor de cero si la relación es falsa y un valor de uno si es verdadera.

Los operadores lógicos son || (o), && (y) y ! (negación). En C++, al igual que en C, un valor diferente de cero representa un uno lógico y un valor igual a cero representa un cero lógico; de esta forma, || devuelve cero cuando sus dos operandos son cero y uno en caso contrario; && devuelve uno si sus dos operandos son uno y cero en caso contrario; ! devuelve uno si su operando es cero y cero si su operando es uno.

Las expresiones en donde aparecen los operadores lógicos se evalúan de izquierda a derecha y la evaluación se interrumpe cuando se conoce el resultado de la expresión. En el siguiente ejemplo nunca se preguntará si *s[maxlong]* es diferente al carácter nulo, pues la condición del ciclo **for** será falsa cuando *i* sea igual a *maxlong*:

```
main() {
     const maxlong = 100; char s[maxlong];
     // ...
     for (int i = 0; i < maxlong && s[i] != '\0'; i++)
          // ...
}
```

3.7 Operadores de incremento y decremento

Hay dos operadores en C++ que permiten incrementar el valor de una variable: **++** , que incrementa en una unidad a su operando, y **--**, que decrementa en una unidad a su operando. Los dos operadores se pueden usar en forma prefija o postfija; de la primer forma, el operando se incrementa primero y después se utiliza su valor en la expresión; de la segunda forma, primero se incrementa al operando y después se utiliza su valor. Por ejemplo, en el siguiente fragmento de programa

```
int i = 7;
int j = i++;
```

la variable *j* tendrá un valor de siete y la variable *i* de ocho, mientras que en

```
int i = 7;
int j = ++i;
```

las dos variables tendrán un valor de ocho.

Los operadores de incremento sólo se pueden aplicar a variables; en particular

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

no es una expresión válida.

3.8 Operadores de asignación

El operador de asignación básico es =. La expresión que constituye al operando derecho se evalúa y luego se asigna al operando izquierdo, que debe ser un variable. El resultado de toda la expresión es el operando derecho, por lo tanto el valor de una expresión de asignación se puede emplear en otras expresiones, como en

```
a = (b = 5) + (c = 7);
```

que asigna los valores cinco, siete y doce a las variables *b*, *c* y *a*, respectivamente. La expresión

```
x = y = z = 0
```

asigna cero a las tres variables (el operador de asignación se asocia de derecha a izquierda).

Además del operador =, muchos de los operadores binarios de C++ tienen un correspondiente operador de asignación. Si **op** es uno de los operadores *, /, %, +, -, <<, >>, &, |, ^, && ó ||, entonces existe un operador binario **op=** que, al usarse en una expresión de la forma

```
e1 op= e2
```

se traduce en

```
e1 = e1 op e2
```

Por ejemplo

```
i += 5
x *= 2
```

son equivalentes a

```
i = i + 5
x = x * 2
```

3.9 Operadores para manejo de bits

C++ permite al programador manipular los bits individuales de una variable utilizando los operadores & (y), | (o), ^ (o exclusivo), << (corrimiento a la izquierda), >> (corrimiento a la derecha) y ~ (complemento a uno). Haciendo uso de estos operadores es posible encender, apagar o invertir un grupo específicos de bits de un valor entero. Las tablas lógicas para los operadores &, |, ^, y ~ son:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

A	~A
0	1
1	0

Estos cuatro operadores actúan sobre cada uno de los bits de

su operando, por ejemplo:

```
int i = 0xb765;
int j = i & 0xff00;
```

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

```
   1011011101100101 == 0xb765 == 46949
&  1111111100000000 == 0xff00 == 65280
-----------------------------------------
   1011011100000000 == 0xb700 == 46848
```

Los operadores de corrimiento desplazan los bits de su operando izquierdo el número de veces especificado por su operando derecho:

```
x << 2
```

desplaza a *x* dos posiciones a la izquierda. El operador >> sustituye los bits desplazados con el bit de signo; el operador << los sustituye con ceros. Por ejemplo:

```
0x672f << 5 == 0xebc0
```

pues

```
0110011100101111 << 5 == 1110010111100000
        0x672f        ==          0xebc0
```

El ejemplo de la figura 3.2 despliega la representación binaria de un entero sin signo de dieciséis bits.

3.10 El operador condicional

El operador condicional de C++ se puede utilizar para evaluar

```
#include <stdio.h>
main() {
     unsigned x;
     scanf("%u", &x); // cin se utiliza para leer valores
     printf("%u == ", x);
     for (int i = 0; i < 16; i++) {
          if (x & 0x8000) // si el primer bit está encendido
               putchar('1');
          else
               putchar('0');
          x = x << 1;
     }
     printf("B\n");
}
```

Fig 3.2. Representación binaria de un entero de dieciséis bits.

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; // z = max(a,b)
```

El ejemplo para desplegar la representación binaria del entero sin signo se puede escribir también como se especifica en la figura 3.3.

3.11 Precedencia y evaluación

En este capítulo se presentaron algunos de los operadores de C++; posteriormente se describirá el modo de operación del resto de ellos. La tabla 3.1 muestra la precedencia y asociatividad de todos los operadores (la precedencia va decreciendo conforme avanza la

tabla; -> indica asociatividad a la derecha, <- asociatividad a la izquierda.

```
#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 << "B\n";
}
```

Fig. 3.3. Representación binaria de un entero de dieciséis bits.

Operador	Asociatividad
::	
() [] ->	->
sizeof	<-
! ~ ++ -- - (tipo) * & - + new delete	<-
* / %	->
.* ->*	->
+ -	->
<< >>	->
< <= > >=	->
== !=	->

Tabla 3.1. Precedencia y asociatividad de los operadores de C++.

Operador	Asociatividad
&	->
^	->
|	->
&&	->
||	->
? :	<-
= op=	<-
,	->

Tabla 3.1. Procedencia y asociatividad de los operadores de C++.

Capítulo 4

Control de flujo

Las acciones básicas de un programa en un lenguaje de alto nivel se describen mediante *proposiciones* (*enunciados* u *oraciones*). Basta con tres tipos de proposiciones para escribir cualquier programa: proposiciones para evaluar expresiones, proposiciones de iteración y proposiciones de decisión. C++ cuenta con proposiciones estructuradas de alto nivel para realizar esas tres funciones.

4.1 Proposiciones simples, compuestas y vacías.

La proposición básica de C++ es la evaluación de una expresión. La sintaxis de este tipo de proposiciones es

expresión;

En algunas ocasiones es necesario escribir varias proposiciones en alguna parte del programa en la que, por sintaxis, sólo se permite colocar una proposición. En estos casos, se puede utilizar una *proposición compuesta*, que consta de un conjunto de proposiciones agrupadas con llaves ({}). Una proposición compuesta,

también llamada *bloque*, es sintácticamente equivalente a una proposición sencilla.

Cuando la sintaxis requiere de una proposición, pero el no es necesario ejecutar ninguna acción, se puede emplear la proposición vacía, que se escribe como punto y coma (;). Por ejemplo:

```
for (int i = 0; getchar() != EOF; i++)
    ;
```

cuenta los caracteres que se proporcionen por la entrada estandard.

4.2 Proposición if.

La *proposición if* se usa para condicionar la ejecución de una o más proposiciones de acuerdo al resultado de cierta expresión. Esta proposición es la base para tomar decisiones dentro de un programa. Una proposición **if** se escribe de la siguiente forma:

```
if (expresión)
        proposición1
else
        proposición2
```

Si el valor de la *expresión* es diferente de cero, se ejecuta la primera proposición; en caso contrario, se ejecuta la segunda. La palabra **else** junto con la segunda proposición son opcionales. Si se coloca una proposición **if** dentro de otra, cada **else** se asocia

con el **if** anterior más cercano.

Las proposiciones del tipo

```
if (expresión1)
      proposición1
else
     if (expresión2)
          proposición2
     else
          if (expresión3)
               proposición3
          else
               ...
else
      proposición_n
```

generalmente se escriben como

```
if (expresión1)
      proposición1
else if (expresión2)
      proposición2
else if (expresión3)
     proposición3
...
else
      proposición_n
```

para evitar anidamientos excesivos.

El siguiente ejemplo ilustra el uso de proposiciones **if** en una implementación del algoritmo de *búsqueda binaria*:

```
int binaria(int x, int v[], int n)
// Encuentra x en v[0], ..., v[n-1]
{
     int inf = 0;
     int sup = n - 1;


     while (inf <= sup) {
          int med = (inf + sup) / 2;
          if ( x < v[med])
               sup = med - 1;
          else if (x > v[med])
               inf = med + 1;
          else
               return med;
     }
}
```

4.3 Proposición switch.

La *proposición switch* es otra forma de tomar decisiones dentro de un programa. Esta proposición permite comparar el resultado de una expresión entera contra una serie de valores constantes y ejecutar un conjunto de proposiciones de acuerdo al resultado de esa comparación. Una proposición switch tiene la forma

```
switch (expresión) {
      case constante1 :
            proposición1
            ...
            break;
      case constante2 :
            proposición2
            ...
            break;
      ...
      case constante_n :
            proposición_n
            ...
            break;
      default:
            proposición_m
            ...
}
```

La palabra *default* y las proposiciones que le siguen son opcionales.

En una proposición **switch** primero se evalúa la expresión que se encuentra entre paréntesis. El resultado de esa evaluación se compara contra la primer constante, si ambos valores son iguales se ejecutan **todas** las proposiciones que se encuentran después de dicha constante; en caso contrario la comparación se hace con la siguiente constante. El proceso continúa hasta encontrar una constante igual al valor de la expresión ó hasta llegar al final de la proposición. Cuando se encuentran presentes, las proposiciones del **default** se ejecutan sólo si ninguna comparación tuvo éxito.

La *proposición break* no forma parte de la sintaxis de la proposición **switch**; sólo se utiliza para evitar que se ejecuten las proposiciones que no se encuentran asociadas con la constante contra la que tuvo éxito la comparación mencionada (la proposición break desvía el flujo del programa al final del **switch**).

```
main() {
int   op1, op2; // operadores
char  op; // operador
int   r; // resultado de la expresión

scanf("%d %c %d", &op1, &op, &op2);
     switch (op) {
     case '+':
          r = op1 + op2;
          break;
     case '-':
          r = op1 - op2;
          break;

     case '*':
          r = op1 * op2;
          break;
     case '/':
          r = op1 / op2;
          break;
     default:
          printf("Operador incorrecto.\n");
          exit();
     }
     printf("%d %c %d = %d\n", op1, op, op2, r);
}
```

Fig. 4.1. Uso de la proposición switch.

Como ejemplo del uso de la proposición **switch**, la figura 4.1 presenta un programa que evalúa expresiones aritméticas binarias de la forma 5 + 3, 4 * 8, etc..

4.4 Proposición while.

C++ cuenta con tres estructuras de repetición: **while**, **for** y **do while**. La estructura **while** tiene la sintaxis

while (*expresión*)
    proposición

La *proposición while* funciona de la siguiente manera: primero evalúa la *expresión* entre paréntesis, si el resultado es diferente de cero, ejecuta la *proposición*. El proceso se repite mientras la evaluación de como resultado un número diferente de cero. Por ejemplo

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;
```

salta todos los blancos que encuentre en el archivo de entrada estandard.

4.5 Proposición for.

La *proposición for* es una forma especial de un ciclo **while**:

```
for (proposición1; expresión1; expresión2)
                proposición2
```

equivale a

```
proposición1
while (expresión1) {
    proposición2
    expresión2;
}
```

La siguiente función devuelve la longitud de la cadena *s*:

```
int strlen(char s[]) {
    for (int l = 0; s[l] != '\0'; l++)
        ;
    return l;
}
```

4.6 El operador coma.

El *operador coma* generalmente se usa en conjunto con la proposición **for**. Este operador tiene la siguiente sintaxis:

expresión1, *expresión2*

El operador coma evalúa primera la expresión que se encuentra a su izquierda, después evalúa la segunda expresión y da como resultado el valor y tipo de esta última evaluación. Por ejemplo

```
i = (j = 4), (k = 5);
```

asigna 5, 4 y 5 a *i*, *j* y *k*, respectivamente. La siguiente función invierte el orden de los elementos de la cadena *s*:

```
void reverse(char s[]) { // invierte la cadena s
    int i, j;
    for (i = 0,j = strlen(s) - 1; i < j; i++, j--) {
        int c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

4.7 La proposición do while.

La *proposición do while* es la tercera estructura de repetición de C++. A diferencia de las estructuras **for** y **while**, la proposición

do while evalúa la condición de terminación después de ejecutar la proposición de su cuerpo. La sintaxis de la proposición **do while** es:

```
do
     proposición
while (expresión);
```

Como ejemplo, el siguiente programa realiza la conversión de un entero a una cadena de caracteres:

```
void itoa(int n, char s[]) {
     int signo;
     if(signo = (n < 0))
          n = -n;
     int i = 0;
     do {
          s[i++] = n % 10 + '0';
     } while ((n /= 10) > 0);
     if (signo < 0)
          s[i++] = '-';
     s[i] = '\0';
     reverse(s);
}
```

Generalmente se utilizan llaves en la proposición **do while** aún cuando su cuerpo solo tenga una proposición, pues de otra forma podría confundirse la parte **while** con un ciclo **while**.

4.8 Proposiciones break, continue y goto.

La *proposición break* se usa para salir de una estructura de repetición (**for**, **while** y **do**) o de la estructura **switch**. **Break** hará que el control se transfiera a la instrucción que sigue a la

estructura en que se usó. Esta proposición se puede usar para simplificar una condición de repetición complicada. Por ejemplo

```
while(1) {
     scanf("%f",&x);
     if (x < 0.0)
          break;
     // ....
}
```

abandona el ciclo **while** cuando lee un número negativo.

La *proposición continue* sólo puede ser usada en los ciclos **for**, **while** y **do**; esta proposición transfiere el control al final de la iteración actual. Por ejemplo

```
for (i = 0; i < N; i++) {
     if (a[i] < 0)
          continue;
     // ....
}
```

salta los elementos negativos del arreglo a.

La *proposición goto* tiene la sintaxis

goto *etiqueta;*

en donde *etiqueta* es un identificador válido. Esta proposición causa un salto incondicional a una proposición precedida por *etiqueta:*. **Goto** puede ser útil cuando se desea abandonar una estructura profundamente anidada. Por ejemplo:

```
while (...) {
     if (...)
          for (...)
               if (...)
                    goto fin;
     ...
}
fin: if (...)
...
```

Capítulo 5

Funciones

Tradicionalmente la modularidad de los sistema 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 tipos 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.

5.1 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 debe 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, se asume 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.

Una lista de parámetros vacía indica que la función no recibe parámetros (note que en el lenguaje C una lista de parámetros vacía indica que la función puede recibir cualquier número de argumentos y que éstos pueden ser de cualquier tipo). Tanto las variables definidas dentro del cuerpo de una función como sus parámetros son locales a ella.

5.2 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 invocación. Por ejemplo, el programa de la figura 5.1 tendrá

como salida

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

```
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);
}
```

Fig. 5.1. Paso de parámetros 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. En 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.

5.3 Prototipos

Todas las funciones deben declararse antes de que otra función las pueda invocar. La declaración de una función se hace mediante un *prototipo*, en donde se especifica el tipo de la función y el número y tipos de sus parámetros, por ejemplo:

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

que especifica que la función *newton_raphson* recibe dos parámetros de tipo **double** (*x0* y *presicion*) 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 presicion);
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, 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, entre ellos TurboC++, no especificar el prototipo de una función genera un error).

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

main() {
    double x = newton_raphson(3.0); // presición de 1e-6
    double y = newton_raphson(3.0, 1e-12); // presición 1e-12
    printf("x = %f   y = %f\n", x, y);
}

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

Fig. 5.2. Argumentos por omisión.

5.4 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 su

valor en la llamada a la función. En el ejemplo de la figura 5.2, la primer llamada a la función *newton_raphson* asigna un valor de $1x10^{-6}$ al parámetro *presicion*, mientras que la segunda le asigna un valor de $1x10^{-12}$.

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

5.5 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 las funciones sobrecargadas (en la versión 2.0 de C++ esto no es necesario). Para un ejemplo de esta característica del lenguaje revise la figura 5.3.

5.6 Funciones en línea.

Las *funciones en línea* se especifican modificando la definición de la función con la palabra reservada **inline**:

```
#include <stdio.h>

overload print; // el nombre 'print' se puede usar para
                // varias funciones (no se necesita en C++ 2.0)
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
     if (n < 0) {
          putchar('-');
          n = -n;
     }
     int i = n / 10;
     if (i != 0)
          printd(i);
     putchar(n % 10 + '0');
}

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

Fig. 5.3. Sobrecarga de funciones.

```
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.

5.7 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* ó *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 comision = 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 interface entre las funciones que las utilizan no se puede 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 8) anula la necesidad de las 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 comision = 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;
   // ...
}
```

5.8 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 tiene 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 5.4 produce la salida

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

```
main() {
     for (int i = 0; i < 10; i++)
          f();
}

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

Fig. 5.4 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 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 sóla vez, antes de que inicie la ejecución de *main*.

5.9 Inicialización.

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

```
int digitos[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.

Capítulo 6

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.

6.1 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 hecha 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 a las variables de tipo stack *s1* y *s2*.

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

```
s1.crea();           // inicializa el stack s1
s1.push(5.67);       // inserta 5.67 en 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 el 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. La figuras 6.1 y 6.2 ilustran este tipo de organización.

```
#ifndef 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 <= tope <= 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
```

Fig. 6.1. Stack.h: Declaración de la clase stack.

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.

```
#include "stack.h"  // Declaración de la clase stack.
#include <stdio.h>  // Declaraciones de fprintf y stderr.

void stack::crea(int nmax) {
// Crea un stack vacío 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];
     n = 0;
}

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

void stack::push(float x) {
// Inserta x en el stack.
     if (n == max) {
          fprintf(stderr,"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) {
          fprintf(stderr, "Error, stack vacío.\n");
          return 0.0;
     }
     return p[--n];
}

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

Fig. 6.2. Stack.c: Definición de los métodos de la clase stack.

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**.

6.2 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 a 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() { // vacío es un método en línea
          return n == 0;
     }
};

// mismas definiciones para push, pop, crea y destruye
```

6.3 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 ó invocar un método para un objeto que ya se haya destruido. 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 un sólo destructor y este no debe recibir parámetros. El ejemplo de la figura 6.3 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:

```
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];
     max = n = 0;
}

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

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

Fig. 6.3. Constructores y destructores.

```
stack s1(20); // crea s1 de 20 elementos
// ...
if ( ...) {
     stack s2 = s1; // ¡cuidado!, s2 y s1 son ahora el mismo
                    // stack
     // ...
} // el ámbito 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 cualquier objeto se invoca en el momento en el que se crea el objeto; el destructor se invoca al llegar al destruirlo. 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 6.2, a 6.6) implementa una calculadora de notación polaca utilizando una versión ligeramente modificada de la clase stack.

```
// 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
                          printf("División entre cero\n");
                     break;
                case '=':
                     printf("\t%f\n",st.peep());
                     break;

                case 'c':
                     st.limpia();
                     break;
                case MUYGRANDE:
                     printf("%20s... es muy grande\n",s);
```

```
                    break;
                default:
                    printf("Comando desconocido %c\n",tipo);
                    break;
            }
}
```

Fig. 6.2. Calc.cpp: Calculadora de notación polaca.

```
// SCANNER.H: Declaraciones para scanner.cpp
#include <stdio.h>
#define NUMERO   '0' // indicador de número
#define MUYGRANDE  '9' // la cadena es muy grande
getop(char *s,int lim);
```

Fig. 6.3. Scanner.h.

```
// SCANNER.CPP: Reconoce operadores y operandos
#include "scanner.h"

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

```
    } else {
          while (c != '\n' && c != EOF)
                c = getchar();
          s[lim - 1] = '\0';
          return MUYGRANDE;
    }
}
```

Fig. 6.4. Scanner.cpp: Reconocimiento de los tokens de la calculadora.

```
// STACK.H: Encabezado para la clase stack.
class stack {
    int n;          // Número de elementos
    int nmax;       // Número máximo 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
};
```

Fig. 6.5. Stack.h: Declaración de la clase stack.

```
//STACK.CPP: Métodos de la clase stack.
#include "stack.h"

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

stack::~stack () {
    delete[nmax] d;
}

double stack::push(double f) {
```

```
        if (n == nmax ) {
                printf("Error: stack lleno.\n");
                return 0;
        } else {
                d[n++] = f;
                return f;
        }
}

double stack::pop(void) {
        if (n == 0) {
                printf("Error: stack vacío.\n");
                return 0;
        } else {
                return d[--n];
        }
}

double stack::peep(void) {
        if (n == 0) {
                printf("Error: stack vacío.\n");
                return 0;
        } else
                return d[n-1];
}

void stack::limpia(void) {
        n = 0;
}
```

Fig. 6.6. Stack.cpp. Métodos de la clase stack.

6.4 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 cual 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 (*ratón*) no permite que se defina más de una instancia de ella:

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

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

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

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

6.5 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 procesoB 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 el atributo privado hora
}
```

Los amigos 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 de la clase X
     // ...
}
```

Capítulo 7

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 abstracto. Este aspecto de C++ se discute en las siguientes secciones.

7.1 Definición de operadores.

El ejemplo de la figura 7.1 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 7.3).

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

  cadena(cadena &s) {  // Este constructor se explica más tarde.
        l = s.l;
        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;
}
```

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

Fig. 7.1. Definición de la clase cadena, versión 1

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));    // 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 b = "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 7.2 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.

```
#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 más tarde.
        l = s.l;
        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
  char operator [] (int i); // elemento i-ésimo de la cadena
  cadena operator () (int n, int m); // m caracteres 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;
}
```

```
char cadena::operator [](int i) {
    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;
}
```

Fig. 7.2. Definición de la clase cadena, versión 2.

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

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

no es equivalente a

```
s = s + t        // s.operator=(s.operator+(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.

7.2 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
      // ...
 }
```

7.3 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 tiene 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 tratar 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.);
}
```

7.4 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+(t(char *)))
```

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*, haciendola amiga de la misma para que pueda usar sus miembros

privados. La versión de la clase con esta última modificación se muestra en la figura 7.3.

```
#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);
  char 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;
```

```
}

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 i) {
  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;
}
```

Fig. 7.3. Definición de la clase cadena, versión 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 <stdio.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,*s2 = s.tope->sig;s2; s2 = s2->sig){
      nodo *tmp = new nodo;
      tmp->d = s2->d;
      s1->sig = tmp;
      s1 = tmp;

    }
    s1->sig = 0;
  }
}

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 = x;
  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;
}

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, *s2 = 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 = t->sig)
          printf("%f ", t->d);
     printf("(NULL)\n");
}
```

Fig. 7.4. Definición de la clase stack.

Capítulo 8

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.

8.1 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 del 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 perímetro
     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 *poligono*; 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 el de un polígono). En vez de iniciar desde cero la construcción de la clase *cuadrado*, podemos aprovechar todo el código de la clase *poligono* mediante el proceso de herencia ó derivación (la clase *poligono* 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 un 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 *poligono::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 poligono
     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 *poligono*) 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
     // ...
}
```

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 8.1 muestra la declaración de las dos clases anteriores.

Una características 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.

8.2 Constructores.

Como ya se mencionó más arriba, una clases 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, pasandole parámetros si los necesita:

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

```
struct punto {
     unsigned x, y;
};

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);
     unsigned perimetro();
};

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

Fig. 8.1. Derivación

```
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
     // ...
}
```

8.3 Polimorfismo y métodos virtuales.

El polimorfismo permite manipular objetos de diferentes clases (relacionadas por herencia) como 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 poligono
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
                              // poligono::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
      // poligono de tal forma que el compilador pueda escoger
      // el método adecuado al invocar a perimetro con un
      // apuntador a poligono.
};
```

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

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