UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACONAL DE ESTUDIOS PROFESIONALES ACATLAN

"TURBO C++ FOR WINDOWS" PARA PROGRAMACION ORIENTADA A OBJETOS; DESCRIPCION Y ALGUNAS APLICACIONES

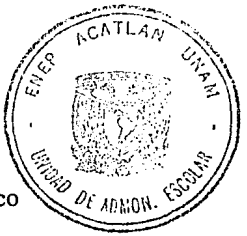
T E S I S

Que para obtener el título de:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

Presenta:

RUBEN ADRIAN SALINAS VERDUZCO



Naucalpan, Edo. de México Julio 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES "ACATLAN"

DIVISION DE MATEMATICAS E INGENIERIA
PROGRAMA DE ACTUARIA Y M.A.C.

SR. RUBEN ADRIAN SALINAS VERDUZCO
Alumno de la Carrrera de Matemáticas
Aplicadas y Computación,
P r e s e n t e

De acuerdo a su solicitud presentada con fecha 23 de noviembre de 1992, me complace notificarle que esta Jefatura tuvo a bien asignarle el siguiente tema de tesis: ""TURBO C++ FOR WINDOWS" PARA PROGRAMACION ORIENTADA A OBJETOS; DESCRIPCION Y ALGUNAS APLICACIONES", el cual se desarrollará como sigue:

INTRODUCCION
CAP. I PRINCIPIOS BASICOS DEL LENGUAJE C++ EN PROGRAMACION ORIENTADA A OBJETOS.
CAP. II CLASES Y OBJETOS.
CAP. III FUNCIONES Y OPERADORES
CAP. IV FUNCIONES VIRTUALES Y POLIMORFISMO
CAP. V APLICACIONES Y ANALISIS
CONCLUSIONES
BIBLIOGRAFIA

Asimismo fué designado como Asesor de Tesis la Lic. Angélica Lozano Cueyas, Profesora de esta Escuela.

Ruego a usted tomar nota que en cumplimiento de lo especificado en la Ley de Profesiones, deberá presentar servicio social durante un tiempo mínimo de seis meses como requisito básico para sustentar examen profesional, así como de la disposición de la Coordinación de la Administración Escolar en el sentido de que se imprima en lugar visible de los ejemplares de la tesis el título del trabajo realizado . Esta comunicación deberá imprimirse en el interior de la tesis.

A T E N T A M E N T E
"POR MI RAZA HABLARA EL ESPIRITU"
Acatlán, Edo. Méx. mayo 27 de 1994.

ACT. LAURA MA. RIVERA BECERRA
Jefe del Programa de Actuaría
y M.A.C.

JEFATURA DEL PROGRAMA DE ACTUARIA Y MATEMÁTICAS APLICADAS Y COMPUTACIÓN

bt

INDICE

INTRODUCCION 1

CAPITULO I
PRINCIPIOS BASICOS DEL LENGUAJE C++ EN PROGRAMACION ORIENTADA A OBJETOS (POO) 3

1.1 Orígenes de la programación orientada a objetos y del lenguaje C++ 3
1.2 Qué es la programación orientada a objetos 5
1.3 Breve descripción del ambiente Windows 8
1.4 Algunos de los aspectos fundamentales de C++ 10
1.5 Constructores y destructores 16

CAPITULO II
CLASES Y OBJETOS 20

2.1 Parámetros en constructores 20
2.2 Funciones Amigas 23
2.3 Funciones en línea para clases 26
2.4 Herencia múltiple 29
2.5 Arreglos y apuntadores en objetos 40
Arreglos de objetos 40
Apuntadores a objetos 41

CAPITULO III
FUNCIONES Y OPERADORES 44

3.1 Funciones constructoras sobrecargadas 44
3.2 Inicialización dinámica de constructores 45
3.3 Sobrecarga en operadores 47
3.4 Funciones operador amigas 51
3.5 Parámetros por referencia 56

CAPITULO IV
FUNCIONES VIRTUALES Y POLIMORFISMO 60

Apuntadores a tipos derivados 60
4.1 Funciones virtuales 63
4.2 Funciones virtuales puras y su relación con los tipos abstractos 69
Ataduras (Binding) 70
4.3 Constructores y destructores en clases derivadas 71
4.4 Múltiples clases base 71

CAPITULO V
APLICACIONES Y ANALISIS 73

5.1 Aspectos fundamentales del sistema operativo Windows 73
5.2 Programación con ObjectWindows 76
5.3 Procesamiento de Mensajes 79
Como generar un mensaje WM_PAINT 92
5.4 Cuadros de mensajes, menús, y cuadros de diálogo 95
Cuadros de mensajes 95
Creando Menús 101
Cuadros de diálogo 111
Controles de lista 115
Controles de edición 117

CONCLUSIONES 120

BIBLIOGRAFIA 122

ANEXO 123

INTRODUCCION

Este trabajo trata la técnica de la Programación Orientada a Objetos (POO denominada en adelante) con C++ para el ambiente operativo Windows de modo introductorio, básico y un poco como referencia, sin embargo es necesario tener algunos conocimientos previos para su mejor entendimiento. En principio se requiere saber programar en lenguaje C, esto es, conocer la sintaxis y estructuración; conocer el ambiente Windows es deseable pero no indispensable. En general el enfoque de todos los capítulos es didáctico, analítico y de referencia.

El principal objetivo de este trabajo es aportar una referencia de la programación orientada a objetos y un análisis de sus elementos utilizando el lenguaje C++ bajo el ambiente gráfico Windows.

El capítulo uno contiene una breve historia de los lenguajes de computación como antecedente de la POO, una sencilla descripción del ambiente operativo Windows, una descripción general de lo que es la POO y una introducción al lenguaje C++.

En el segundo capítulo se tratan más a fondo las clases y los objetos, sus propiedades, elementos y gramática; cómo, en forma general, agregar objetos y funciones a las clases y utilizar argumentos o parámetros en las funciones de las clases, la manera en que se controla el acceso a los objetos y funciones como datos de una clase, la relación que existe entre las estructuras y uniones formando parte de la programación estructurada con las clases por parte de la programación orientada a objetos, qué es y cómo crear herencia múltiple y el poder utilizar arreglos y apuntadores siendo características de un ambiente híbrido de programación.

En el capítulo tres se describen qué son y cómo utilizar las funciones y los operadores para funciones constructoras en programación orientada a objetos, la inicialización dinámica de los objetos y su aplicación a los constructores descritos en este capítulo; cuáles son los operadores utilizados en POO y una de las características más poderosas en C++ para la POO, las funciones operador.

El capítulo cuarto define y ejemplifica el polimorfismo con pequeñas aplicaciones junto con las funciones virtuales, así como su uso y relación con los tipos abstractos; cómo utilizar apuntadores en funciones simples para lograr tipos abstractos de datos; se utilizan y describen clases derivadas, múltiples clases básicas, constructores y destructores .

Estos capítulos se enfocan principalmente a la referencia de los elementos principales de la POO en el lenguaje de programación. La importancia de estos capítulos es el entender la gramática y sintaxis de los elementos de la POO en C++ para poder aplicarlos y analizarlos correctamente en el siguiente capítulo.

En el capítulo cinco se explica más a fondo lo que es el ambiente operativo Windows, se introduce a la programación con ObjectWindows y se desarrollan pequeñas aplicaciones con las partes y componentes principales de Windows, como ventanas, cuadros de mensaje, menús y cuadros de diálogo. Paralelo a la programación de los ejemplos se da una breve explicación y un análisis de su funcionamiento.

La información contenida en este capítulo es la necesaria para crear las aplicaciones Windows más comúnes, sin embargo no se discuten todos los aspectos del ambiente operativo Windows, ya que el API (siglas en inglés de Interface de Programación de Aplicaciones) cuenta con más de 600 funciones, y la finalidad que se persigue es utilizar la POO soportada en C++ para Windows.

El anexo contiene el listado completo de la última aplicación realizada en el capítulo cinco, la cual ejemplifica la creación de los principales componentes de un programa para Windows combinando un menú con algunos cuadros de diálogo, botones, controles de lista y controles de edición.

La notación utilizada en todos los capítulos es el tipo de letra Times para el texto, `Courier` para el código de los programas, **Times Negrita** para la referencia de variables, palabras reservadas, títulos y subtítulos en el texto y el tipo *Times Itálico* para la referencia de los prototipos de las funciones, estructuras y uniones.

Por último quiero mencionar que el material contenido en este trabajo es suficiente para iniciar en la programación orientada a objetos en C++ y con el uso del ObjectWindows para la programación de aplicaciones bajo Windows, sin embargo es necesario estudiar los manuales de Borland para C++ así como las referencias para programadores de Windows de Microsoft, ya que Windows es un ambiente operativo bastante complicado y entre más conocimiento se tenga de él, mejor se desarrollarán las aplicaciones.

CAPITULO I

PRINCIPIOS BASICOS DEL LENGUAJE C++ EN PROGRAMACION ORIENTADA A OBJETOS (POO)

1.1 Orígenes de la programación orientada a objetos y del lenguaje C++

La programación orientada a objetos (POO) es una nueva forma de mejorar la programación, las técnicas de ésta han evolucionado enormemente desde la invención de la computadora, la primera razón para estos cambios es el incremento en la complejidad de los programas. Por ejemplo, cuando las computadoras fueron inventadas, la programación o codificación era en código binario, conocido como lenguaje máquina a través de un panel de control y los programas utilizaban varios cientos de instrucciones para trabajar más o menos correctamente. Como el código y la exigencias de los programas crecieron, el lenguaje ensamblador fue inventado como un remedio a las necesidades temporales de los programadores; éste se basa en representaciones simbólicas de instrucciones binarias llamadas nemónicos pero, aún resultaban demasiado extensos los códigos y muy complicados para depurar o modificar. Como los programas continuaban extendiéndose, se introdujeron los programas en lenguaje de *alto nivel* los cuales utilizaban más herramientas e instrucciones para trabajar. El primer lenguaje de este tipo fue FORTRAN, y mientras éste dio un muy impresionante primer paso, era un lenguaje con muchas complicaciones para entender con claridad y facilidad los programas.

En la década de los 60's se inició la *programación estructurada*, método utilizado por lenguajes como C y Pascal. Utilizando lenguajes estructurados surgió por primera vez la posibilidad de crear programas moderadamente complejos con cierta facilidad. De cualquier manera, cuando un proyecto crecía en forma desmesurada, se volvía imposible de manejar y, en algún punto, esta complejidad excedía lo que un programador podía controlar tan sólo con programación estructurada.

Ante esta situación, los desarrolladores de software han encontrado en la programación orientada objetos una novedosa herramienta para combatir las deficiencias de la programación estructurada.

La mayoría de estos programadores aún piensan que la POO es un invento de la década de los 80's pero en realidad, en 1992, cumple 25 años de existencia. Dentro del círculo de la programación se está de acuerdo en que el primer lenguaje orientado a objetos es SIMULA (mejor conocido hoy en día como SIMULA 67) en el cual se basan prácticamente los conceptos fundamentales de la POO.

La historia de SIMULA comienza a finales de los 40's y principios de los 50's cuando se contrataron a dos noruegos, Kristen Nygaard y Ole-Johan Dahl, en el Departamento de defensa de Noruega para los cálculos de absorción y resonancia

relacionados con la construcción del primer reactor nuclear noruego en el que se involucró el método de Monte Carlo para la simulación. En el campo de la simulación tuvieron serios problemas para encontrar un poderoso y flexible vehículo para estructurar el modelo y el sistema en el que trabajaban. En 1960 Nygaard se cambio al Centro de Cómputo de Noruega (NCC) donde posteriormente, junto con Dahl y Bjrn Myhrhaug, encontró que muchos proyectos civiles tenían problemas metodológicos similares a los que tuvó en el campo militar. Con estas experiencias, Nygaard identificó los conceptos y creó un lenguaje llamado SIMULA 1.

Con el contrato entre NCC y UNIVAC, la computadora UNIVAC 1107 contenía un compilador de ALGOL 60 y esta confluencia de conceptos, máquina y plataforma lingüística presentaba los elementos principales para la implementación de SIMULA 1.

Posteriormente se desarrollaron importantes características; el proyecto de renovación para ALGOL 60 fue abandonado por la construcción de un más poderoso concepto que reunía bloques de datos y acciones de procesamiento en la noción de encapsulación y objetos. Tony Hoare contribuyó con algunas técnicas de seguridad para complementar el acceso externo a los atributos de los objetos, con esto se le daba seguridad total en el tiempo de corrida con la más económica técnica de verificación en tiempo de compilación. Le era posible reconocer que las clases de objetos muchas veces tenían propiedades en común, con lo que se sugirió la *factorización*, esencia de lo que hoy se conoce como herencia. Percibía que las clases de objetos podían estar relacionadas o siempre eran idénticas en su forma y comportamiento por lo que su realización concreta podría ser muy similar, lo que llevó al concepto y noción de unión dinámica.

El producto final de estos desarrollos fue la definición de un nuevo lenguaje, llamado SIMULA 67, que estaba muy cerca de ser un superconjunto de ALGOL 60 pero provisto con nuevos y poderosos elementos con toda la filosofía de la programación orientada a objetos y reemplazando algunos de los conceptos más débiles por unos más útiles. Así, lo que comenzó por un dialecto de ALGOL, SIMULA era ya un lenguaje de propósito general que ofrecía capacidades de simulación como una aplicación de sus conceptos básicos.

De esta forma 1967 marcó el final del período de gestación de SIMULA y se abrió una nueva posibilidad para implementar lenguajes, compiladores, aplicaciones e instituciones dedicadas al desarrollo y consolidación de la POO.

Algunos lenguajes derivados de SIMULA son :

• ***Eiffel.-*** Diseñado por Bertrand Meyer miembro de la asociación de usuarios de SIMULA y principal usuario de SIMULA durante su estancia en el consejo francés de electricidad.

• ***Smalltalk.***- Debe su existencia a Alan Kay, cuyas perspectivas fueron profundamente influenciadas por los conceptos que percibió él en una muy temprana implementación de SIMULA.

• ***Ada.***- Esta idea es sustentada por el hecho de que su diseñador, Jean Ichbiah, dirigió un equipo que implementó un subconjunto de SIMULA.

• ***C++.***- El orígen del lenguaje C++ es la invención del lenguaje C implementado por Dennis Ritchie por primera vez en una DEC PDP-11 usando el sistema operativo UNIX V, el lenguaje es resultado del proceso de desarrollo de un viejo lenguaje llamado BCPL, creado por Martin Richards, el cual influenció a Ken Thompson en la invención de un lenguaje llamado B el cual llevó a la creación de C en los 70's. Por muchos años el standard de C fue la versión implementada para UNIX V, pero la creciente popularidad de las microcomputadoras originó la creación de un gran número de implementaciones y versiones de C dentro de las cuales los códigos fuente fueron altamente compatibles entre sí, pero aún existian discrepancias debido a la falta de un standard . En 1983 ANSI creo un comité para estandarizar el lenguaje C, lo cual finalmente fue logrado en 1990, y Turbo C creó una implementación completa del ANSI C.

El C++ es una versión expandida o un superconjunto de C. El lenguaje C es poderoso y muy flexible por lo que ha sido utilizado para crear algunas de las aplicaciones más importantes de los últimos 15 años. Aún así muchas de las nuevas aplicaciones tienen de 25,000 a 100,000 líneas en un sólo proyecto, lo que las hace muy difíciles de manipular en su totalidad. Pensando en este problema en 1980 en los laboratorios Bell en Nueva Jersey, E.U., Bjarne Stroustrup llegó a una solución mediante la adición de muy variadas características y extensiones al lenguaje C, inicialmente llamado "C con clases", al que en 1983 se le cambió el nombre a C++. Muchas de estas adiciones son para soportar programación orientada a objetos (POO), las cuales son inspiradas por SIMULA 67 y con esto C++ representa una gran combinación de lenguaje estructurado y orientado a objetos, por lo que se le clasifica como un lenguaje híbrido con gran potencial para la reutilización de código.

C++ ha sido revisado en 3 ocasiones desde que fue inventado: la primera vez en 1985, la segunda en 1989 (versión 2.0) y la última en 1991 (versión 2.1). Borland implementa la versión 2.0 por primera vez en mayo de 1990 y la nueva versión 2.1 en el Turbo C++ 3.0 para Windows.

1.2 Qué es la programación orientada a objetos

En cada corriente de programación, los métodos utilizados fueron creados para ayudar al programador a realizar proyectos cada vez más complejos y cada nueva corriente o método de programación tomaba los mejores elementos de su antecesor y los incorporaba a su metodología. Hoy en día, muchos proyectos están cerca del punto en que

la programación estructurada no sea suficiente para el fin requerido. Para resolver este problema, es posible implementar programación orientada a objetos (POO).

La programación orientada a objetos ha tomado las mejores ideas de la programación estructurada y las ha combinado con nuevos y poderosos conceptos que permiten vislumbrar una forma mejorada de trabajar en la programación. La POO permite descomponer más fácilmente un problema en subgrupos de partes relacionadas del problema. Después, utilizando el lenguaje, se pueden transladar estos subgrupos en unidades llamadas objetos.

La programación orientada a objetos (POO) es la forma de aprovechar el trabajo del programador tratando de hacer códigos reutilizables. Esta idea parte de la necesidad de lograr que a los programas se les pueda dar mantenimiento en lugar de tener que desarrollarlos nuevamente para adecuarlos a las nuevas necesidades.

En cuanto al mantenimiento, cada cambio funcional a un sistema ya existente implica un gran problema, debido a que los sistemas están firmemente ligados a los datos. Cualquier cambio a las estructuras de información afecta a todo el sistema. Por ejemplo, en el caso de que no se puedan agregar nuevas mercancías porque al no estar definidos los nuevos tipos, no se podrían facturar. Por otro lado, si el sistema no está diseñado en forma modular, un cambio mínimo puede tener repercusiones no dimensionables hasta que se libera el cambio y ocasiona graves problemas en la producción, ya que se daba por completado el nuevo desarrollo.

La reutilización de código se alcanza haciendo uso de una facultad llamada herencia. Un objeto descrito por su clase es la unión de datos privados llamados atributos y un conjunto de operaciones o servicios que hacen uso de ellos. Conceptualmente, se habla de clase como la abstracción del comportamiento del objeto, por lo que una clase es meramente un esqueleto de objeto. Un objeto se entiende como la implementación de la clase; contiene valores en sus atributos y se le llama también instancia de clase. Al recibir comportamiento (total de atributos y servicios) de otro objeto del sistema se convierte en su hijo. Se reutiliza toda la técnica aplicada en desarrollar al objeto padre, y al añadirle comportamiento (servicios o atributos) se le da funcionalidad expandida. A su vez, este objeto puede ser heredado por otros, creando una jerarquía de objetos que eventualmente formará una biblioteca de código reutilizable. Al liberar un objeto a esa biblioteca, se le considera totalmente probado y su comportamiento está totalmente descrito. El código está totalmente disponible a cualquier otro programador que puede, con confianza, utilizarlo. Si requiere modificarlo, basta con que herede el comportamiento necesario y así crear un objeto nuevo, haciendo sumamente fácil extender la biblioteca del sistema.

El comportamiento de un objeto (los servicios provistos) y sus atributos deben estar aislados de los demás, y se acceden únicamente mediante una invocación al objeto. A esto se le llama mandar un mensaje, lo que significa hacer un llamado a un servicio del objeto en cuestión, por lo que los términos mensaje y servicio (o método) se utilizan indistintamente. Los objetos con quienes tiene relación de acción, es decir de quienes recibe o envía mensajes, no deben saber cómo se implementa un comportamiento, o de

dónde se obtiene el atributo solicitado. Si un objeto esconde información a otros objetos que requieran interactuar con él, cuando sufra una modificación el efecto del cambio lo resiente únicamente este objeto, así como sus servicios y atributos; no se expande a través del sistema, con lo que se logra un alto grado de modularidad.

Dado que en un lenguaje orientado a objetos cualquier cosa debe considerarse como un objeto, y si sabemos que el comportamiento de un objeto es su propia responsabilidad, es fácil pensar que se puede solicitar el mismo servicio o mandar el mismo mensaje a dos objetos diferentes, y asumir que cada uno de ellos sabrá manejarlo o darle respuesta. Esto es polimorfismo y aunado a todo lo descrito tiene el efecto directo de hacer que un cambio o extensión a una aplicación sea extremadamente sencillo.Por ejemplo, se puede pedir a un objeto que controle un disco duro y a otro que tome el control de una impresora para escribir un texto que recibe como parámetro. El resultado en el primer caso será guardar un archivo en disco y en el segundo caso tener una impresión del archivo. Es completa responsabilidad del objeto receptor saber qué hacer con el mensaje y nadie más debe preocuparse de cómo se lleva a cabo la operación.

Otro ejemplo sería el cambiar una impresora por un plotter (o graficador), no es necesario cambiar la aplicación entera, sino sólo asegurarse de que el nuevo objeto ***plotter*** pueda dar servicio al mensaje ***escribe.*** Si se quiere imprimir tanto en plotter como en impresora, se manda el mismo mensaje de ***escribe*** a ambos objetos, y la extensión o ampliación deseada está hecha.

Dado que cada objeto sabe responder a los mensajes como es conveniente para él, por ejemplo puede evitarse el recordar el tipo del dato que se envía como parámetro, pues ese dato sabrá hacer lo correcto en una operación. Es posible aplicar la operación ***suma*** a objetos dispares, como sumar números o sumar caracteres. En 3+6, por ejemplo, el mensaje + se envia al objeto número **3**, el cual recibe un **6** como parámetro y el resultado es que al objeto **3** se suman **6** unidades quedando un **9** al final de la operación.

Ahora definiremos en forma general los conceptos más importantes en la programación orientada a objetos :

Objetos.- La singularidad y más importante característica de un lenguaje orientado a objetos son los objetos en sí. Explicado sencillamente, un objeto es una entidad lógica que contiene datos y código para manipular los datos. En un objeto, mucho del código y/o datos pueden ser privados para el objeto e inaccesibles para cualquier parte fuera del objeto. Otro código y/o datos pueden ser públicos y accesibles para cualquier otra parte del programa. Haciendo privada alguna parte de los objetos, éstos pueden prevenir que cualquier parte del programa sin relación con el objeto, pueda hacer alguna modificación accidental o algún uso inadecuado del mismo. El ligar o relacionar el código y los datos de esta forma es conocida como *encapsulación.*

Para todos los casos y propósitos, un objeto es una variable de tipo definido por el usuario. Suena extraño pensar en un objeto, el cual liga código y datos, como una

variable. Pero en la programación orientada a objetos, éste es precisamente el caso; cuando se define un objeto, se está creando implícitamente un nuevo tipo de datos.

Polimorfismo.- La programación orientada a objetos soporta *polimorfismo*, lo cual esencialmente significa que un nombre puede ser usado para relaciones complicadas pero para muy diferentes propósitos. El polimorfismo da un nombre para ser usado específicamente en una clase general de acción. De todas formas, dependiendo de que tipo de dato es introducido, una parte específica del caso general es ejecutada. Por ejemplo se tiene un programa que define tres tipos de colas; una cola es usada para valores de tipo entero, una para reales y otra para enteros largos (32 bits); como se utiliza polimorfismo se pueden crear tres conjuntos de funciones llamadas **push( )** y **pop( )** (poner y sacar valores para todos los tipos de colas). En este caso el compilador puede seleccionar la función correcta para cada cola dependiendo del tipo de valor con que se llame o haga referencia.

Los primeros lenguajes de programación orientada a objetos eran intérpretes, entonces el polimorfismo era en tiempo de ejecución o tiempo de corrida (run-time). En C++ el polimorfismo es soportado tanto en tiempo de corrida como en tiempo de compilación.

Herencia.- La herencia es el proceso mediante el cual un objeto puede tomar características de otro objeto; esto es importante porque de aquí parte el concepto de clasificación. Pensando en esto, muchos de los conceptos que se manejan en POO son creados mediante la clasificación jerárquica. Por ejemplo, un *jet* es parte de la clasificación *avión*, la cual es parte de la clasificación *transportes aereos*, la que a su vez es parte de una clasificación más general llamada *medios de transporte*. Sin el uso de clasificaciones, cada objeto puede hacer su propia definición de características. Pero cuando se usan las clasificaciones, un objeto sólo necesita definir algunas cualidades que lo hacen único en su clase. Este es el mecanismo de herencia que hace posible que un objeto sea parte específica de un caso más general.

1.3 Breve descripción del ambiente *Windows*

En 1981, IBM crea la primera computadora personal y Microsoft realiza el sistema operativo MS-DOS, con lo que comienza la revolución de las computadoras tipo PC. A través de los 80's, millones de usuarios aprenden a utilizar los comandos del MS-DOS y una gran variedad de aplicaciones. Para el final de la década, muchos usuarios tenían un procesador de palabras, una hoja de cálculo y/o utilizaban frecuentemente alguna aplicación con una base de datos. De hecho, estos usuarios habian estado buscando la forma más simple para intercambiar información entre las aplicaciones, un método que pudiera eliminar la necesidad de finalizar una aplicación antes de examinar la información contenida en otra.

En 1990, Microsoft introduce Windows 3.0, un programa diseñado para maximizar la productividad. Windows 3.0 hace las computadoras más fáciles de usar, las aplicaciones más fáciles de aprender y además posibilita el correr varias aplicaciones al mismo tiempo con la característica de poder intercambiar información entre ellas de una manera muy sencilla.

En 1992, Microsoft lanza Windows 3.1, el cual está provisto de un *linker* o ligador de objetos (llamado ***clipboard*** o portapapeles) para ayudar a compartir los datos entre las aplicaciones, *TrueType Fonts* (fuentes o tipos de letra) con los que es posible que los caracteres sean de diferentes tamaños e imprimirlos exactamente iguales de como aparecen en la pantalla; ayuda más completa y un tutorial en línea. Windows 3.1 contempla MULTIMEDIA e integra a sus accesorios algunas utilidades para su uso.

Windows es un ambiente gráfico que logra una forma de trabajo más personalizada con la microcomputadora personal. No sólo brinda más control sobre los trabajos, sino que permite utilizar al máximo las capacidades de la computadora sin los límites de memoria que anteriormente la restringían. Sus menus, íconos (símbolos) y cuadros de diálogo reemplazan los muchas veces crípticos comandos que DOS necesita.

En el ambiente de Windows se hace referencia a la pantalla como el escritorio, éste presenta todo el trabajo en áreas rectangulares que se denominan ventanas. Todos los trabajos de aplicaciones o documentos corren dentro de estas ventanas. Las aplicaciones se representan a través de íconos, que a su vez abren la ventana propia de esta aplicación.(figura 1.1)

Las ventanas se organizan en la superficie del escritorio de la misma manera que se organizan documentos sobre un escritorio real en una oficina. Por ejemplo para revisar una hoja de cálculo y un informe creados en dos aplicaciones distintas, se cambia simplemente el tamaño de las ventanas para presentarlas lado a lado sobre el escritorio o en pantalla. Si se desea cerrar momentaneamente una aplicación o dejar de utilizarla , la ventana se puede convertir en un ícono, pero la aplicación continúa ejecutándose y el ícono permanece en el borde inferior de la ventana hasta que se desee convertir nuevamente en ventana.

Con Windows es posible ejecutar varias aplicaciones a la vez y pasar rápidamente de una a otra, con simples pasos se puede ir de una base de datos a una hoja de cálculo y luego a un procesador de textos sin tener que esperar a salir de una aplicación e iniciar otra para entrar. Al trabajar con dos o más aplicaciones, ya sean diseñadas o no para Windows, se puede transferir información de una a otra, simplemente seleccionando la información que desea transferirse en una aplicación, copiandose en el *portapapeles* y luego pegandola en otra. De esta forma se pueden combinar trabajos hechos en distintas aplicaciones. Por ejemplo, se crea el texto para una invitación en un procesador de palabras, luego con una aplicación para gráficos se dibuja el mapa para localizar el lugar o sitio donde es la cita, se copia el mapa en el *portapapeles* y se pega junto al texto realizado en el procesador.

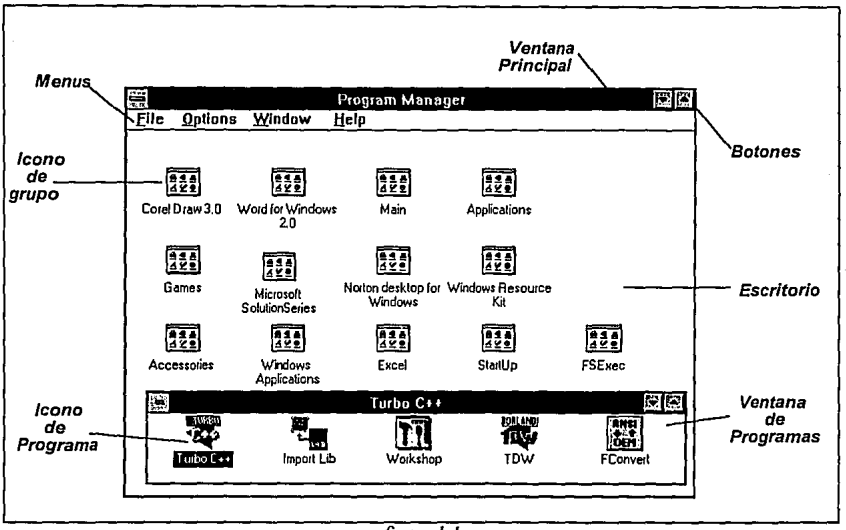


figura 1.1

Mientras se trabaja en Windows con una aplicación, se puede dejar de trabajar momentáneamente en un proyecto en cualquier instante para dedicarse a otras tareas. Por ejemplo, se tiene necesidad de insertar o agregar algunas tarjetas al *archivero*, especificar en el *calendario* la fecha de una reunión o usar la *calculadora* para realizar los cálculos que en ese momento se necesita, todo gracias a los accesorios que vienen integrados en Windows.(figura 1.2)

1.4 Algunos de los aspectos fundamentales de C++

Dado que C++ es un superconjunto de C, muchos de los programas en C son implícitamente programas C++ también (hay algunas pequeñas diferencias entre el ANSI C y C++, las cuales previenen una buena compilación de un programa en C en un compilador C++). Esto significa que se pueden escribir programas en C++ iguales a programas en C. De cualquier manera, es obvio que es mucho más eficiente hacer los programas en C++; es como poder escojer entre ir manejando en el periférico o por la lateral ya que se utilizan todas sus ventajas. Así , también es importante conocer algunas particularidades de C++.

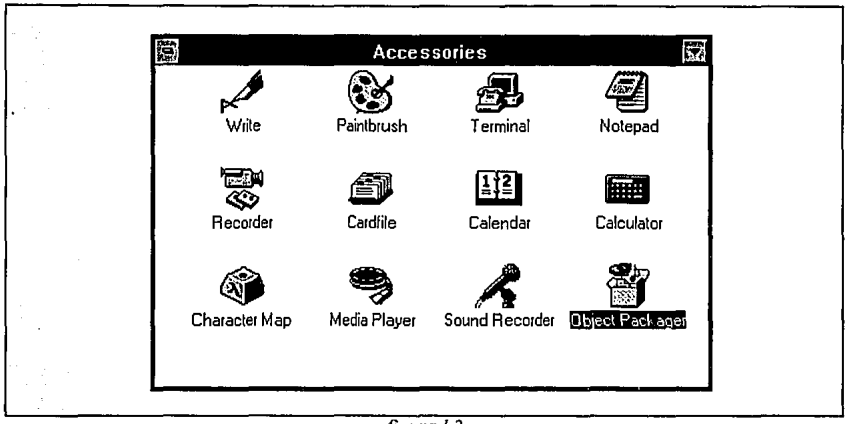


figura 1.2

Para observar algúnas diferencias tomaremos como ejemplo el siguiente programa:

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

void main(void) {
      int num;
      char palabra[80];

      cout << " Este es un ejemplo para desplegar en pantalla \n";
            // así se hacen lo comentarios en C++
            /* Pero se pueden hacer como en C también*/

      printf("Pero se puede utilizar printf ( ) si se desea\n");

      // En esta línea se introduce un número utilizando >>
      cout << "Introduzca un número : ";
      cin >> num;

      // Aquí se imprime en pantalla el número intoducido
      cout << "El número es " << num << "\n";

      // Lectura de una cadena de caracteres
      cout << " Introduzca una palabra (max. 80 letras) : ";
      cin >> palabra;

      // Ahora se imprime la cadena
      cout << palabra;

      return;
}
```

Como se puede ver, este programa es algo diferente a los programas en C; para empezar se incluye el archivo de cabecera IOSTREAM.H , este archivo es definido para C++ y utilizado para soportar las operaciones de entrada/salida en C++. STDIO.H es sólo para soportar la función **printf( )**; la cual no es necesaria para el sistema de entrada/salida de C++.

Las siguientes líneas con algo diferente son :

```
cout << "Este es un ejemplo para desplegar en pantalla\n";
      //así se hacen lo comentarios en C++
```

Aquí se pueden ver dos nuevas características de C++; primero cout << produce una salida o despliegue en pantalla con un retorno y salto de línea; y luego las dos diagonales // significan que todo lo que esté a la derecha es texto de comentario. En C++, el operador << tiene un doble papel. Sigue siendo un operador de corrimiento de bits a la izquierda, pero cuando es usado como en el ejemplo anterior, es un operador de salida.

La siguiente línea a analizar es:

```
cin >> num;
```

En C++, el operador >> aún es para corrimientos de bits a la derecha, pero cuando se usa como en esta línea, produce que se introduzca un valor (en este caso de tipo entero) para la variable **num** a través de la función **cin**, la cual por *default* es esperada del teclado. En general se puede usar **cin >>** para capturar cualquier tipo básico de variable y de cadenas. Las funciones estandard **cin** y **cout** son abiertas automáticamente cuando un programa en C++ comienza su ejecución.

También se pueden hacer algunas combinaciones como en la siguiente línea :

```
cout << "El número es " << num << "\n";
```

En esta línea se muestra como desplegar el valor de una variable, una presentación o constante alfanumérica y un salto de línea en la misma expresión. El resto del programa muestra el uso de **cin >>** y **cout <<** para cadenas de caracteres.

Como se puede ver en el programa y con otras que no, como el uso de scanf( ), cualquier instrucción de C funciona correctamente, pero en el caso de utilizar funciones de entrada/salida es más común el hacerlo con **cin** y **cout** para la programación en C++.

C++ incluye todas las palabras reservadas del lenguaje C y agrega 17 nuevas palabras reservadas que son :
asm, catch, class, delete, friend, inline, new, operator, overload, private, protected, public, template, this, throw, try, virtual.

De estas palabras reservadas, **catch, throw y try** son para aplicaciones más complicadas o avanzadas y **overload** es obsoleta, pero aún es aceptada para compilar programas antiguos sin errores. Como se sabe, estas palabras reservadas tampoco se pueden usar como nombre de variables o funciones.

La característica más importante en C++ es la clase. Para crear un objeto en C++ primero se debe declarar su forma general mediante la palabra reservada **class**. Una clase es sintácticamente similar a una estructura. Por ejemplo, definiremos un tipo llamado ***cola*** que será usado para crear un objeto llamado cola :

```
<include iostream.h>

// Aquí se crea la clase cola
class cola {
    int q[100];
    int inicio, fin;
public :
    int iniciar( );
    int poner(int i );
    int quitar( );

};
```

Una clase puede contener tanto partes públicas como privadas. Por omisión o *default*, todo lo que se define en una clase es privado, por ejemplo, las variables **q, fin** y **inicio** son privadas; esto significa que no pueden ser accesadas por ninguna función que no es miembro de la clase. Esta es una forma en que el encapsulamiento funciona y así se lleva certeramente el control de los datos mediante su declaración de forma privada. De la misma forma también se pueden declarar funciones de tipo privado, las cuales sólo pueden ser llamadas por miembros de la clase.

Para ser parte de una clase pública, esto es, ser accesible para otras partes del programa, se tienen que declarar las variables o funciones después de la palabra reservada **public**, con esto son accesibles para todas las demás partes del programa. Esencialmente, el resto del programa accesa objetos a través de estas funciones y datos públicos. Es importante mencionar que se debe limitar o eliminar el usos de variables públicas; esto es, se deben hacer todos los datos privados y el control de acceso a éstos a través de funciones públicas. En el ejemplo las funciones **iniciar( ), poner( )** y **quitar( )** son llamadas *funciones miembro* porque son parte de la clase **cola**. Sólo estas funciones declaradas en la clase tienen acceso a las partes privadas de esta clase.

Una vez que se define una clase, se puede crear un objeto de este tipo mediante el uso del nombre de la clase, de hecho, el nombre de la clase recibe un nuevo especificador de tipo de dato, por ejemplo, la siguiente línea crea un objeto llamado **cola_ent** de tipo **cola** :

```
cola cola_ent;
```

También se pueden declarar variables cuando es definida la clase, poniendo el nombre de la variable después de la llave izquierda (}) , de la misma forma que se hace para la declaración de estructuras. Así , en general, en C++ la palabra reservada **class** crea un nuevo tipo de datos que puede ser usado para crear objetos de este tipo. Un objeto es una instancia específica de una clase.

La forma general de declarar una clase es:

class *nombre_de_clase{*
    datos y funciones privadas
public:
    datos y funciones públicos
} nombre_del_objeto; // El nombre de los objetos, aquí, es opcional

Es importante entender que en C++ cuando se necesita decir algo acerca de las funciones al compilador se debe usar completamente la forma de funciones prototipo. Cabe hacer mención de que todas las funciones en C++ son prototipos, es decir, las funciones prototipo no son opcionales a diferencia del ANSI C.

Cuando se define una función miembro se le debe decir al compilador a qué clase pertenece la función mediante una clasificación del nombre de la función con el nombre de la clase de la cual es miembro. Por ejemplo, aquí se muestra una forma de codificar una función :

```
void cola :: poner( int i)
{
      if(fin==100)  {
      cout << " La cola esta llena ";
      return;
       }
       fin++;
       q[fin] = i;
};
```

El operador : : (operador de resolución de ámbito) indica al compilador que esta versión de **poner( )** pertenece a la clase **cola** o, de otro modo, que esta función **poner( )** es del ámbito **cola.** Como en C++ muy diferentes clases pueden usar el mismo nombre de funciones, el compilador sabe qué función pertenece a qué clase mediante el operador de resolución de clase y el nombre de la clase. Como la función **poner( )** es miembro de la clase **cola,** ésta tiene acceso directo a las variables **q, inicio** y **fin.**

Para llamar a una función miembro desde alguna parte de un programa que no es parte de la clase, se debe usar el nombre del objeto y el operador punto. Por ejemplo, aquí se llama a la función **iniciar( )** por el objeto **uno** :

```
cola uno,dos;
uno.iniciar( );
```

Aquí es muy importante entender que **uno** y **dos** son objetos separados, esto significa, por ejemplo, que la inicialización de **uno** no implica también la inicialización de **dos**. La única relación que tiene uno con dos es que son objetos del mismo tipo. Cada objeto de una clase contiene sus propias copias de los datos incluidos en la clase; esto significa, por ejemplo, que **inicio, fin y q** de **uno** son completamente separados de las copias de **dos** de las mismas variables.

También cabe mencionar que una función miembro puede llamar a otra función miembro directamente, sin utilizar el operador punto; sólo cuando una función miembro es llamada por código que no pertenece a la clase es que el nombre de variable y el operador punto deben utilizarse.

El siguiente programa muestra con detalles una aplicación de una cola con un arreglo mediante la clase **cola** y un máximo de 100 elementos :

```
<include iostream.h>

// Aquí se crea la clase cola
class cola {
    int q[100];
    int inicio, fin;
public :
    int iniciar( );
    int poner(int i );
    int quitar( );
};

// Esta función inicializa los indices de la cola
void cola :: iniciar ( )
/{
     inicio=fin=0;
}

//Con esta función se ponen elementos al final de la cola
void cola :: poner(int i) {
       if(fin==100) {
      cout << "La cola esta llena";
      return;
        }
        fin++;
        q[fin] = i;
}

// Con esta función se elimina el primer elemento de la cola
int cola :: quitar( )  // Con esta función se elimina el primer elemento
de la cola
{
         if (inicio==fin)  {
           cout << "La cola esta vacía";
```

```
            return 0;
        }
        inicio++;
        return q[inicio];
}

main ( )
{
      cola uno, dos;  // Crea dos objetos de tipo cola

      uno.iniciar( );
      dos.iniciar( );   // Se inicializan las variables

       uno.poner(20);
       dos.poner(10);

       uno.poner(27);
       dos.poner(17);

       cout << uno.quitar( ) << " ";
       cout << uno.quitar( ) << " ";
       cout << dos.quitar( ) << " ";
       cout << dos.quitar( ) << "\n";

       return 0;
}
```

Este programa desplegará en pantalla

20 10 27 17

Las partes privadas de un objeto son accesibles sólo por funciones que son miembros de ese objeto. Por ejemplo, una instrucción como ésta:
uno.inicio = 0;
no puede estar en alguna función **main( )** de un programa previo.

Muchos programas en C tienen la función **main( )** como la primera función en su código. Por el contrario el programa ejemplo de la cola, tiene definidas las funciones miembro antes que la función **main( )**. Esta no es una regla general, ya que las funciones pueden ser definidas en cualquier parte del programa, pero ésta es la forma más común de escribir código en C++. Para hacerlos más comprensibles actualmente en aplicaciones reales, las clases asociadas con un programa están normalmente contenidas en un archivo de cabecera o *header file*.

1.5 Constructores y destructores

Es muy común para muchas partes de un objeto, que estas partes requieran una inicialización antes de ser utilizadas. Por ejemplo, retomando la clase **cola** creada en el programa anterior, antes de que la cola pueda ser utilizada, las variables **inicio** y **fin**

tienen que comenzar con un valor de cero. Esto se hizo mediante la función **iniciar( )**. Como el proceso de inicialización es muy común, C++ provee algo para que los objetos puedan inicializarce ellos mismos al ser creados. Esta inicialización automática se hace a través de una *función constructora.*

Una función constructora es una función especial que es miembro de una clase y tiene el mismo nombre que esta clase. Por ejemplo, aquí se muestra como la clase **cola** es definida con una función constructora para inicialización :

```
// Aquí se crea la clase cola

class cola {
    int q[100];
    int inicio, fin;
public :
    cola( );    // Función constructora
    int poner(int i );
    int quitar( );
};
```

Para el constructor cola( ) no se especifica un tipo de valor para retornar porque las funcionas constructoras no pueden retornar valores.

Ahora la función constructora **cola( )** codificada queda de este modo :

```
//  Esta es la función constructora
cola :: cola( )
{
      llena= vacia= 0;
      cout << "La cola ha sido inicializada\n";
}
```

Como se puede ver, **cola( )** inicializa las variables **llena** y **vacia** en cero.

El constructor del objeto es llamado cuando el objeto es creado; esto significa que la inicialización de variables se ejecuta en el momento que se ejecuta la declaración del objeto. Pero, para objetos locales, el constructor es llamado cada vez que la declaración del objeto es encontrada y para objetos globales el constructor es llamado cuando comienza la ejecución del programa.

El complemento de los constructores son los destructores. En muchos casos, un objeto puede necesitar realizar alguna acción o acciones cuando éstos son destruidos. Por ejemplo, un objeto puede necesitar liberar memoria que tiene previamente ocupada; en C++ son las funciones destructoras las que controlan esta desactivación. El destructor tiene el mismo nombre que el constructor pero precedido por un tilde (~). Aquí se muestra cómo codificar una función destructora para el caso de la clase cola (que aunque no lo necesita, se utiliza para ejemplificar) y una versión mejorada del programa cola creado anteriormente :

```
// Aquí se crea la clase cola
class cola {
    int q[100];
    int inicio, fin;
public :
    cola( );   //  Función constructora
    ~cola( );  //  Función destructora
    int poner(int i );
    int quitar( );
};

//  Esta es la función constructora
cola :: cola( )
{
      llena= vacia= 0;
      cout << "La cola ha sido inicializada\n";
}

// Esta es la función destructora
cola :: ~cola( )
{
      cout << " La cola ha sido desactivada\n";
}

void cola :: poner(int i )  // Con esta función se ponen elementos al
final de la cola
{
      if(fin==100) {
      cout << "La cola esta llena";
      return;
        }
       fin++;
       q[fin] = i;
}

int cola :: quitar( )  // Con esta función se elimina el primer elemento
de la cola
{
         if (inicio==fin)  {
           cout << "La cola esta vacía";
             return 0;
         }
         inicio++;
         return q[inicio];
}

main ( )
{
      cola uno, dos;  // Crea dos objetos de tipo cola

      uno.iniciar( );
      dos.iniciar( );   // Se inicializan las variables
```

```
    uno.poner(20);
    dos.poner(10);

    uno.poner(27);
    dos.poner(17);

    cout << uno.quitar( ) << " ";
    cout << uno.quitar( ) << " ";
    cout << dos.quitar( ) << " ";
    cout << dos.quitar( ) << "\n";
    return 0;
}
```

Este programa despliega en pantalla lo siguiente :

```
La cola ha sido inicializada
La cola ha sido inicializada
20 27 10 17
La cola  ha sido desactivada
La cola  ha sido desactivada
```

Como se podrá ver a lo largo de los siguientes capítulos, constructores y destructores juegan un papel muy importante en el desarrollo de aplicaciones para Windows.

CAPITULO II

CLASES Y OBJETOS

2.1 Parámetros en constructores

Con frecuencia, cuando un objeto es creado, es necesario o deseable la inicialización de varios elementos de datos con valores específicos. Como se vió en el capítulo anterior, mediante el uso de una función constructora, es posible la inicialización de varias variables cuando el objeto es creado. De cualquier forma, en C++, el concepto de inicialización de objetos se expande hasta la inicialización de objetos específicos usando valores definidos por el programador. Esto se realiza pasando argumentos a una función constructora de objetos.

Como un simple ejemplo se utilizará la definición del objeto **cola** del capítulo anterior extendiendolo para aceptar un argumento el cual actua como ID (Inicio Determinado) de la **cola** :

```
// Aquí se crea la clase cola

class cola {
    int q[100];
    int inicio, fin;
    int quien;  // Retiene el número ID de la cola
public :
    cola(int id);   // Función constructora
    ~cola( );  //  Función destructora
    int poner(int i);
    int quitar( );
};
```

La variable **quien** es usada para tener el valor de inicio determinado, el cual puede identificar la cola. Este valor actual será determinado porque es pasado a la función constructora en la variable **id** cuando una variable de tipo **cola** es creada. La función constructora **cola** queda así :

```
//  Esta es la función constructora
cola :: cola(int id)
{
      llena = vacia = 0;
      quien = id;
      cout << "La cola ha sido inicializada\n";
)
```

Para pasar un argumento a la función constructora, se debe asociar el valor o valores que serán pasados con el objeto cuando este objeto está siendo declarado. En C++ existen dos alternativas para hacer esto; la primera es como sigue :

```
cola uno = cola (101);
```

Esta declaración crea una cola llamada **uno** y pasa el valor de 101 a ésta. Esta forma es usada rara vez porque la segunda opción, también llamada método corto, es más pequeña y precisa. En la forma corta, el argumento o argumentos deben seguir al nombre del objeto y estar entre parentesis. Por ejemplo, la siguiente instrucción realiza la misma operación que la anterior :

```
cola uno(101);
```

Esta forma corta es la que se utilizará en los ejemplos y aplicaciones siguientes, ya que es la más utilizada por la mayoría de los programadores de C++.

La forma general de pasar argumentos a las funciones constructoras es :

tipo_de_clase nom_var (lista_argum);

En la lista de argumentos se debe utilizar una coma para separar cada argumento que será pasado al constructor.

Con esto podemos tener una nueva versión del ejemplo de la cola y mostrar el uso de argumentos en funciones constructoras :

```
#include <iostream.h>

// Aquí se crea la clase cola
class cola {
    int q[100];
    int inicio, fin;
    int quien;  // Retiene el número ID de la cola
public :
    cola(int id);    //  Función constructora
    ~cola( );  //  Función destructora
    int poner(int i );
    int quitar( );
};

//  Esta es la función constructora
cola :: cola(int id)
{
      llena= vacia= 0;
      quien= id;
      cout << "La cola" << quien << "ha sido inicializada\n";
}

// Esta es la función destructora
cola :: ~cola
{
       cout << " La cola" << quien << "ha sido desactivada\n";
}
```

```
void cola :: poner(int i)   //Con esta función se ponen elementos al
final de la cola
{
      if(fin==100) {
        cout << "La cola está llena";
      return;
        }
        fin++;
        q[fin] = i;
}

int cola :: quitar( )  // Con esta función se elimina el primer elemento
de la cola
{
         if (inicio==fin)  {
           cout << "La cola está vacía";
           return 0;
         }
         inicio++;
         return q[inicio];
}

main ( )
{
      cola uno(1), dos(2);  // Crea dos objetos de tipo cola

       uno.poner(20);
       dos.poner(10);

       uno.poner(27);
       dos.poner(17);

       cout << uno.quitar( ) << " ";
       cout << uno.quitar( ) << " ";
       cout << dos.quitar( ) << " ";
       cout << dos.quitar( ) << "\n";

       return 0;
}
```

Este programa despliega en pantalla lo siguiente :

```
La cola 1 ha  sido inicializada
La cola 2 ha  sido inicializada
20 27 10 17
La cola  2 ha sido desactivada
La cola  1 ha sido desactivada
```

Como se puede ver en la función **main( )**, a la cola asociada con **uno** se le pasa el valor ID número 1 y a la cola asociada con **dos** se le pasa el valor número 2.

En éste ejemplo sólo se pasa un argumento cuando el objeto es creado, pero es posible pasar dos o más argumentos.

2.2 Funciones Amigas

Las funciones amigas son las funciones no miembros de una clase y que tienen acceso a las partes privadas de ésta, simplemente por declararlas amigas de la clase. Por ejemplo, la función **amigo( )** es declarada amiga de la clase **clas1** :

```
class clas1 {
.
.
.
public :
      friend void amigo( );
.
.
.
};
```

Como se puede ver la palabra reservada **friend** precede la declaración de la función, lo que es la forma general para las declaraciones. Las funciones amigas se incluyen en C++ por varias razones: a) ayudan a crear operadores sobrecargados más flexibles que se pueden utilizar cuando se sobrecarga el sistema de entrada/salida de C++, b) su uso permite a una función tener acceso a los miembros privados de dos o más clases; esta situación más comúnmente surge cuando dos diferentes clases tienen aspectos que se desean comparar; por ejemplo, se tienen dos clases, una tiene información de aviones y la otra acerca de automóviles. Asumiendo que ambas clases contienen información del número de pasajeros que lleva cada uno, se puede querer comparar la capacidad de un avión con un auto en específico. Para hacer esto de la manera más eficiente, se requiere que una función tenga acceso a los miembros privados de ambas clases.

Para mostrar cómo se usan las funciones amigas se muestra el siguiente programa que define dos clases llamadas **triángulo** y **rectángulo**. La clase **triángulo** contiene las dimensiones de la base, la altura y área de un triángulo, la clase **rectángulo** contiene el ancho, el largo y el área de un rectángulo. Ambas clases comparten la función **mismo_tam( )** para determinar si un triángulo y un rectángulo tienen la misma área. La declaración de las clases queda de la siguiente manera:

```
#include <iostream.h>
class triángulo;

class rectángulo {
      int area;  // área del rectángulo
      int ancho, largo;  // dimensiones del rectángulo
public:
      friend int mismo_tam(triángulo t, rectángulo r);
      void define(int l, int a);
      void imp_dim( );
};
```

```
class triángulo {
      int area;  // área del triángulo
      int base, altura;  // dimensiones del triángulo
public:
      friend int mismo_tam(triángulo t, rectángulo r);
      void define(int b, int a);
      void imp_dim( );
};
```

La función **mismo_tam( )**, la cual no es función miembro de ninguna pero es amiga de ambas clases, retorna verdadero en caso de que el objeto **triángulo** y el objeto **rectángulo**, los cuales forman sus argumentos, tengan la misma área o cero en cualquier otro caso. Esta es la función **mismo_tam( )** :

```
//Retorna verdadero si el triángulo y el rectángulo tienen la misma área
int mismo_tam(triángulo t, rectángulo r)
{
      if (t.area==r.area) return 1;
      return 0;
}
```

Como se puede ver, esta función necesita tener acceso a las partes privadas de ambas clases para desarrollar esta tarea eficientemente.

La razón para la declaración vacía de la clase **triángulo** al inicio de la decalaración de clases es que desde que la función **mismo_tam( )** en **rectángulo** hace referencia a **triángulo** antes de ser declarado, entonces **triángulo** debe ser declarado antes de hacerle referencia. Si no es así , el compilador no sabrá que es **triángulo** cuando lo encuentra en la declaración de **rectángulo**. En C++ , una referencia adelantada a una clase es simplemente la palabra reservada **class** seguida del nombre del tipo de clase. Normalmente la única ocasión en que se necesita hacer una referencia adelantada es cuando existe una función amiga involucrada.

Este es el programa completo, que muestra las clases **triángulo** y **rectángulo** y cómo una función amiga puede accesar las partes privadas de una clase :

```
#include <iostream.h>

class triángulo;

class rectángulo {
      int area;  // área del rectángulo
      int ancho, largo;  // dimensiones del rectángulo
public:
      friend int mismo_tam(triángulo t, rectángulo r);
      void define(int l, int a);
      void imp_dim( );
};

class triángulo {
```

```
        int area;  // área del triángulo
        int base, altura;  // dimensiones del triángulo
public:
        friend int mismo_tam(triángulo t, rectángulo r);
        void define(int b, int a);
        void imp_dim( );
};

//Retorna verdadero si el triángulo y el rectángulo tienen la misma área
int mismo_tam(triángulo t, rectángulo r)
{
        if (t.area==r.area) return 1;
        return 0;
}

void rectángulo :: define(int l, int a)
{
        longitud=l;
        ancho=a;
        area=l*a;
}

void rectángulo :: imp_dim()
{
        cout << "El rectángulo es de " << longitud << " por " << ancho;
        cout << '\n';
}

void triángulo :: define(int b, int a)
{
        base=b;
        altura=a;
        area=base*altura /2;
}

void triángulo :: imp_dim()
{
        cout << "El triángulo es de " << base << " por " << altura;
        cout << '\n';
}

main ( )
{
        rectángulo r1,r2;
        triángulo t1;

        r1.define(10,10);
        r2.define(5,7);

        t1.define(10,20);

        r1.imp_dim( );
        r2.imp_dim( );
        t1.imp_dim( );
```

```
    if(mismo_tam(t1,r1)) cout << "r1 y t1 tienen la misma área\n";
    return 0;
}
```

Es importante entender que una función amiga no es una función miembro; así, en el programa anterior, es completamente inválido intentar ejecutar la función **mismo_tam( )** utilizando la siguiente instrucción :

```
r1,mismo_tam(t1, r1);  // Esta instrucción está mal
```

Como una función amiga no es miembro, ésta no puede ser llamada usando un objeto y el operador punto. Hay que recordar que, una función amiga es simplemente una función "norma", es decir, una función que rige un comportamiento ya que tiene derecho a accesar los miembros privados de la clase o clases de las cuales es amiga. En general, las funciones amigas son pasadas a los objetos con los que pueden operar.

2.3 Funciones en línea para clases

Aunque no pertenece específicamente a la programación orientada a objetos, C++ tiene una muy importante característica, que no tiene C, llamada funciones en línea. Una función en línea es una función que es expandida cuando es invocada en vez de estar siendo llamada. Esto es similar a las funciones macro de C, pero más flexibles.

Existen dos formas de crear funciones en línea, la primera es la que utiliza el modificador (palabra reservada) **inline**. La forma general para declarar funciones en línea es:
inline declaración_de_función;

Por ejemplo, para crear una función en línea llamada funlin( ) sin parámetros y que retorna un valor entero, se debe declarar así :

```
inline funlin( )
{
.
.
.
}
```

Es decir, el modificador **inline** debe preceder todo el cuerpo de la función.

La importancia de las funciones en línea es la eficiencia. Cada vez que una función es llamada, una serie de instrucciones deben ser ejecutadas para tener la función lista para ser utilizada, incluyendo poner cualquier argumento o parámetro en la pila o *stack* y retornarlo de la función. En algunos casos, muchos ciclos del procesador son usados para completar estos procedimientos. De todas formas, con las funciones en línea nada semejante a esto existe y la velocidad de corrida o ejecución y procesamiento del programa se incrementa. Pero en caso de tener una función en línea muy grande, el

tamaño del programa ejecutable también se incrementará. Por este motivo, el mejor uso de funciones en línea es cuando son muy pequeñas y las funciones grandes deben ser utilizadas como funciones normales.

El siguente ejemplo muestra el modo de utilizar funciones en línea :

```
#include <iostream.h>

class clase1 {
      int algo;  // Variable privada por default
public:
      int toma_algo( );
      void pon_algo(int unidad);
};

inline int clase1 :: toma_algo( )
{
      return algo;
}

inline void clase1 :: pon_algo(int unidad)
{
      algo=unidad;
}

main ( )
{

      clase1 obj1;

      obj1.pon_algo(10);
      cout << obj1.toma_algo( );

      return 0;
}
```

Es importante aclarar que, técnicamente, el modificador **inline** no es un comando para que el procesador genere código en línea, sino que es más bien una petición o instrucción. Hay varios casos que impiden que el compilador pueda cumplir con esta instrucción. Las más comunes son que exista un ciclo (loop), una instrucción **switch** o **goto**; con esto el procesador no generará código en línea, ni en el caso que la función no retorne algún valor, si contiene la instrucción **return.** No se pueden generar funciones en línea recursivas ni con variables **static**. En cambio es correcto crear funciones constructoras y destructoras en línea.

La otra forma de crear funciones en línea en C++ es definir el código de una función dentro de la definición de la clase, es decir, cualquier función definida dentro de una clase es automáticamente definida como una función en línea. El ejemplo anterior puede ser reescrito de la siguiente manera :

```
#include <iostream.h>

class clase1 {
      int algo;  // Variable privada por default
public:
// Funciones en línea
      int toma_algo( ) {return algo;}
      void pon_algo(int unidad){algo=unidad;}
};

main ( )
{

      clase1 obj1;

      obj1.pon_algo(10);
      cout << obj1.toma_algo( );

      return 0;
}
```

El código de la función está definido en una sola línea y es el estilo más común que se utiliza en C++ para funciones muy pequeñas; pero no hay razón para no escribir el código completo de la siguiente manera :

```
#include <iostream.h>

class clase1 {
      int algo;  // Variable privada por default
public:
// Funciones en línea
      int toma_algo( )
  {
      return algo;
  }
      void pon_algo(int unidad)
  {
      algo=unidad;
  }
};
```

Como la forma de codificar las funciones en línea en el mismo renglón es la forma más común de escribir programas en C++, ésta será utilizada en el resto de los ejemplos y aplicaciones descritas aquí.

2.4 Herencia múltiple

Como se vió en el capítulo I, es posible para una clase heredar atributos a otra clase y convertirse así en clase base o clase padre. A las clases que se les heredan

atributos se les llama clases derivadas o clases hijos. Los términos aquí utilizados son clase base y clase derivada por ser los más comunes.

En C++ es posible clasificar a lo miembros de una clase en tres categorias; las dos primeras ya se han visto, público y privado; la tercera es protegido. Esta forma se logra mediante la palabra reservada **protected**. Como ya se sabe los miembros públicos se pueden accesar por cualquier función del programa. Los miembros privados sólo pueden ser accesados por funciones miembro o amigas. Los miembros protegidos también pueden ser accesados únicamente por las funciones miembro o amigas, pero la forma de heredar de los miembros protegidos es diferente a la de los privados.

Hay que recordar que cuando una clase hereda a otra clase, todos los miembros privados de la clase base son inaccesibles para la clase derivada. El siguiente ejemplo muestra el modo en que difieren las formas de heredar entre la parte privada y la protegida:

```
class uno {
      int a;
      int b;
public :
      void toma_ab( ) { return a*b; }
};

class dos : public uno {
public :
      dos(int i, int j) { a=i; b=j; }   // línea mal, parte inaccesible
      void show( ){cout << toma_ab( );} //Correcto, toma_ab() es público
};
```

Aquí, los elementos de **dos** pueden accesar la función pública **toma_ab** de **uno** pero no pueden accesar **a** y **b** por que son privadas de **uno.**

La forma para que los miembros de una clase derivada tengan acceso a los miembros privados de una clase base es haciendolos protegidos. Cuando se hace esto, los miembros declarados como protegidos siguen siendo privados para la clase, pero pueden ser accesados por una clase derivada. Por ejemplo :

```
class uno {
protected :
      int a; // Aún privada para uno
      int b; // pero puede ser usada por dos
public :
      void toma_ab( ) { return a*b; }
};


class dos : public uno {
public :
      dos(int i, int j) { a=i; b=j;}    // Ahora es correcto
```

```
        void show() {cout << toma_ab( );} //Correcto, toma_ab() es público
};
```

De esta forma **dos** tiene acceso a **a** y **b** pero aún son inaccesibles para el resto del programa. La clave es que cuando se declara algún elemento como protegido, se está restringiendo su acceso a sólo las funciones miembro de la clase, pero está cediendo este acceso al ser heredado. Cuando un elemento es privado, su acceso es denegado a clases derivadas.

La forma general de declarar una clase derivada es :

class nom_clase : acceso nom_clase {

.

.

.

};

Aquí, *acceso* puede ser público o privado: en caso de omisión será público si la clase base es una estructura, o privado si la clase base es una clase. Si *acceso* es público, todos los miembros públicos y protegidos de la clase base se volveran públicos y protegidos respectivamente de la clase derivada. Si *acceso* es privado, todos los miembros públicos y protegidos de la clase base serán privados en la clase derivada. Para entender esta ramificación se muestra el siguiente programa ejemplo :

```
#include <iostream.h>

class uno {
protected :
        int a; // Aún privada para uno
        int b; // pero puede ser usada por dos
public :
        void toma_ab( );
        void pon_ab( );
};

//En dos, a y b de uno se vuelven miembros protegidos
class dos : public uno {
        int llave;
public :
        int toma_llave { return llave; }
        void hacer_llave { llave=a*b; }
};

// tres tiene acceso a a y b de uno, pero no a llave de dos
// ya que es privado por default
class tres : public dos {
public :
        void fun( ) { a=2; b=3; }
};
```

```
void uno :: toma_ab( )
{
      cout << "Introduzca dos numeros : ";
      cin >> a >> b;
}

void pon_ab( )
{
      cout << a << " " << b <<"\n";
}

main ( )
{
      dos obt1;
      tres obj2;

      obj1.toma_ab( );
      obj1.pon_ab( );

      obj1.hacer_llave( );
      cout << obj1.toma_llave( )<< "\n";

      obj2.fun( );
      obj2.pon_ab( );

      return 0;
}
```

Como **dos** hereda de **uno** de modo público, los miembros protegidos de **uno** se vuelven miembros protegidos de **dos**. Esto significa que estos pueden ser heredados a **tres** y con esto el programa corre correctamente, pero si **dos** hereda de **uno** como privado, entonces a **tres** se le niega el acceso a **a** y **b** porque son miembros privados de **dos**. De esta forma las declaraciones del programa anterior quedan así :

```
// Este programa no corre porque tiene un error
#include <iostream.h>

class uno {
protected :
      int a;
      int b;
public :
      void toma_ab( );
      void pon_ab( );
};
// Ahora en dos, a y b de uno se vuelven miembros privados
class dos : public uno {
      int llave;
public :
      int toma_llave { return llave; }
      void hacer_llave { llave=a*b; }
};
```

```
// Como a y b son privados en dos por lo tanto
// no pueden ser heredados por tres y no son validos para su uso

class tres : public dos {
public :
      // Esta función no es valida ya que a y b son inaccesibles aquí
      void fun( ) { a=2; b=3; }
};

void uno :: toma_ab( )
{
      cout << "Introduzca dos numeros : ";
      cin >> a >> b;
}

void pon_ab( )
{
      cout << a << " " << b <<"\n";
}
```

Cuando **uno** es declarado privado en la declaración de **dos**, provoca que **a** y **b** se vuelvan privados en **dos**. Esto significa que no pueden ser heredados por **tres** y por consiguiente su función **fun( )** no puede tener acceso a **a** y **b**.

Acerca de las palabras reservadas **private**, **public** y **protected**, éstas pueden aparecer en cualquier orden y cualquier número de veces en la declaración de una clase. Por ejemplo esta declaración es perfectamente válida :

```
class clase1 {
protected :
      int a;
      int b;
public :
      void f1( );
      void f2( );
protected :
      int r;
public :
      int t;
};
```

De cualquier manera, se considera de mejor forma tenerlas declaradas una sola vez dentro de cada estructura o clase.

Es posible para una clase heredar dos o más atributos de otra clase. Para lograr esto se utiliza la siguiente forma general :

class *nom_clase_deriv : lista_clase_base*
{
.
.
.
};

Por ejemplo en este programa, la clase tres hereda de uno y dos :

```
#include <iostream.h>

class uno {
protected :
      int a;
public :
      void hacer_a( int i  )   { a = i ; }
};

class dos {
protected :
      int b;
public :
      void hacer_b( int i )   { b = i ; }
};

//  tres hereda de ambas clases, uno y dos
class tres : public uno, public dos {
public :
      int hacer_ab( )  { return a * b ; }
};

main ( )
{
      tres obj;

      obj.hacer_a(10);
      obj.hacer_b(20);

      cout << obj.hacer_ab( );

      return 0;
}
```

En este ejemplo, **tres** tiene acceso a las partes públicas y protegidas de ambas clases, **uno** y **dos**. Sin embargo este ejemplo no contiene constructores, y la situación se complica cuando la clase base contiene una función constructora. En este caso el ejemplo anterior quedará de la siguiente manera con constructores para las tres clases :

```
#include <iostream.h>

class uno {
protected :
      int a;
public :
      uno( ){ a = 10; cout << "Construyendo UNO \n" ; }
};

class dos {
protected :
      int b;
public :
      dos( ) { b = 20; "Construyendo DOS\n" ; }
};

//  tres hereda de ambas clases, uno y dos
class tres : public uno, public dos {
public :
      tres( ) { cout << "Construyendo TRES\n"; }
      int hacer_ab( ) { return a * b ; }
};

main ( )
{
      tres obj;

      cout << obj.hacer_ab( );
      return 0;
}
```

Cuando se corre este programa se despliega lo siguiente :

```
Construyendo UNO
Construyendo DOS
Construyendo TRES
200
```

Cabe notar que las clases base son construidas en el orden en el que aparecen en la declaración de **tres**. Este resultado es generalizable porque en C++ las funciones constructoras para cualquier clase base heredada serán llamadas en el orden en que aparecen. Una vez que la clase o clases base han sido inicializadas, los constructores de las clases derivadas son ejecutados.

Así como las clases base no toman argumentos, las clases derivadas no necesitan tener funciones constructoras siempre que una o más clases base lo hagan. De cualquier manera, cuando una clase base contiene una función constructora que toma uno o más argumentos, cualquier clase derivada debe contener también una función constructora. La razón para esto es que permite un medio para pasar argumentos a las funciones constructoras de la clase o clases base. Para pasar argumentos a las clases base, estos se

deben especificar después de la declaración de las funciones constructoras de las clases derivadas como en la siguiente forma general :

```
const_deriv(list_argum) :
      base1(list_argum, base2(list_argum),...,baseN(list_argum)
{
.
.
.
}
```

Aquí, de *base1* a *baseN* son nombres de clases base heredadas por la clase derivada. Es importante entender que la lista de argumentos asociados con las clases base puede consistir en constantes, variables globales, y/o parámetros de las funciones constructoras de las clases derivadas. Como la inicialización de un objeto ocurre en tiempo de corrida, se puede usar como argumento cualquier identificador que es definido en el cuerpo de la clase.

El siguiente ejemplo ilustra la forma en que la clases base pasan argumentos a las clases derivadas mediante la modificación del programa anterior :

```
#include <iostream.h>

class uno {
protected :
      int a;
public :
      uno(int i){ a = i; cout << "Construyendo UNO \n" ; }
};

class dos {
protected :
      int b;
public :
      dos(int i) { b = i; "Construyendo DOS\n" ; }
};

//  tres hereda de ambas clases, uno y dos
class tres : public uno, public dos {
public :
      tres(int x, int y);
      int hacer_ab( ) { return a * b ; }
};

//  Inicialización de uno y dos mediante el constructor de tres
tres :: tres(int x, int y) : uno(x), dos(y)
{
       { cout << "Construyendo TRES\n"; }
}
```

```
main ( )
{
      tres obj(10,20);

      cout << obj.hacer_ab( );
      return 0;
}
```

Hay que notar que el constructor **tres** no está usando parámetros directamente. De hecho, en este ejemplo, los parámetros son pasados mediante las funciones constructoras para **uno** y **dos**; pero, de cualquier manera, no es razón para que **tres** no pueda usar estos u otros argumentos.

Para hacer esto más claro en el ejemplo anterior, **tres( )** no fue definido en línea dentro de la clase **tres**, pero es posible hacerlo así. Es decir, es perfectamente válido declarar la clase **tres** de la siguiente forma :

```
//  tres hereda de ambas clases, uno y dos
class tres : public uno, public dos {
public :
      tres(int x, int y);
      {
            cout << "Construyendo TRES\n";
      }
      int hacer_ab( ) { return a * b ; }
};
```

Los objetos como parámetros de funciones

Un objeto puede ser pasado a una función del mismo modo que cualquier otro tipo de dato, es decir, se utiliza la notación convencional de C++ para llamado por valor del paso de parámetros. Esto significa que una copia del objeto, no el mismo objeto, es pasado a una función. De cualquier manera, salvo una importante excepción descrita posteriormente, cualquier cambio hecho al objeto dentro de la función no afecta al objeto en el resto del programa. El siguiente programa muestra este punto :

```
#include <iostream.h>

class OBJ {
      int i;
public :
      void ini_i ( int x ) { i = x; }
      void sale_i( )  { cout << i << " "; }
};

void fn( OBJ x) ;

main ( )
{
```

```
        OBJ ob;
        ob.ini_i(10);
        fn(o);
        ob.sale_i( );  //  La salida será 10, el valor de i no ha cambiado
        return 0;
}

void fn(OBJ x)
{
        x.sale_i( );  // La salida será 10
        x.ini_i(100);  // Aquí sólo se afecta a la copia local
        x.sale_i( );  //  La salida será 100
}
```

Cuando el objeto es pasado a una función, la copia actúa como un doble idéntico del objeto llamado; esto es que los datos en la copia son iguales a los del argumento. Esto significa que la copia de un objeto es hecha cuando ésta pasa a una función y que la función opera con una copia, no con el objeto original. De cualquier manera, el que una copia sea creada significa, en escencia, que otro objeto es creado.

Para el caso de las funciones constructoras y destructoras de algún objeto que pasa como argumento a una función, tenemos el ejemplo siguiente :

```
#include <iostream.h>

class mi_clase {
        int i;
public :
        mi_clase( int n );
        ~mi_clase( );
        void ini_i ( int n)   { i = n; }
        int toma_i ( ) { return i; }
};

mi_clase :: mi_clase(int n)
{
        i = n;
        cout << "Construyendo " << i << "\n";
}

mi_clase :: ~mi_clase( )
{
        cout << "Destruyendo " << i << "\n";
}

void fn( mi_clase obj);

main ( )
{
        mi_clase ob1(1);

        fn(ob1);
```

```
        cout << "Esta es i en la función main( ) :  ";
        cout << obj.toma_i( ) << "\n";

        return 0;
}
void fn( mi_clase obj)
{
        obj.ini_i(2);
        cout << "Esta es la i local : " << obj.toma_i( ) << "\n";
}
```

La salida que produce este programa es :

```
Construyendo 1
Esta es la i local :  2
Destruyendo 2
Esta es i en la función main( ) : 1
Destruyendo 1
```

En este ejemplo sólo se hace una llamada a la función constructora, pero se hacen dos llamados a la función destructora. La razón para que no se haga una llamada a la función constructora cuando una copia del objeto es hecha es que cuando se pasa un objeto a una función se quiere el estado actual del objeto y si el constructor es llamado cuando la copia es creada, la inicialización ocurrirá, posiblemente cambiando el valor del objeto; así que las funciones constructoras no pueden ser ejecutadas cuando la copia de un objeto es generada en el llamado a una función.

Aunque la función constructora no es llamada cuando un objeto es pasado a una función, es necesario hacer el llamado a la función destructora cuando la copia es destruida, es decir, cuando la función termina, ya que la copia también es destruida como cualquier variable local. Esto significa que la copia del objeto opera a lo largo de la ejecución de la función, sin embargo aún ocupa espacio en memoria por lo que es necesario utilizar destructores para liberar memoria ocupada por copias de objetos.

Cuando un objeto es pasado por valor a una función, en teoría, las modificaciones a este objeto dentro de la función sólo afectan a la copia del objeto que fue generada cuando fue hecha la llamada a la función. Pero existen algunos casos donde muy probablemente efectos laterales pueden afectar al objeto usado como argumento en un llamado por valor. Hay que recordar que una copia temporal del objeto es creada cuando un objeto es usado como argumento de una función y que el destructor de la copia es llamado cuando esta función termina, lo que permite muy raros tipos de *basura* (vestigios o sobras de otros programas que quedan en memoria) perturben los programas.
Esto se puede ilustrar con el siguiente programa :

```
//  PRECAUCION, este programa tiene un error no se intente correr

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

```
class mi_clase {
      int *p
public :
      // alojamiento dinámico de memoria para tener un valor entero
      mi_clase( int i ) {
            p=(int *) malloc(size of(int));
            if (p) *p = i;
      }
      ~mi_clase( ) { if (p) free(p); } //  Liberar la memoria
      void muestra( ) { cout << " Liberando p " << p << "\n"; }
};

void fn(mi_clase ob2);

main( )
{
      mi_clase ob1;

      fn(ob1);

      ob1.muestra( );    //   Error lógico, p ya ha sido liberado

      return 0;
}

void fn(mi_clase ob2)
{
      ob2.muestra( );    //  ob2 será destruido a la salida de la función

/*  Esto significa que la memoria apuntada por p será liberada a pesar
      de que ésta aún es necesitada por ob1 en la función main( )    */
}
```

Este ejemplo puede provocar el bloqueo de la computadora si se intenta correr, ya que se destruye el sistema de alojamiento dinámico.

La explicación es que cuando **ob1** en **main( )** es creado primero, la memoria es reservada utilizando **malloc( )**; un apuntador a esta memoria es puesto en **p**; y esta memoria es utilizada para alojar un valor entero especificado cuando **ob1** es creado; esto es perfectamente válido. Después, cuando **ob1** es pasado a **fn( )**, una copia de **ob1** es creada y copiada en el parámetro **ob2**, lo que aún no causa problema. Aún así, cuando **fn( )** termina, la copia alojada en **ob2**, es destruida y su destructor es llamado. Esto ocasiona que se libere **p**, pero esta liberación es de la misma memoria que la de la **p** original, en la función **main( )**, que es utilizada aún. Así, cuando el programa termina, **ob1** dentro de **main( )** es destruido, causando la liberación nuevamente de **p**, lo que ocasiona que el sistema de alojamiento de memoria falle.

La clave de este punto es que cuando se pasa un objeto como parámetro, se hace una copia de este objeto; cuando la función termina, esta copia es destruida. Se debe estar

seguro de que no haya efectos laterales causados por la destrucción de la copia. Una manera de evitar este problema es pasar apuntadores a estos tipos de objetos. En el siguiente capítulo se detallarán las formas de evitar este problema.

2.5 Arreglos y apuntadores en objetos

Arreglos de objetos

Es posible crear arreglos de objetos del mismo modo que se crean arreglos de cualquier otro tipo de datos. Por ejemplo, el siguiente programa declara una case llamada **pantalla** que maneja información acerca de varios tipos de monitores que pueden ser utilizados en una PC (computadora personal). Específicamente, contiene el número de colores que pueden ser desplegados y el tipo de adaptador de video. En la función **main( )**, un arreglo de tres objetos de tipo **pantalla** es creado, y los objetos que forman los elementos del arreglo son accesados mediante el método normal de índices.

```
//  Un ejemplo de arreglos de objetos

#include <iostream.h>

enum tipo_pant {mono, cga, ega, vga};

class pantalla {
      int colores;  //  número de colores
      enum tipo_pant tp;  //  tipo de pantalla
public :
      void ini_colores(int num) { colores = num; }
      int trae_colores( ) { return colores; }
      void ini_tipo(enum tipo_pant tipo)  { tp = tipo; }
      enum tipo_pant trae_tipo( ) { return tp; }
} ;

char nombres[4] [5] = {
      "mono",
      "cga",
      "ega",
      "vga"
} ;

main( )
{
      pantalla monitor [3];
      register int i;

      monitor[0].ini_tipo(mono);
      monitor[0].ini_colores(1);

      monitor[1].ini_tipo(cga);
```

```
        monitor[1].ini_colores(4);

        monitor[2].ini_tipo(vga);
        monitor[2].ini_colores(16);


        for(i=0; i<3; i++) {
                cout << nombres[monitor[i].trae_tipo( )] << " ";
                cout << "tiene " << monitor[i].trae_colores( );
                cout << "color(es)\n";
        }

        return 0;
}
```

Este programa despliega lo siguente en pantalla :

```
mono tiene 1 color(es)
cga tiene 4 color(es)
vga tiene 16 color(es)
```

Aunque no tiene que ver con los arreglos de objetos, cabe hacer notar que el arreglo de dos dimensiones de tipo caracter, llamado **nombres**, es usado para convertir entre un valor enumerado a su equivalente en cadena de caracteres. En todas los enumeraciones que no tienen inicializaciones explicitas, la primera constante tiene un valor de cero, la segunda 1, y así sucesivamente. De este modo, el valor retornado por **trae_tipo( )** puede ser usada para los índices del arreglo **nombres**, obligando a la correcta impresión del nombre.

Los arreglos multidimensionales de objetos utilizan el mismo modo de índices que los arreglos de cualquier otro tipo de datos.

Apuntadores a objetos

Como se sabe, en C, se puede accesar una estructura directamente o a través de apuntadores a esta estructura. Así, en C++, se puede hacer referencia a un objeto directamente (como en todos los ejemplos anteriores) o mediante el uso de apuntadores a este objeto. Como se podrá ver, los apuntadores a objetos son de las más importantes características en C++.

Para accesar un miembro de un objeto cuando se utiliza el mismo objeto, se debe utilizar el operador punto (.). Para accesar un miembro especifico de un objeto cuando se utiliza un apuntador al objeto, se debe usar el operador flecha (->). El uso del operador punto y del operador flecha para objetos es paralelo al del uso en las estructuras y uniones.

Para la declaración de un apuntador en objetos se utiliza la misma sintaxis que se usa en las declaraciones de otros tipos de datos. El siguiente programa crea una clase simple llamada **A_ejem** y define un objeto de esta clase llamado **obj** y un apuntador al objeto de tipo **a_ejem** llamado ap. Aquí se muestra entonces como accesar directa e indirectamente usando un apuntador.

```
//   Un simple ejemplo usando un apuntador a un objeto

#include <iostream.h>

class a_ejem {
      int num;
public:
      void ini_num(int val) { num = val; }
      void imprime( );
};

void a_ejem :: imp_num( )
{
      cout << num << "\n";
}

main( )
{
      a_ejem obj, *ap;  //  Declaración del apuntador al objeto

      obj.ini_num(1);  //  Acceso directo

      obj.imp_num( );

      ap=&obj;  //  Asignación a ap de la dirección de obj
      ap->imp_num( );  //  Acceso a obj mediante un apuntador

      return 0;
}
```

Hay que notar que la dirección de **obj** es obtenida usando el operador ampersan (&) del mismo modo que se obtiene la dirección de cualquier tipo de dato. Como se sabe, cuando un apuntador es incrementado o decrementado, es avanzado o retrocedido de forma que apunte al siguiente o anterior elemento de su tipo base. Lo mismo ocurre cuando un apuntador a un objeto es incrementado o decrementado : el siguiente o anterior objeto es apuntado. Para mostrar esto, el ejemplo anterior ha sido modificado para que **obj** sea un arreglo de dos elementos de tipo **a_ejem**. Es importante notar como **ap** es incrementado y decrementado para accesar los dos elementos del arreglo.

```
// Incremento de un apuntador a un objeto

#include <iostream.h>
```

```
class a_ejem {
      int num;
public:
      void ini_num(int val) { num = val; }
      void imprime( );
};

void a_ejem :: imp_num( )
{
      cout << num << "\n";
}

main( )
{
      a_ejem obj, *ap;  //  Declaración del apuntador al objeto

      obj.ini_num(1);   //  Acceso directo

      obj.imp_num( );

      ap=&obj;  //  Asignación a ap de la dirección de obj
      ap->imp_num( );   //  Impresión de obj[0] mediante un apuntador

      ap++;      // Avance al siguiente objeto
      ap->imp_num( );  //  Impresión de obj[1] mediante un apuntador

      ap--;  //  Retrocede al objeto anterior
      ap->imp_num( );   //  Acceso a obj[0] mediante un apuntador

      return 0;
}
```

La salida de este programa es 10, 20, 10.

CAPITULO III

FUNCIONES Y OPERADORES

Funciones y operadores sobrecargados son dos de las más importantes y versátiles características en C++. De hecho, éstas son usadas en casi todas las pequeñas aplicaciones o programas de C++. En el capítulo 1 se introdujo a las funciones sobrecargadas y en este capítulo se examinan más a fondo y se abarcan los operadores sobrecargados.

3.1 Funciones constructoras sobrecargadas

A pesar de que las funciones constructoras prestan un único servicio, no son muy diferentes a las demás funciones y también pueden ser sobrecargadas. Para sobrecargar una función constructora de una clase simplemente se declaran las diferentes formas que esta puede tomar y se definen las acciones relativas a estas formas. Por ejemplo, el siguiente programa declara una clase llamada **tiempo** que actúa como un contador descendente. Cuando un objeto de tipo **tiempo** es creado, éste adquiere un valor de tiempo inicial. Cuando la función **corre( )** es llamada, se hace una cuenta regresiva hasta cero y entonces suena una campana. En este ejemplo, el constructor ha sido sobrecargado para que el tiempo pueda ser especificado como tipo entero, una cadena de caracteres o como dos enteros correspondientes a los minutos y los segundos.

Este programa hace uso de la función **clock( )** de turbo C++, la cual retorna el número de tiempos, momentos o ticks del reloj del sistema desde que el programa comienza a correr. Dividiendo este número entre el macro **CLK_TCK** convierte el valor retornado a segundos. La función **clock( )** y la definición de **CLK_TCK** se encuentran en el archivo de cabecera TIME.H.

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

class tiempo {
      int segundos;
public :
   //   los segundos especificados como cadena de caracteres
      tiempo(char *t) { segundos = atoi( t ); }

   //   los segundos especificados como enteros
      tiempo(int t) { segundos = t; }

   //   el tiempo epecificado en minutos y segundos
      tiempo(int mins, int segs)  { segundos = mins * 60 + segs; }

void corre( );
};
```

```
void tiempo :: corre( )
{
      clock_t  t1, t2;

      t1 = t2 = clock( ) / CLK_TCK;
      while(segundos)  {
            if(t1 / CLK_TCK + 1 <= (t2 = clock( )) / CLK_TCK ) {
                  segundos--;
                  t1 = t2;
                  cout << '.';
            }
      }
      cout << "\a\n";      //  sonido
}

main ( )
{
      tiempo a(10), b("20"), c(1,10);

      a.corre( );  //  cuenta 10 segundos
      b.corre( );  //  cuenta 20 segundos
      c.corre( );  //  cuenta 1 minuto y 10 segundos

      return 0;
}
```

Como se puede ver, cuando **a, b** y **c** son creados dentro de **main( )**, se les da un valor inicial usando los tres diferentes métodos soportados por la función constructora sobrecargada. Cada inicio causa la apropiada tarea constructora a ser utilizada.

En el programa se pueden ver pequeños valores de inicialización porque no es difícil en la forma de representar el tiempo. Pero, si se está creando una librería de clases para cualquiera que sea su uso, se pueden sustituir los constructores por las formas de inicialización más comunes para hacerlas más flexibles al usuario.

3.2 Inicialización dinámica de constructores

Como simples variables, los objetos pueden ser inicializados dinamicamente cuando son creados. Esto significa que se pueden crear objetos exactamente del tipo que se desean mediante la información que se conocc a través del tiempo de corrida o al ejecutar el programa, ya sea un objeto local o global. Para mostrar la inicialización dinámica de objetos se utilizara el programa anterior del tiempo.

En el primer ejemplo del programa del tiempo, aparecen pequeños incrementos mediante la sobrecarga en el constructor **tiempo( )**. De cualquier manera, existen casos en los que la inicialización de un objeto debe hacerse en tiempo de corrida del programa, es decir en la ejecución de este. Esto puede significar una gran ventaja al permitir varios formatos para ser usados, ya que permite la flexibilidad de usar el constructor lo más

parecido posible al formato de los datos. Por ejemplo, en esta versión del programa del tiempo, dos objetos, **b** y **c**, son construidos en tiempo de corrida utilizando inicio dinámico. Un objeto, **a**, es construido usando una constante.

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

class tiempo {
      int segundos;
public :
   //   los segundos especificados como cadena de caracteres
      tiempo(char *t) { segundos = atoi( t ); }

   //   los segundos especificados como enteros
      tiempo(int t) { segundos = t; }

   //   el tiempo epecificado en minutos y segundos
      tiempo(int mins, int segs)  { segundos = mins * 60 + segs; }

void corre( );
};

void tiempo :: corre( )
{
      clock_t  t1, t2;

      t1 = t2 = clock( ) / CLK_TCK;
      while(segundos)  {
            if(t1 / CLK_TCK + 1 <= (t2 = clock( )) / CLK_TCK ) {
                  segundos--;
                  t1 = t2;
                  cout << '.';
            }
      }
      cout << "\a\n";      //  suena la campana
}

main ( )
{
      tiempo a(10);    //  Constructor usando una constante
      a.corre( );  //  cuenta 10 segundos

      cout << "Introduzca los segundos: ";
      char cdn[80];
      cin >> cdn;
      tiempo b(cdn);  //  Inicio dinámico en tiempo de corrida
      b.corre( );

      cout << "Introduzca los minutos y los segundos: ";
      int mins, segs;
      tiempo c(mins, segs);    // Inicio dinámico
      c.corre( );
```

```
    return 0;
}
```

Como se puede ver, el objeto **a** es construido con una constante de tipo entero, pero los objetos **b** y **c** son construidos con información enteramente introducida por el usuario. Para **b**, desde que el usuario introduce una cadena de caracteres, ésta obliga a la función **tiempo( )** a ser sobrecargada para aceptarla. De modo similar, el objeto **c** es construido con información introducida por el usuario. En este caso, como el tiempo es introducido en minutos y segundos, es lógico utilizar este formato para construir el objeto **c**. Como se permiten varias formas de inicialización, se debe tener cuidado en crear cualquier conversión innecesaria de una forma a otra cuando se inicializa un objeto. Con esto, es posible escoger el mejor método para crear un objeto de acuerdo al tipo de datos utilizados en el momento, en cualquier parte del programa. En conjunto, los constructores sobrecargados, la inicialización dinámica y la declaración de variables cercanas al lugar de su uso constituyen una de las herramientas más poderosas de programación.

Hay que tener en cuenta que la importancia de sobrecargar funciones constructoras es ayudar al manejo de grandes complicaciones mediante la construcción de objetos en la forma más natural relativa a su uso específico. Como existen tres formas comunes de pasar variables de tiempo a un objeto, la sobrecarga de la función **tiempo( )** acepta cualquiera de las tres. De cualquier manera, sobrecargar la función **tiempo( )** para aceptar horas o días, o hasta nanosegundos, probablemente no sea una buena idea. Con constructores sobrecargados, que rara vez son utilizados, se puede *ensuciar* el código y con esto tener influencias que podrían desestabilizar un programa. Lo importante de esto es decidir cuándo son una buena opción los constructores sobrecargados y cuándo pueden ser contraproducentes.

3.3 Sobrecarga en operadores

Otra característica de C++, la cual está relacionada con las funciones sobrecargadas, son los operadores sobrecargados. Salvo algunas excepciones, los operadores de C++ pueden dar significados diversos a su operar o funcionar con relación a clases específicas. Por ejemplo, una clase que define una lista ligada puede usar el operador + para agregar elementos a la lista. Cuando un operador está sobrecargado ninguno de sus atributos o funciones originales se pierden. Esto es que, simplemente una nueva operación, relativa a una clase específica, es definida. Así, la sobrecarga del operador + para manipular una lista ligada, no causa un cambio en su operación de adición con los números.

Para sobrecargar un operador, se debe definir qué operación llevará a cabo en relación con la clase a la que será aplicada. Esto se logra mediante la creación de una función operador, la cual define esta acción. La forma general de definir una función operador es :

tipo nom_clase :: operator# *(list_argum)*
{
// Operación definida para la clase
}

Aquí *tipo* es el tipo de valor que será retornado por la operación especificada; muchas veces este valor retornado es del mismo tipo de la clase, pero puede ser de cualquiera que se escoja. Los operadores sobrecargados retornan muchas veces valores del mismo tipo por lo que se justifica su sobrecarga ya que facilita su uso en expresiones complejas. El caracter # será remplazado por el operador a sobrecargar.

Las funciones operador deben ser miembros o amigas de la clase para la cual serán utilizadas. Aunque los métodos son muy similares, existen algunas diferencias entre la forma de sobrecargar funciones operador miembros y la forma de sobrecargar funciones operador amigas. A continuación se verá la forma para las funciones operador miembro.

El siguiente ejemplo crea una clase llamada **tres_d** que mantiene las coordenadas de un objeto en un espacio tridimensional; aquí se sobrecarga el operador + y el operador = para la clase **tres_d**.

```
#include <iostream.h>

class tres_d {
      int x, y, z;       //  Coordenadas tridimensionales
public:
      tres_d operator + (tres_d t );
      tres_d operator = (tres_d t );

      void muestra( );
      void asigna(int mx, int my, int mz);
} ;

//  Sobrecarga del operador +
tres_d tres_d :: operator+ (tres_d t)
{
      tres_d temp;

      temp.x = x + t.x;     //  aquí se hace una suma entera
      temp.y = y + t.y;     //  y el operador + retiene sus
      temp.z = z + t.z;     //  atributos originales de adición
      return temp;
}

//  Sobrecarga del operador =
tres_d tres_d :: operator= (tres_d t)
{
      x = t.x;     //  aquí se hace una asignación entera
      y = t.y;     //  y el operador = retiene sus
```

```
        z = t.z;     //  atributos originales de asignación
        return *this;
}

//  Mostrar las coordenadas x, y, z
void tres_d :: muestra( )
{
        cout << x << ", ";
        cout << y << ", ";
        cout << z << "\n ";
}

//  Asignar coordenadas
void tres_d :: asigna(int mx, int my, int mz)
{
        x = mx;
        y = my;
        z = mz;
}

main( )
{
        tres_d a, b, c;

        a.asigna(1,2,3);
        b.asigna(10,10,10);

        a.muestra( );
        b.muestra( );

        c = a + b;  //  Ahora se suman a y b juntos
        c.muestra( );

        c = a + b + c;    //    suma a, b y c juntos
        c.muestra( );

        c = a = b;  //  Múltiple asignación
        c.muestra( );
        b.muestra( );

        return 0;
}
```

Este programa produce la siguiente salida :

```
1, 2, 3
10, 10, 10
11, 12, 13
22, 24, 26
1, 2, 3
1, 2, 3
```

Como se prodrá ver al examinar este programa, las funciones operador tienen sólo un parámetro cada una, a pesar de que se están sobrecargando operadores binarios, ya que por ser funciones miembro simplemente es necesario pasar explícitamente un argumento a ésta, el otro argumento pasa implícitamente usando el apuntador **this**. Con esto, en la línea

```
temp.x = x + x.t;
```

la **x** se refiere a **this->x**, la cual es la **x** asociada con el objeto que puntualiza la llamada a la función operador. El objeto del lado derecho es pasado a la función.

En general cuando se usa una función miembro, no son necesarios los parámetros al sobrecargar un operador unitario, y sólo un parámetro es requerido para el caso de operadores binarios. (No es posible sobrecargar el operador ternario **?**). En cualquier caso, el objeto que causa el llamado de la función operador es implícitamente pasado vía el apuntador **this**.

Examinado este programa es posible entender mejor cómo trabajan los operadores sobrecargados, comenzando con la sobrecarga al operador +. Cuando dos objetos de tipo **tres_d** son operados por el operador +, las magnitudes de sus respectivas coordenadas son sumadas juntas, como se muestra en la función **operator + ( )** asociada con esta clase. Cabe hacer notar que de cualquier forma esta función no modifica el valor de cada operando. De hecho, un objeto de tipo **tres_d** es retornado por la función que contiene el resultado de la operación. Para entender por qué el signo + no cambia el contenido de cada objeto, se puede pensar en la operación aritmética 10+12. El resultado de esta operación es 22, pero 10 y 12 no son cambiados por el operador. Otro punto importante acerca de la forma en que el operador + es sobrecargado es que éste retorna un objeto de tipo **tres_d**, ya que la función puede retornar cualquier tipo válido en C++, de hecho que retorne un objeto de tipo **tres_d** permite que el operador + pueda ser usado en expresiones más complejas, como **a+b+c**.

Contrastando con el operador +, la asignación de un operador en realidad causa que uno de sus argumentos sea modificado. Esto es, después de todo, la esencia de la asignación. Como la función **operator = ( )** es llamada por el objeto que está en el lado izquierdo de la asignación, éste es el objeto que es modificado por la operación de asignación. De cualquier forma, siempre la operación de asignación debe retornar un valor porque, en C++ como en C, la operación de asignación produce el valor que está del lado derecho de la expresión. De esta forma, para permitir una instrucción como

```
a = b = c = d;
```

operator = ( ) debe retornar el objeto apuntado por **this**, el cual será el objeto que ocupe el lado izquierdo de la asignación, así es posible hacer una cadena de asignaciones.

La acción de un operador sobrecargado es aplicada a la clase para la cual es definido sin que necesite alguna relación con el uso que se le da por default. Por ejemplo, los operadores << y >> del modo que son utilizados con **cin** y **cout** no tienen nada en común con el uso que se les da al aplicarlos a valores numéricos. De cualquier manera, para fines de estructura y legibilidad del código, lo ideal es que un operador sobrecargado sea reflejo de la operación original para la que el operador es utilizado, esto cuando sea posible. Por ejemplo, el operador + relativo a **tres_d** es conceptualmente similar al operador + relativo a los enteros. La clave de esto es que mientras se crea un operador con el significado que se quiera, lo mejor es, por claridad, que este nuevo significado tenga alguna relación con el significado original.

Algunas restricciones se aplican a los operadores sobrecargados, primero, no se puede alterar el precedente de cualquier operador; segundo, no se puede alterar el número de operandos que el operador requiere originalmente, aún así es posible que el operador sobrecargado escoja o ignore un operando. Finalmente, excepto para el operador =, los operadores sobrecargados son heredados por cualquier clase derivada. Naturalmente, se pueden sobrecargar estos operador relativos a esta clase derivada, si es necesario, y cada clase puede definir su propio operador sobrecargado = si ésta lo necesita.

Los únicos operadores que no es posible sobrecargar son :

. :: .* ?

3.4 Funciones operador amigas

Es posible para una función operador ser amiga de una clase antes que ser miembro. Como se vió anteriormente en este capítulo, las funciones amigas no utilizan argumentos a través del apuntador **this**. De cualquier manera, cuando una función es utilizada para sobrecargar un operador, ambos operandos son pasados explícitamente cuando se sobrecargan operadores binarios y un sólo operando es pasado cuando se sobrecargan operadores unitarios. Los únicos operadores que no pueden utilizar funciones amigas son =. ( ), [ ] y ->. El resto puede utilizar cualquier miembro o función amiga para implementar la operación relativa a su clase. Como ejemplo se utilizara el programa anterior para demostrar la implementación de las funciones amigas en la sobrecarga del operador +.

```
//  Demostración de la función operador amiga
#include <iostream.h>

class tres_d {
      int x, y, z;       //  Coordenadas tridimensionales
public:
      // Ahora operator+ es amiga
      friend tres_d operator+ (tres_d op1, tres_d op2);

      tres_d operator= (tres_d op2);  // se emplea op2
```

```
        tres_d operator++ ( );        // prefijo
        tres_d operator++ (int i);  //  postfijo

        void muestra( );
        void asigna(int mx, int my, int mz);
} ;

/*  Sobrecarga del operador + como amigo, ambos operandos
    pasan explícitamente */
tres_d operator+ (tres_d op1, tres_d op2)
{
        tres_d temp;

        temp.x = op1.x + op2.x;     //  aquí se hace una suma entera
        temp.y = op1.y + op2.y;     //  y el operador + retiene sus
        temp.z = op1.z + op2.z;     //  atributos originales de adición
        return temp;
}

//  Sobrecarga del operador =
tres_d tres_d :: operator= (tres_d op2)
{
        x = op2.x;      //  aquí se hace una asignación entera
        y = op2.y;      //  y el operador = retiene sus
        z = op2.z;      //  atributos originales de asignación
        return *this;
}

// Sobrecarga del operador unitario ++ en su forma prefija
tres_d tres_d :: operator++( )
{
        x++;    //  ++ retiene sus atributos originales
        y++;    //  relativos a los enteros
        z++;
        return *this;   //  retorno de los valores incrementados
}

// Sobrecarga del operador unitario ++ en su forma postfija
tres_d tres_d :: operator++(int i)
{
        tres_d temp=*this;    //  Se hace una copia del operando

        x++;    //  ++ retiene sus atributos originales
        y++;    //  relativos a los enteros
        z++;
        return temp;   //  retorno del valor original
}

//  Mostrar las coordenadas x, y, z
void tres_d :: muestra( )
{
        cout << x << ", ";
        cout << y << ", ";
        cout << z << "\n ";
```

```
}

//  Asignar coordenadas
void tres_d :: asigna(int mx, int my, int mz)
{
      x = mx;
      y = my;
      z = mz;
}

main( )
{
      tres_d a, b, c;

      a.asigna(1,2,3);
      b.asigna(10,10,10);

      a.muestra( );
      b.muestra( );

      c = a + b;  //  Ahora se suman a y b juntos
      c.muestra( );

      c = a + b + c;   //   suma a, b y c juntos
      c.muestra( );

      c = a = b;  //  Múltiple asignación
      c.muestra( );
      b.muestra( );

      cout << " Incremento postfijo: ";
      a=c++;  //  Demuestra el incremento postfijo
      a.muestra( );
      c.muestra( );

      cout << " Incremento prefijo: ";
      a=c++;  //  Demuestra el incremento prefijo
      a.muestra( );
      c.muestra( );

      return 0;
}
```

Como se puede ver en el programa es posible sobrecargar el operador ++ y de igual forma el operador -- también puede ser sobrecargado. Como se mencionó anteriormente, cuando se sobrecarga un operador unitario, ningún objeto es pasado explícitamente a la función operador. Esto es que, una función operador unitaria no toma objetos como parámetros; en su lugar, la operación es realizada en el objeto que genera la llamada a la función a través del paso implícito del apuntador **this**. También cabe señalar que la forma de sobrecargar estos operadores varía un poco de acuerdo a la forma en que se desea se realice la operación, ya sea postfija o prefija. Las formas generales de sobrecarga de estos operadores son:

```
// Forma generica para el modo prefijo de ++ o --
tipo operator## ()
{
        // Cuerpo de la función
}

// Forma generica para el modo postfijo de ++ o --
tipo operator## (int i)
{
        // Cuerpo de la funcón
}
```

En general una función operador unitaria miembro no toma parámetros, pero al realizar incrementos o decrementos en la forma postfija se utiliza una variable de tipo entero como colchón.

En la instrucción **operator+( )**, ambos operandos son pasados a esta, el operando izquierdo es pasado en **op1** y el operando derecho en **op2**. En muchos casos, no es conveniente utilizar una función amiga en lugar de una función miembro cuando se sobrecarga un operador, pero ésta es una situación en la cual se debe utilizar una función amiga. Como se sabe, un apuntador al objeto que invoca una función operador miembro es pasado a través del apuntador **this**. En el caso de operadores binarios, el objeto del lado izquierdo invoca a la función. Esta es una ventaja, ya que asegura que el objeto del lado izquierdo defina la operación específica a realizar. Por ejemplo, asumiendo que existe un objeto llamado **Obj** que tiene definidas las funciones de asignación y adición, la siguiente instrucción es perfectamente valida:

```
Obj = Obj + 10;  //  Funciona correctamente
```

Como el objeto **Obj** está del lado izquierdo del operador +, éste es el que hace el llamado a su función operador sobrecargado, la cual, presumiblemente es capaz de sumar un entero a cualquier elemento de **Obj**. De esta forma, y en consecuencia, la siguiente instrucción es erronea:

```
Obj = 10 + Obj;  //  No funciona
```

Esta instrucción no funciona porque el objeto en el lado izquierdo es un entero, que es un tipo predefinido y para el cual no se involucran enteros y objetos de tipo **Obj** en la operación de adición.

El problema de los tipos predefinidos en el lado izquierdo puede ser eliminado si el operador + es sobrecargado utilizando dos funciones amigas. En este caso, la función operador pasa explícitamente ambos argumentos y es invocada como cualquier otra función sobrecargada, basada en el tipo de sus argumentos. Sobrecargando el operador + (o cualquier otro operador binario) utilizando una función amiga es posible para los tipos

predefinidos ser utilizados en lado izquierdo de la operación. El siguiente ejemplo demuestra esto:

```
#include <iostream.h>

class CL {
public:
      int count;
      CL operator=(int i);
      friend CL operator+ (CL ob, int i);
      friend CL operator+ (int i, CL ob);
};

CL CL :: operator=(int i)
{
      count = i;
      return *this;
}

//  Esta función realiza objeto + entero
CL operator+(CL ob, int i)
{
      CL temp;

      temp.count = ob.count + i;
      return temp;
}

//  Esta función realiza entero + objeto
CL operator+(int i, CL ob)
{
      CL temp;

      temp.count = ob.count + i;
      return temp;
}

main( )
{
      CL obj;

      obj = 10;
      cout << obj.count << " ";  //  Salida de 10

      obj = 10 + obj;  //   Se suma un objeto al entero
      cout << obj.count << " ";  //  Salida de 20

      obj = obj + 12;  //   Se suma un entero al objeto
      cout << obj.count;  //  Salida de 32

      return 0;
}
```

Como se puede ver, la función operator+( ) es sobrecargada dos veces para soportar las dos maneras en que se puede realizar la adición de un objeto de tipo CL y un entero.

3.5 Parámetros por referencia

Por omisión, C y C++ pasan argumentos a una función mediante el llamado por valor, es decir, se realiza una copia de los argumentos para ser utilizados por la función evitando así que cualquier modificación que sufran no altere los valores originales. En C (y opcionalmente en C++), cuando una función necesita poder hacer alguna alteración en los valores originales de las variables utilizadas como argumentos, los parámetros necesitan ser declarados explícitamente de tipo apuntador, y la función debe operar las variables del llamado con el operador de apuntadores *. Por ejemplo, el siguiente programa implementa una función llamada **canje( )**, la cual cambia dos argumentos de tipo entero:

```
#include <iostream.h>

void canje(int *a, int *b);

main( )
{
      int i, j;

      i = 10;
      j = 20;

      cout << i << " " << j << "\n";
      canje(&i, &j );  //   cambia ambos valores
      cout << i << " " << j << "\n";
      return 0;
}

//   Función al estilo C de apuntadores explicitos
void canje(int *a, int *b)
{
      int t;

      t = *a;
      *a = *b;
      *b = t;
}
```

Cuando la función **canje( )** es llamada, las variables usadas en el llamado deben estar precedidas por el operador **&** para producir un apuntador en cada argumento. Esta es la forma en que se genera la llamada por referencia en C; pero en C++, aún soportando este método, existe una forma más transparente de hacerlo, llamada parámetros por referencia o parámetros referenciados.

En C++ es posible decirle al compilador que genere de forma automática una llamada por referencia en vez de una llamada por valor para uno o más parámetros de una función en particular. Esto se hace precediendo el nombre de cada parámetro en la declaración de la función con el operador &. Por ejemplo, aquí aparece una función llamada **fun( )**, la cual toma un parámetro por referencia de tipo **int** o entero:

```
void fun(int &f)
{
      f = rand( );     //   Esto modifica el argumento llamado
}
```

Esta forma de declarar es utilizada también en las funciones prototipo. Hay que notar que la instrucción **f = rand( )** no usa el operador apuntador *. Cuando se hace una declaración de parámetros por referencia, el compilador de C++ reconoce automáticamente qué es un apuntador y hace la diferencia por el programador.

Una vez que el compilador ha reconocido esta declaración, éste pasa automáticamente la dirección de cualquier argumento que haga un llamado a la función **f( )**. Por ejemplo:

```
int val;

f(val);    //  Obtiene un valor aleatorio
printf("%d", val);
```

con este fragmento se pasa la dirección de la variable **val**, no su valor, y de este modo la función **f( )** puede modificar directamente el valor de **val**.

Para demostrar cómo funcionan los parámetros por referencia en una pequeña aplicación, se utilizara la función **canje( )** descrita anteriormente usando referencias y un modo distinto en su declaración y llamado.

```
#include <iostream.h>

void canje(int &a, int &b);

main( )
{
      int i, j;

      i = 10;
      j = 20;

      cout << i << " " << j << "\n";
      canje(i, j);  //   cambia ambos valores
      cout << i << " " << j << "\n";

      return 0;
}
```

```
//   aquí se define la función canje( ) usando llamado por referencia
void canje(int &a, int &b)
{
      int t;

      t = a;
      a = b;
      b = t;
}
```

Una segunda forma en que una referencia pude ser utilizada es cuando es retornado por la función. Para declarar una función de forma que retorne una dirección, simplemente se coloca el operador & a continuación del tipo de valor a retornar y antes del nombre de la función. Por ejemplo, la siguiente función prototipo **f( )** retorna una dirección de tipo **int** o entero :

```
int &f( );
```

Dentro de una función que retorna una referencia, en realidad, simplemente retorna un objeto o dicho de otra forma, el compilador automáticamente retorna la dirección del objeto. Por ejemplo, aquí **f( )** retorna la dirección o referencia de la variable **i**:

```
int &f( )
{
      int i;

      cin >> i;    //  Obtiene un valor para i

      return i;     //  automáticamente retorna la dirección de i
}
```

Es posible declarar una referencia que no es parámetro o valor de retorno de una función. Esto se hace mediante lo que se llama referencia independiente. Ya que este tema está algo fuera contexto y rara vez son una buena opción, simplemente se dará una breve definición. Una referencia independiente es, esencialmente, sólo otro nombre para una variable. Una referencia independiente se debe inicializar al ser declarada, lo que significa que se le asignará la dirección de una variable declarada anteriormente. A continuación se tiene un pequeño ejemplo de esto:

```
#include <iostream.h>

main( )
{
      int k;
      int &i = j;

      j = 10;
```

```
        cout << j << " " << i;     // desplegará 10 10

        k = 121;
        i = k; // Copia el valor de k en j
        cout << "\n " << j;     // desplegará 121
        return 0;
}
```

CAPITULO IV

FUNCIONES VIRTUALES Y POLIMORFISMO

El polimorfismo es un componente indispensable para la programación orientada a objetos y en C++ el término es utilizado para describir el proceso por el cual diferentes implementaciones de una función pueden ser accesadas mediante el mismo nombre. Esto significa que una clase general de operaciones puede ser accesada en la misma forma aunque la acción específica asociada con cada operación puede diferir.

En C++ el polimorfismo es soportado tanto en *tiempo de corrida* (tiempo de ejecución del programa) como en *tiempo de compilación* (tiempo en que el compilador checa errores de sintaxis, etc.). Operadores y funciones sobrecargadas son ejemplos de polimorfismo en *tiempo de compilación* y a pesar de su eficiencia no les es posible realizar todas las tareas requeridas por un verdadero lenguaje orientado a objetos. Por esto, C++ implementa el polimorfismo en *tiempo de corrida* a través de clases derivadas y fuciones virtuales, tópicos a describir en este capítulo. Para comenzar es necesario explicar el funcionamiento de los apuntadores a tipos derivados ya que son requeridos para soportar polimorfismo en tiempo de corrida.

Apuntadores a tipos derivados

Los apuntadores a tipos base y a tipos derivados están relacionados. Suponiendo que se tiene un tipo base llamado **clase_B** y un tipo llamado **clase_D** derivado de **clase_B**, en C++, cualquier declaración de apuntador a **clase_B** puede ser utilizado como apuntador en **clase_D**, por ejemplo:

```
clase_B *p;    //   Apuntador a un objeto de tipo clase_B
clase_B obj_B;      //   objeto de tipo clase_B
clase_D obj_D;      //   objeto de tipo clase_D
```

con esto las siguientes líneas son perfectamente válidas:

```
p = &obj_B;    //  p apunta a un objeto de tipo clase_B
p = &obj_D;   /* p apunta a un objeto de tipo clase_D,
                 el cual es un objeto derivado de clase_B */
```

Mediante el uso de **p**, todos los elementos de **obj_D** heredados de **obj_B** pueden ser accesados, pero los elementos específicos de **obj_D** no pueden ser referenciados por **p**.

Es importante entender que, en general, en C++ un apuntador de algún tipo no puede ser utilizado para apuntar a un objeto de otro tipo. El hecho de que un apuntador

base pueda hacer referencia a un objeto de un tipo derivado es una excepción a las reglas de compatibilidad de tipos en C++ .

El siguiente programa sirve de ejemplo concreto de un apuntador base accesando un objeto derivado; define una clase base llamada **clase_B** y una clase derivada llamada **clase**_D las cuales implementan un simple directorio telefónico.

```
//  Uso de apuntadores a objetos de una clase derivada

#include  <iostream.h>
#include  <string.h>

class clase_B {
      char nombre[80];
public:
      void pon_nombre(char *s)  { strcpy(nombre,s);  }
      void mustra_nom( ) { cout <<  nombre  << "  ";}
};

class clase_D : public clase_B {
      char num_tele[80];
public:
      void pon_num(char *num) {strcpy(num_tele,num); }
      void muestra_num( ) { cout << num_tele << "\n"; }
};

main( )
{
      clase_b obj_B, *p;
      clase_D obj_D, *dp;

      p = &obj_B;   //  dirección del objeto base
      p->pon_nombre("Miguel Hidalgo"); // acceso a clase_B via apuntador
      p = &obj_D;   //  acceso a clase_D via apuntador base
      p->pon_nombre("Vicente Guerrero");

   //  Se muestra cada nombre en su propio objeto
      obj_B.muestra_nom( );
      obj_D.muestra_nom( );

  // Es posible usar la función muestra_nom( ) mediante el apuntador p
      p = &obj_B;     // p apunta al objeto base
      p->muestra_nom( );
      p = &obj_D;     // p apunta al objeto derivado
      p->muestra_nom( );

      cout << "\n";

      /*  Ya que pon_num( ) y muestra_num( ) no son parte de la clase
          base, no son accesibles para el apuntador base p y deben ser
          accesados directamente o, como se muestra a continuación, a
          través de un apuntador de tipo derivado.
```

```
        */

        dp = &obj_D;
        do->pon_num("555-25-25");
        ->muestra_nom( );     // p o dp pueden ser utilizados en esta línea
        p->muestra_num( );
        return 0;
}
```

En este ejemplo, el apuntador **p** está definido como un apuntador a **clase_B**, pero le es posible apuntar a un objeto de la clase derivada **clase_D** por lo que puede ser utilizado para accesar los elementos de la clase derivada definidos por la clase base. Aún así, un apuntador base no puede accesar aquellos elementos específicos de la clase derivada sin el uso de un tipo *molde*, por ejemplo, la función **muestra_num( )** es accesada mediante el uso del apuntador **dp**, el cual es declarado dentro de la clase derivada.

Si se desea accesar elementos definidos por un tipo derivado usando un apuntador base, éste se debe moldear en un apuntador del tipo derivado, por ejemplo, la siguiente línea hace una apropiada llamada de **obj_D** a la función **muestra_num( )** :

```
( ( clase_D * ) p ) -> muestra_num( ) ;
```

El conjunto externo de paréntesis es necesario para asociar el molde con **p** y no el tipo que retorna **muestra_num( )**. Aunque técnicamente no está mal utilizar esta manera de moldear un apuntador, muy probablemente resulte confuso su implementación dentro del código, por lo que no es recomendable.

Otro punto importante es que mientras se pueden utilizar apuntadores base a cualquier objeto de tipo derivado, al revés no es así. Es decir, no es posible accesar objetos de tipo base con apuntadores de tipo derivado.

Finalmente, un apuntador es incrementado y decrementado relativamente a su tipo base. Aún así, cuando un apuntador de una clase base señala a una clase derivada, el incrementarlo o decrementarlo no hace que éste apunte al siguiente o anterior objeto de clase derivada, sino que apuntará a lo que considera el siguiente objeto de la clase base. Es por esto que no se debe considerar el incrementar o decrementar apuntadores cuando éstos señalan a un objeto de clase derivada.

El hecho de que un apuntador a un tipo base pueda ser utilizado para apuntar cualquier objeto derivado de esta clase es fundamental para C++, de hecho, es crucial (como se verá más adelante) para la implementación del polimorfismo en tiempo de corrida.

4.1 Funciones virtuales

Como se mencionó anteriormente en este capítulo, el polimorfismo en tiempo de corrida se lleva a cabo mediante el uso de tipos derivados y funciones virtuales. En pocas palabras una *función virtual* es una función que es declarada como virtual en una clase base y redefinida en una o más clases derivadas. Lo que hacen de especial las funciones virtuales es que cuando son accesadas usando un apuntador base a un objeto de clase derivada, C++ determina qué función llamar al momento de la ejecución (tiempo de corrida) basándose en el tipo de objeto al que apunta. Así, cuando diferentes objetos son apuntados, diferentes versiones de la función virtual son automáticamente ejecutadas. De este modo se realiza el polimorfismo en tiempo de corrida.

Una función virtual se declara dentro de la clase base, precedida por la palabra reservada **virtual** y cuando se redefine la función virtual dentro de las clases derivadas no es necesario que tengan la palabra **virtual** en su declaración (aunque el hacerlo no ocasiona un error).

Como primer ejemplo de funciones virtuales se tiene el siguiente programa:

```
//  Un pequeño ejemplo de funciones virtuales
#include <iostream.h>

class Base {
public:
      virtual void quien( )  {      //  Se especifica la función virtual
            cout << "Base\n" ;
      }
};

class primera_d : public Base {
public:
      void quien( )  {      //  define quien( ) relativa a primera_d
            cout << " Primera derivación \n";
      }
};

class segunda_d : public Base {
public:
      void quien( )  {      //  define quien( ) relativa a segunda_d
            cout << " Segunda derivación \n";
      }
};

main( )
{
      Base obj_base;
      Base *p;
      primera_d  primer_obj;
      segunda_d  segundo_obj;
```

```
    p = &obj_base;
    p->quien( );      //   Acceso a la función quien( ) base

    p = &primer_obj;
    p->quien( );      //   Acceso a la función quien( ) de primera_d

    p = &segundo_obj;
    p->quien( );      //   Acceso a la función quien( ) de segunda_d

    return 0;
}
```

Este programa despliega en pantalla

```
Base
Primera derivación
Segunda derivación
```

Como se puede ver, en la clase **Base**, la función **quien( )** es declarada como virtual. Esto significa que la función puede ser redefinida por una clase derivada. Dentro de las funciones **primera_d( )** y **segunda_d( )**, **quien( )** es redefinida en relación a cada clase. En **main( )**, cuatro variables son declaradas: **obj_base**, la cual es un objeto de tipo **Base**; **p**, la cual es un apuntador a objetos de tipo **Base**; **primer_obj** y **segundo_obj**, las cuales son objetos de una clase derivada. A continuación a **p** se le asigna la dirección de **obj_base** y la función **quien( )** es llamada. Ya que **quien( )** es declarada como función virtual, C++ determina, en tiempo de corrida, a qué versión de **quien( )** es referida por el tipo de objeto apuntado por **p**. En este caso, es un objeto de tipo **Base**, por lo que la versión de la función **quien( )** declarada en la clase **Base** será ejecutada. Después, a **p** se le asigna la dirección de **primer_obj** y en el segundo llamado a la función **quien( )** C++ vuelve a examinar qué tipo de objeto es apuntado por **p** para determinar qué versión de la función **quien( )** será invocada. Ya que **p** apunta a un objeto de tipo **primera_d**, su versión de **quien( )** es utilizada. Del mismo modo, cuando **p** apunta a **segundo_obj**, la de **quien( )** declarada dentro de **segunda_d** es ejecutada.

La clave del utilizar funciones virtuales para llevar a cabo el polimorfismo en tiempo de corrida es que se deben accesar estas funciones a través del uso de un apuntador declarado para la clase base. Así es posible llamar una función virtual explícitamente usando el nombre del objeto como se suele llamar a cualquier función miembro de la clase. Esta es la única forma de realizar polimorfismo en tiempo de corrida.

La redefinición de una función virtual en una clase derivada es, en algunas ocasiones, como una función sobrecargada; pero la razón para no definirlas de este modo es que se aplican ciertas restricciones. Primero, los prototipos para las funciones virtuales deben ser iguales. Como se sabe, cuando se sobrecarga una función normal el tipo de valor que puede retornar y el número y tipo de parámetros puede diferir. Pero cuando se sobrecarga una función virtual, sus elementos no pueden cambiarse. Si los prototipos de las funciones difieren, la función es simplemente considerada como sobrecargada, y su

naturaleza de función virtual se pierde. Otra restricción es que una función virtual debe ser función miembro de la clase para la cual está definida (no puede ser función amiga). Es posible crear una función virtual amiga de otra clase, así, funciones destructoras pueden ser virtuales, pero las constructoras no. Debido a las restricciones y diferencias entre sobrecargar funciones normales y sobrecargar funciones virtuales, el término de *sobremontadas* es usado para describir la redefinición de funciones virtuales.

Una vez declarada una función como virtual, ésta continúa siendo virtual no importando por cuántos niveles de clases derivadas tenga que atravesar. Por ejemplo, si **segunda_d** es derivada de **primera_d** en lugar de **Base**, como se muestra, **quien( )** sigue siendo virtual y la versión apropiada es correctamente seleccionada:

```
//  segunda_d derivada de primera_d y no de Base

class segunda_d : public primera_d {
public :
  void quien( ) {      // define la función quien( ) relativa a segunda_d
       cout << "Segunda derivación\n";
  }
};
```

Cuando una clase derivada no sobremonta una función virtual, la versión de una función en la clase base es utilizada. Por ejemplo, si se cambia la definición de la clase segunda_d en el programa anterior por la siguiente:

```
class segunda_d : public Base {
       // No se hace la definición de quien( ) relativa a segunda_d
};
```

El programa desplegará entonces:

```
Base
Primera derivación
Base
```

Cabe señalar que las características de la herencia son jerárquicas. Así, si en el ejemplo anterior se utiliza la clase **segunda_d** como derivación de **primera_d** en lugar de serlo de **Base**, al hacer referencia a un objeto de tipo **segunda_d**, la versión de la función **quien( )** definida en **primera_d** será llamada, ya que es la clase más cercana a **segunda_d**, y no la que pertenece a la clase **Base**.

Como se mencionó al principio de este capítulo, las funciones virtuales en combinación con tipos derivado permiten a C++ soportar polimorfismo en tiempo de corrida y la razón de la importancia del polimorfismo en la programación orientada a objetos es que proporciona a una clase generalizada el permitir a sus funciones ser comunes a todos los tipos derivados y limitar las implementaciones de estas funciones a ciertas o a todas las clases derivadas. Esta misma idea puede expresarse también de la siguiente forma: la clase base indica la forma general de la interface en la que cualquier

objeto derivado de esta clase puede ser utilizado, pero permite a la clase derivada definir por sí misma el método a usar. De aquí que al polimorfismo se le conozca también por "una interface, múltiples métodos ".

Parte del poder aplicar óptimamente el polimorfismo es entender qué clases base y clases derivadas forman una jerarquía que se mueve de mayor a menor generalización (de base a derivación). Por lo tanto, cuando es usada correctamente, la clase base provee todos los elementos que una clase derivada puede usar directamente más las funciones que pueda implementar por sí misma; es decir, ya que la forma de la interface es definida por la clase base, cualquier clase derivada puede compartir esta interface común. De aquí que cuando se hace un diseño y uso apropiado de las funciones virtuales, la clase base define una interface genérica que puede ser utilizada por todas las clases derivadas.

En este punto es necesario recalcar la importancia de una interface consistente de múltiples implementaciones para la POO, ya que es la principal herramienta en la manipulación de programas grandes y/o complejos. Por ejemplo, si se desarrolla correctamente un programa, entonces se sabe que todos los objetos que se deriven de alguna clase base son accesados del mismo modo en general, aunque las acciones específicas varien de una clase derivada a la siguiente. Esto significa que es necesario recordar solamente una interface en vez de muchas. Posteriormenete la separación de interfaces y su implementación permite la creación de librerías de clases, las cuales pueden ser utilizadas en una tercera aplicación. Si estas librerías son implementadas correctamente pueden generar una interface en común que puede ser utilizada para clases derivadas de necesidades específicas.

Como ejemplo de lo útil que puede resultar el uso del concepto "*una interface, múltiples métodos*" tenemos el siguiente programa, el cual crea una clase base llamada **figura**. Esta clase es usada para almacenar las dimensiones de varios objetos bi-dimensionales y los resultados del cálculo de sus áreas. La función **inic_dim( )** se declara como miembro ya que es igual para todas las clases derivadas. La función **muestra_area( )** es virtual porque el modo de calcular las áreas de cada figura varían. El programa utiliza la clase base figura para derivar a dos clases específicas llamadas **triángulo** y **cuadrado**.

```
#include <iostream.h>

class figura  {
protected:
      double x, y;
public:
      void inic_dim(double i, double j)  {
            x = i;
            y = j;
      }
      virtual void muestra_area( )   {
            cout << "Cálculo de área no definida.\n";
      }
```

```
};

class triángulo : public figura  {
public :
      void muestra_area( ) {
            cout << "Un triángulo con altura ";
            cout << x << " y  base " << y;
            cout << " tiene un área de " ;
            cout << x * 0.5 * y << "\n";
      }
};

class cuadrado : public figura  {
public :
      void muestra_area( ) {
            cout << "Un cuadrado con dimensiones de ";
            cout << x << " por " << y;
            cout << " tiene un área de " ;
            cout << x *  y << "\n";
      }
};

main ( )
{
      figura *pf;     //  Crea un apuntador de tipo base

      triángulo t1;
      cuadrado c1;      //   Crea objetos de tipo derivado

      pf = &t1;
      p->inic_dim(10.0 , 5.0);
      p->muestra_area( );

      pf = &c1;
      p->inic_dim(10.0 , 5.0);
      p->muestra_area( );

      return 0;
}
```

Como se puede ver al examinar el programa, la interface de ambas clases derivadas, **cuadrado y triángulo** es la misma a excepción del cálculo de sus áreas, que son provistos por cada objeto. Con la declaración de la clase **figura** es posible crear otra clase derivada llamada, por ejemplo, **circulo** para calcular el área de un círculo dado su radio. Lo que se necesita hacer para esto es crear un nuevo tipo derivado que haga dicho cálculo. La importancia de las funciones virtuales se basa en la facilidad con que se pueden derivar nuevos tipos que compartan interfaces con otros objetos. La función para el círculo puede quedar de la siguiente forma :

```
class circulo : public figura {
public :
      void muestra_area( ) {
```

```
        cout  <<  "Un círculo con radio ";
        cout  <<  x;
        cout  <<  " tiene un área de ";
        cout  <<  3.14159 * x * x;
    }
};
```

En este ejemplo, para el cálculo del círculo, se tiene que revisar la función **muestra_area( )**, ya que ésta usa únicamente el valor de x, el cual es asumido para retener el valor del radio (el área del círculo se calcula con la formula πr^2). El problema es que la función **inic_dim( )**, definida en la clase figura, asume que pasará dos valores, no el único valor (x) que necesita la clase **circulo**.

Existen dos maneras de resolver este problema: la primera, y menos recomendable, es simplemente hacer un llamado a **inic_dim( )** usando un valor como colchón en el segundo parámetro de algún objeto de la clase círculo. La desventaja aquí es que se debe recordar poner el valor de colchón, con lo que se rompe con la regla esencial del polimorfismo de "una interface, múltiples métodos". La mejor forma de resolverlo es dando al parámetro y un valor por omisión dentro de **inic_dim( )**; de esta manera cuando se llama a **inic_dim( )** para el cálculo de un círculo, sólo es necesario especificar el radio. En los llamados para triángulos y cuadrados se deben especificar ambos parámetros. La siguiente función **inic_dim( )**, mostrada a continuación, puede reemplazar a su homónima del programa anterior, para que se le pueda agregar el cálculo de círculos.

```
class figura {
protected:
    double x, y;
public:
    void inic_dim( double i, doublej=0) {        //   Asigna un valor
por omisión
        x = i;
        y = j;
    }
    virtual void muestra_area( )  {
        cout << " Calculo de área no definida ";
        cout << " para esta clase.\n";
    }
} ;
```

Lo importante de este ejemplo es dejar en claro que al definir una clase base ésta debe ser lo más flexible posible para no crear restricciones innecesarias.

4.2 Funciones virtuales puras y su relación con los tipos abstractos

Como ya se dijo, cuando una función virtual, que no se sobremonta en una clase derivada, es llamada por un objeto de la misma clase derivada, la versión definida en la clase base es la que se usa. Pero estas funciones virtuales, de las clases base, en muchas circunstancias no tienen definido un significado específico para ciertas acciones. Por ejemplo, en la clase base **figura** utilizada en el programa anterior para el cálculo de áreas, la definición de **muestra_area( )** es simplemente un *acarreador*; es decir, no realiza ninguna operación y sólo despliega el área de cualquier tipo de objeto.

Al crear librerías de clases se podrá ver que es muy común tener funciones virtuales que no tienen definidos algunos significados o acciones para ciertos objetos dentro de las clases base. Cuando ocurre esto, existen dos maneras de resolverlo, una medida es tener, como en el ejemplo de las áreas, un mensaje de error. Pero esta forma, aunque útil en ciertas ocasiones, no es lo más conveniente para todas las situaciones. Por ejemplo, puede haber funciones virtuales definidas por una clase derivada que únicamente tienen significado para ésta y sirven para cualquiera de sus casos. Considere la clase **triángulo**, ésta no tiene sentido si la función **muestra_area( )** no esta definida. La solución óptima para estos problemas son las funciones virtuales puras.

Una función virtual pura es una función declarada en una clase base que no tiene definiciones relativas a la clase. Lo que sucede al no tener definiciones relativas a la base es que cada clase derivada puede definir su propia versión, es decir no puede utilizar la versión definida en la base. Para declarar una función virtual pura se usa la siguiente sintaxis :

- **virtual** *tipo nom_func (lista_param)* = 0;

donde *tipo* es el tipo del valor a retornar por la función y *nom_func* es el nombre de la función. Por ejemplo, en la siguiente versión de figura, la función muestra_area( ) es una función virtual pura :

```
class figura {
protected:
      double x, y;
public:
      void inic_dim( double i, double j = 0) {
          x = i;
          y = j;
      }
      virtual void muastra_area( ) = 0     //  Función virtual pura
} ;
```

Mediante la declaración de una función virtual pura se forza a cualquier clase derivada a definir su propia implementación. En caso de que alguna clase falle al hacer su propia implementación, el compilador Turbo C++ reporta un error.

Si una clase tiene por lo menos una función virtual pura se le llama *abstracta*. Las clases abstractas tienen una importante característica: no se pueden declarar objetos de estas clases. En su lugar, una clase abstracta puede ser utilizada sólo como base que otras clases pueden heredar. La razón de que las clases abstractas no puedan tener objetos es que una o más de sus funciones no tienen definición. Lo práctico de las clases base abstractas es la declaración de apuntadores de su tipo para soportar polimorfismo en tiempo de corrida.

Ataduras (Binding)

Existen dos términos que son comúnmente usados en la programación orientada a objetos: ataduras tempranas (early binding) y ataduras tardías (late binding). En C++ estos términos se refieren a los eventos que ocurren en tiempo de compilación y los que ocurren en tiempo de ejecución, respectivamente.

En términos de orientación a objetos, atadura temprana significa que un objeto es atado, en tiempo de compilación, al llamado que realiza de una función. Estó es que, toda la información necesaria para determinar cuál función será llamada, es reconocida cuando el programa es compilado. Ejemplos de ataduras tempranas son las llamadas a funciones estandard, llamadas a funciones sobrecargadas y las llamadas a funciones operador sobrecargadas. La principal ventaja de las ataduras tempranas es su eficiencia, son más rapidas y muchas veces requieren menos memoria. Su desventaja es que carecen de flexibilidad.

Atadura tardía significa que un objeto es atado en tiempo de corrida al llamado de funciones que realiza. Esto es, se determina qué funciones son relativas a un objeto *en el camino* del tiempo de ejecución. Como se explicó anteriormente es posible realizar ataduras tardías en C++ mediante el uso de funciones virtuales y tipos derivados. La ventaja de las ataduras tardías es que proveen de gran flexibilidad permitiendo que los programas respondan a eventos que sólo se reconocen en tiempo de corrida. Es posible usar ataduras tardías para soportar una interface común mientras que varios objetos pueden utilizar esta interface para definir su propias implementaciones; posteriormente, también puede ayudar a crear librerías de clases, las que a su vez pueden ser reutilizadas y extendidas.

Es importante tener en mente que la desventaja de utilizar ataduras tardías es que se pierde un poco de velocidad de proceso del programa, sin embargo la flexibilidad que resulta del utilizarlas es mayor que la pequeña baja en rendimiento de ejecución. No existe una receta para saber cuándo usar una técnica o la otra, simplemente depende de la aplicación del programa y de un buen análisis de resultados. Actualmente muchos de los programas grandes hechos en C++ utilizan combinaciones de ambos tipos de ataduras para lograr los mejores resultados deseados.

4.3 Constructores y destructores en clases derivadas

Como los elementos para el polimorfismo en C++ dependen en gran medida de las clases derivadas, es apropiado tener un pequeño análisis de estos elementos. Una importante consideración relativa a las clases derivadas es el momento en que constructores y destructores son ejecutados. Tanto para las clases base como para las derivadas es posible crear funciones constructoras; pero cuando una clase base contiene un constructor, éste es ejecutado antes que el constructor de la clase derivada. La razón de esto es que la clase base no tiene conocimiento de la existencia de las clases derivadas, por lo que cualquier declaración que necesite llevar al cabo es independiente de éstas y posiblemente prerequisito para cualquier clase derivada. En pocas palabras, las funciones constructoras son llamadas y ejecutadas en el mismo orden en que se derivan las clases comenzando por la base.

Contrarias a las funciones constructoras, las funciones destructoras en las clases derivadas son ejecutadas antes que las funciones destructoras de las clases base y la razón para ello es que, ya que los destructores en las clases base implican la destrucción de clases derivadas, el destructor derivado debe ser ejecutado, lógicamente, antes de ser destruido. En otras palabras, los destructores o funciones destructoras son llamadas y ejecutadas en orden inverso a la derivación de sus clases.

4.4 Múltiples clases base

Como se explicó anteriormente, es posible para las clases derivadas, ser usadas ellas mismas como una clase base para la creación de otra clase derivada. Cuando esto pasa, los constructores son ejecutados en orden de la derivación, comenzando por la clase base, y los destructores son ejecutados en orden inverso a la derivación. El siguiente programa muestra cómo son ejecutados constructores y destructores en clases derivadas :

```
#include <iostream.h>

class base {
public:
      base( )    { cout << "\nBase creada\n"; }
      ~base( ) {cout << "\nBase destruida\n\n"; }
};

class clase_D1 : public base {
public :
      clase_D1( )     { cout  <<  "clase_D1 creada\n"; }
      ~clase_D1( )  { cout  <<  "clase_D1 destruida\n"; }
};

class clase_D2 : public base {
public :
      clase_D2( )     { cout  <<  "clase_D2 creada\n"; }
```

```
      ~clase_D2( )  { cout  <<  "clase_D2 destruida\n"; }
};

main ( )
{
   clase_D1  d1;
   clase_D2  d2;

   cout << "\n";

   return 0;
}
```

Este programa produce la siguiente salida :

```
Base creada
clase_D1 creada

Base creada
clase_D1 creada
clase_D2 creada

clase_D2 destruida
clase_D1 destruida
Base destruida

clase_D1 destruida
Base destruida
```

CAPITULO V

APLICACIONES Y ANALISIS

Este último capítulo muestra cómo usar el Turbo C++ para crear aplicaciones para Windows utilizando las librerías necesarias. Como se podrá ver, crear aplicaciones para Windows no es tan sencillo como hacer aplicaciones para DOS por ejemplo, pero tampoco lo es tan difícil como se cree; de hecho lo importante para el buen desarrollo de aplicaciones para Windows es seguir un conjunto de reglas bien definidas, de este modo no se tendrá ningún problema.

Turbo C++ for Windows tiene una librería de clases llamada *ObjectWindows* que simplifica bastante la programación en Windows, por esta razón todos los ejemplos contenidos en esta parte utilizan esta librería. Una de las ventajas de su uso, es que permite concretar en muchos detalles tediosos de la programación normal (como el manejo de muchos apuntadores) que se hace en Windows sin las herramientas de POO que provee Borland.

5.1 Aspectos fundamentales del ambiente operativo Windows

Una buena definición o explicación de lo que es Windows, depende de a quien esté dirigida. Para un usuario final, Windows es una interface con la cual es más fácil interactuar con las aplicaciones en comparación a sistemas de tipo texto como DOS. En cambio, para los programadores es un "sistema operativo" orientado a gráficos, multitareas y con una colección de más de seiscientas funciones o instrucciones API. Estas funciones soportan una muy específica filosofía en el diseño de aplicaciones. Desde el punto de vista de un programador, Windows es una caja gigante de herramientas que, cuando son utilizadas correctamente, permiten la creación de muchos programas de aplicación que comparten una interface en común.

El objetivo de Windows es permitir a un usuario que tiene los principios básicos del uso del sistema poder ejecutar casi cualquier aplicación sin una enseñanza o entrenamiento previos. En teoría, si se sabe correr un programa para Windows entonces se pueden correr todos. Por supuesto que las aplicaciones cambian en el fondo de sus tareas pero no en la forma. De hecho, mucho del código para Windows está diseñado para soportar la interface con el usuario.

Es importante señalar que no todos los programas que corren bajo Windows tienen el estilo o interface propia de este ambiente operativo, ésto se puede observar al correr los ejemplos de los capítulos anteriores, en los que no se tienen las ventajas de una interface gráfica, pero se ejecutan de acuerdo a lo requerido. Con tales ejemplos queda clara la importancia de utilizar la filosofía de programación para Windows, es decir, programar aplicaciones Windows con el estilo Windows.

Como ya se mencionó, Windows es orientado a gráficos, lo que significa que está provisto de una interface gráfica para el usuario o GUI (Graphical User Interface). A pesar de los muy diversos tipos gráficos y modos de video que existen para una PC, no es necesario preocuparse por sus diferencias, ya que Windows los controla casi siempre.

Un importante punto a considerar, es que a pesar de que Windows (versión 3.1) corre bajo el sistema operativo DOS (Disk Operating System), los programas para Windows generalmente no interactuan con DOS directamente, ya que Windows tiene su propio conjunto de servicios por lo que si una aplicación llama algún servicio de DOS ésta no correrá bajo Windows. Por otro lado las aplicaciones propias para Windows toman sus servicios del API o de la librería de clases ObjectWindows.

Con algunas excepciones, el principio de las interfaces para usuarios basadas en ventanas, tales como OS/2, X-Window o Motif de UNIX y Windows, es proveer o simular en la pantalla el equivalente al área de trabajo de un escritorio. En un escritorio se pueden encontrar varios papeles diferentes, unos encima de otros. El equivalente al escritorio en Windows es la pantalla y el quivalente de los papeles son las ventanas en la pantalla. Del mismo modo que se mueven e intercambian papeles en un escritorio, es posible hacerlo con las ventanas y así poder controlar las partes de la pantalla de la manera que se hace en un escritorio.

A diferencia de DOS, Windows permite el uso del *mouse* para lograr un control completo del sistema, seleccionar y manejar objetos de manera más sencilla. De hecho las interfaces gráficas, hechas a base de ventanas, están diseñadas para usarse con un mouse, y aún que se permite el uso del teclado, las aplicaciones que ignoran el uso del mouse violan uno de los principios básicos en la programación para Windows.

Un ícono es un pequeño símbolo que se usa para representar algún programa o aplicación que puede ser activado mediante el señalamiento del mouse y oprimiendo dos veces su botón derecho o, dicho de la forma más común, dando un "doble-click". El uso de íconos e imagenes gráficas (bit-maps) obedece al viejo refrán de que "Una imagen dice más que mil palabras" logrando con ésto una forma más sencilla de trabajo para el usuario.

En cualquier ventana standard, Windows provee ventanas de propósito especial, siendo las más comúnes los menús y cuadros de diálogo. Un menú es una ventana especial que contiene una lista de opciones para las cuales el usuario puede hacer una selección. El menú de selecciones se crea mediante el uso de simples funciones de Windows. Un cuadro de diálogo es una ventana especial que permite una mejor interacción usuario-aplicación. Por ejemplo al seleccionar *archivo,* en el menú principal del Turbo C++ para Windows, y la opción *abrir* genera una ventana para escoger el archivo deseado (figura 5.1).

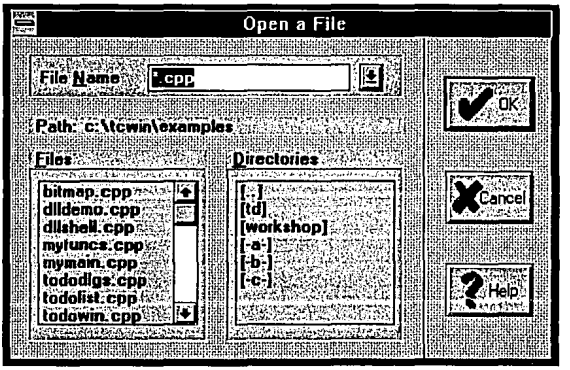


figura 5.1

Interacción Windows-Aplicaciones

Cuando se crea un programa para otros sistemas operativos (de tipo texto), es el propio programa el que debe comenzar la interacción con el sistema operativo. Por ejemplo, en un programa para DOS, el programa es el que envía mensajes de requerimientos como funciones de entrada/salida; dicho de otra forma los programas hechos de manera tradicional llaman al sistema operativo. En cambio, Windows funciona de forma opuesta, ya que Windows es el que hace el llamado al programa. Este procedimiento funciona de la siguiente manera : Un programa para Windows espera hasta que Windows envía un mensaje, éste mensaje es pasado al programa mediante una función especial de Windows, una vez que el programa recibe el mensaje espera a realizar acciones adecuadas para las que fue creado mediante el llamado a una o varias funciones del API,pero sigue siendo Windows el que comienza la actividad y el que determina la forma general de trabajo para todas sus aplicaciones.

Existen mucho tipos de mensaje que Windows puede enviar a un programa, por ejemplo, cada vez que se oprime un botón del mouse en una ventana que pertenece a la aplicación que se está usando, un mensaje de *mouse-click* es enviado, otro mensaje puede ser simplemente que se oprimió una tecla, etc. Lo importante de ésto es que, tan grande como sea la aplicación que se hace, éste recibirá un gran número de mensajes en forma aleatoria, por lo que pueden parecer programas de interrupción ya que no es posible saber cual será el siguiente mensaje.

Por la naturaleza basada en mensajes de Windows es especialmente compatible con C++ y la programación orientada a objetos. Como se podra ver, cada ventana que se crea se convierte en un objeto y todas las interacciones con este objeto ocurren a través de funciones miembro especiales para responder a los mensajes enviados por Windows.

Componentes de una ventana

Antes de comenzar con aspectos de programación para Windows es necesario tener en claro algunos términos. La figura 5.2 muestra una ventana standard con cada uno de sus componentes.

Todas las ventanas tienen un *marco* que define el límite de la ventana y es usado para mover o cambiar de tamaño la ventana; en la parte superior izquierda está el *cuadro del menú del sitema* en el que al hacer un click se despliega el menú del sistema. Arriba a la derecha están los *botones para maximizar y minimizar* la ventana. El *área del cliente* es la parte de la ventana donde los programas se desarrollan o llevan al cabo su tarea. Muchas ventanas tienen además *barras de movimiento horizontal y vertical* para poder desplegar texto o lo que esté activo dentro de la ventana.

5.2 Programación con ObjectWindows

Como se mencionó, el ambiente Windows es accesado a través de la interface de programación de aplicaciones API, que cuenta con aproximadamente 600 funciones que proveen de todos los servicios que el sistema Windows requiere. ObjectWindows es una compleja librería de clases jerárquicas que encapsulan porciones del API para simplificar la creación de programas para Windows, por lo que eventualmente ObjectWindows usara algunas funciones del API para realizar varias de sus operaciones. Existe un subsistema del API llamado GDI (Graphics Device Interface) o interface de control de gráficos que hace posible que Windows y sus aplicaciones corran bajo diferentes tipos de videos.

Antes de crear alguna aplicación para Windows es necesario manejar la siguiente información: ObjectWindows cuenta con dos clases fundamentales para la elaboración de programas para Windows, **TApplication** que indica que se creará una aplicación del ObjectWindows y **TWindow** que crea las ventanas. Su importancia es precisamente que todo programa para Windows necesita de una aplicación y una ventana.

Dos tipos de datos son comunes para toda la programación en Windows, el primero es **HANDLE** el cual es un valor de 16 bits que identifica o maneja una ventana, ya que todas las ventanas son referenciadas por sus identificadores o *manejadores*. El segundo tipo de dato es LPSTR, el cual es un apuntador largo o de tipo *far* para una cadena de caracteres.

Todos los programas para Windows, no sólo los creados por el ObjectWindows, comienzan su ejecución con una llamada a la función **WinMain( )** en vez de la función **main( )** tradicional para DOS. La función **WinMain( )** tiene algunas propiedades especiales que la diferencían de otras aplicaciones en un programa, primero, debe ser compilada como una función en Pascal de tipo largo o far. Como se sabe el lenguaje C soporta los llamados de funciones utilizando el formato de Pascal y viceversa, Pascal los de C. Turbo C++, y algunos otros compiladores de C para Windows definen el tipo **PASCAL** para acoplar las necesidades de la función **WinMain( )**, de este modo al declarar **WinMain( )** de tipo **PASCAL**, la función es automáticamente compilada usando los formatos adecuados.

Como ya se mencionó, todos los programas del ObjectWindows contienen partes en común que pueden ser de propósito general o reutilizable, por lo que es una buena opción tener un programa prototipo o *machote* que contenga lo que todo programa para Windows necesita. A diferencia de la programación para DOS, en el que un programa contiene por lo menos 5 líneas de código, los programas para Windows contienen un mínimo de 40 líneas de código, de hecho si no se usa ObjectWindows el mínimo aumenta considerablemente.

El siguiente programa forma un prototipo generalizado y contiene lo mínimo requerido para crear una aplicación para Windows que al compilar y ejecutar crea una aplicación y una ventana con la capacidad de minimizar, maximizar, mover, cambiar de tamaño y cerrar la aplicación.

```
//   Prototipo de una aplicación para Windows

#include <owl.h>

// Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HANDLE ThisInstance,
                  HANDLE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                              Args, VidMode) {};
            virtual void InitMainWindow();
};

//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {};
      //  Aquí se agregan detalles adicionales de la ventana
```

```
};


//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
      MainWindow = new TWindow(NULL, Name);
}

int PASCAL WinMain(HANDLE ThisInstance,HANDLE PrevInstance,
                   LPSTR Args, int VidMode)
{
      AppName App("Mi primera aplicación para Windows",ThisInstance,
                  PrevInstance, Args, VidMode);

      App.Run();  // Ejecuta una aplicación para Windows
      return App.Status;  // retorna el estatus de corrida
}
```

Analizando este programa, se puede observar que todos los programas del ObjectWindows deben incluir el archivo de cabecera OWL.H, el cual al compilarlo causa que automáticamente se incluyan varios archivos de cabecera más. Estos archivos de cabecera contienen los prototipos y definiciones usadas por la librería de ObjectWindows así como funciones del API y definiciones de tipos de variables.

El programa crea dos clases, una llamada **AppName,** la cual es simplemente un lugar para poner un nombre con el propósito de definir una aplicación y la facilidad de que casi todo el trabajo lo realiza la clase **TAplication**, heredada por **AppName**. El constructor **AppName** toma algunos parámetros, los cuales son simplemente pasados a través del constructor **TAplication**. El significado de estos parámetros es el siguiente: **App_Name** apunta a una cadena de caracteres que contiene el nombre de la aplicación. **ThisInstance** y **PrevInstance** son manejadores que hacen referencia a la aplicación actual y a cualquier instancia (o copia) previa de la aplicación. Esto es porque se pueden estar corriendo varios ejemplos del mismo programa al mismo tiempo (hay que recordar que Windows es un sistema multitareas). **PrevInstance** debe tener un valor de cero si no hay ninguna copia previa de la aplicación. El parámetro **Args** apunta a una cadena de caracteres en la cual está algún argumento enviado al ser ejecutada la aplicación. El parámetro **VidMode** contiene un valor que indica el modo en que será desplegada la ventana.

Dentro de **AppName**, la función virtual **InitMainWindow( )** es declarada o redefinida, ya que está declarada dentro de **TAplication**. Esto es para inicializar la ventana principal de la aplicación. Toda aplicación necesita de una ventana y con **InitMainWindow( )** se inicializa, al menos, la principal.

Después de que la aplicación ha sido definida, se debe construir la clase de la ventana principal, lo cual se lleva al cabo mediante la herencia de la clase base **TWindow**. En el programa prototipo o machote, la clase que crea la ventana principal es

llamada **AppWindow** y en su lugar se debe poner un nombre adecuado a la aplicación que se esté haciendo. La única función de **AppWindow** pasa los argumentos de sus constructor a través del constructor de **TWindow**. El parámetro **WType** describe el tipo general de la ventana que será creada y en caso de tener un valor nulo se crea una ventana estandar. El segundo parámetro apunta a una cadena de caracteres con el título de la ventana. La clase **TWindow** define muchas funciones virtuales que pueden ser redefinidas por **AppWindow**, lo que permite la personalización de la operación de una ventana.

Una vez que la aplicación y las clases de la ventana principal han sido definidas, se crea **InitMainWindow( )**. En el programa prototipo una nueva ventana es creada mediante el uso de **new**. Las variables **MainWindow** y **Name** son miembros de la clase **TAplication**. **MainWindows** apunta a la ventana principal que esta siendo creada y **Name** apunta a una cadena de caracteres que contiene el nombre de la aplicación.

La parte final del programa prototipo es la función **WinMain( )** con la que comienzan su ejecución todos los programas para Windows el cual controla la copia actual del programa y cualquier otra anterior, pasa un apuntador a cualquier argumento que se le haya enviado e inicia el modo del video. Para comenzar construye una aplicación, después corre su aplicación haciendo un llamado a la función miembro de **TAplication** llamada **Run( )**. Finalmente cuando la aplicación es terminada mediante el cierre de la ventana principal, **WinMain( )** retorna el estatus de terminación a Windows especificado en **Status**, variable miembro de **TAplication**.

5.3 Procesamiento de Mensajes

La comunicación entre objetos es a través de un sistema de mensajes, el cual forma parte de la programación orientada a objetos de Windows y en consecuencia de sus aplicaciones. El propósito de este sistema de comunicación en ObjectWindows es recibir y procesar los mensajes enviados por Windows. En un programa del ObjectWindows este sistema es creado automáticamente por **TAplication** y el programa prototipo simplemente usa el procesamiento de mensajes por default de **TAplication**. Por esta razón es responsabilidad de cada aplicación la forma de procesar estos mensajes y adecuarlos a sus necesidades.

Existen más de 100 mensajes de Windows, cada mensaje es representado por un único valor entero de 16 bits y en el archivo de cabecera WINDOW.H, incluido automáticamente por OWL.H, están los nombres estandar para estos mensajes.

Generalmente se utilizan los nombres macro, no los valores enteros, cuando se hace referencia a un mensaje. Otros dos valores acompañan a cada mensaje y contienen información relativa a su mensaje, uno es un valor entero y el otro un entero largo (**long int**) para los que Turbo C++ tiene los parámetros **WParam** y **LParam** respectivamente,

con los que se recibe estos valores. Típicamente manejan coordenadas del mouse o del cursor, el valor obtenido al oprimir una tecla, o valores relativos al sistema como el tamaño o tipo de letra, etc.

Para que un programa reciba y procese los mensajes, primero se deben crear funciones de procesamiento de mensajes, a las que más comúnmente se les llama funciones de respuesta de mensajes, las cuales deben ser miembro de la clase de la ventana a la que estén asignadas. Como ya se mencionó, la clase base **TWindow** provee un procesamiento por default para algunos mensajes ya que, en general, sólo es necesario crear funciones de respuesta para cada mensaje importante para el programa, de lo contrario los mensajes serán ignorados y en consecuencia inaccesibles para el programa. Debido a que existen más de 100 diferentes mensajes, es muy común que los programas ignoren muchos mensaje simplemente porque no tienen ninguna relación con las actividades a realizar. En pocas palabras, si se necesita procesar un mensaje, se debe crear una función de respuesta para éste.

Cuando se usa ObjectWindows, las funciones de respuesta de mensajes se implementan de un modo especial, sus llamados son mediante una tabla virtual de atención dinámica o DDVT (siglas de Dynamic Dispatch Virtual Table) para abreviar, en la cual cada función tiene un único índice asociado con ella. La dirección para cada función del DDVT es puesta en una tabla en una localización especificada por un índice, mediante el cual las funciones son llamadas y ejecutadas. Todas las funciones de respuesta tienen el índice del mensaje al que tienen que responder, así, cada vez que se recibe un mensaje se hace un llamado automático a la función adecuada que previamente se declaró. En pocas palabras de lo único que hay que preocuparse es de crear las funciones de respuesta de mensaje, ya que ObjectWindows enruta automáticamente los mensajes a la función apropiada mediante el DDVT y de las funciones **MessageLoop( )** y **PrecessAppMsg( )** definidas en **TAplication**. La función **MessageLoop( )** es la que recibe todos los mensajes y **PrecessAppMsg( )** es la que los despacha.

Para crear una función de respuesta se debe usar un constructor específico de Turbo C++ el cual tiene el índice asociado con la función. La forma general de declarar una función de respuesta es la siguiente:

virtual void *ejemplo*(RTMessage *Msg*) = [*índice*]

Aquí *ejemplo* es el nombre de la función de respuesta e *índice* es el mensaje al que se dará respuesta. Como ejemplo se tiene la declaración de la función de respuesta **WMChar( )** para despachar todos los mensajes de **WM_CHAR**:

```
virtual void WMChar (RTMessage Msg) = [WM_CHAR];
```

WM_CHAR es el nombre de un macro para un mensaje, el cual es enviado cada vez que se oprime una tecla. El nombre de la función, **WMChar( )** en este caso, es totalmente arbitrario ya que Turbo C++ para Windows llama a la función de respuesta

mediante el índice y no por su nombre, pero Borland (fabricante de Turbo C/C++, Turbo Pascal,etc.) recomienda utilizar la siguiente notación para nombrar funciones de respuesta : Al nombre del mensaje se le quita la línea de en medio y las letras siguientes a la primera letra de la derecha de la línea serán minúsculas. Por ejemplo, para el mensaje **WM_PAINT** el nombre de la función será **WMPaint( )**. De cualquier forma, el nombre de las funciones de respuesta puede ser cualquiera.

Aunque la declaración anterior para una función de respuesta es corrrecta, Turbo C++ para Windows incluye otra macro llamada **WM_FIRST** la cual tiene un valor de cero. Borland sugiere agregar este valor al mensaje para asegurar que el valor del mensaje no se modifique. Con esto una mejor forma de declarar funciones de respuesta es la siguiente:

```
virtual void WMChar(RTMessage Msg) = [WM_FIRST + WM_CHAR];
```

RTMessage es una referencia a la estructura **TMessage** que contiene, entre otras cosas, los dos valores adicionales que vienen con todo mensaje. Estos dos valores están organizados de la siguiente manera:

```
union {
      WORD WParam;
      struct tagWP {
            BYTE Lo;
            BYTE Hi;
      } WP;
};

union {
      DWORD LParam;
      struct tagLP {
            WORD Lo;
            WORD Hi;
      } LP;
};
```

Estas uniones permiten el acceso a los valores mediante enteros de 16 y 32 bits o a través de sus componentes de tipo byte y entero. **Result** es un entero de la estructura **TMessage** y es para asignar los valores de respuesta que ocasionalmente requieren los mensajes de Windows.

Ya se ha visto en teoría cómo responder a los mensajes de Windows ahora extendiendo el programa prototipo se responderá a un mensaje generado al oprimir una tecla. Para ésto simplemente se declara una función de respuesta como miembro de **AppWindow** de la siguiente manera: (Dentro de la función **WMChar( )** hay algo de código desconocido como **GetDC( )** y **ReleaseDC( )** los cuales se describiran más adelante.)

```
//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {};

      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMesagge Msg) = [M_FIRST + WM_CHAR];

      // Información adicional de la ventana
};


// Procesamiento al mensaje WM_CHAR
void AppWindow::WMChar(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      TextOut(DC, 1, 1, "   ", 3); // Borra un caracter previo
      ostr << (char) Msg.WParam << ends; // construye una cadena
                                         //  de caracteres
      TextOut(DC, 1, 1, s, strlen(s)); // Da salida a la cadena
      ReleaseDC( )(HWindow, DC);
}
```

Como se puede ver **WMChar( )** es función miembro de **AppWindow( )** el cual especifica el índice del mensaje a despachar. Este mensaje es enviado por Windows cada vez que se oprime una tecla en el teclado de la computadora. Cuando el mensaje es enviado **Msg.WParam** contiene el valor *ASCII* (American Standard Code for Information Interchange) de la tecla oprimida; **Msg.LP.Lo** contiene el número de veces que se ha repetido la misma tecla cuando ésta se sostiene oprimida. Los significados de los bits de **Msg.LP.Hi** se muestran a continuación:

15	1 si la tecla se oprimió, 0 si la tecla se ha liberado o despachado.
14	1 si la tecla se oprimió antes de ser enviado el mensaje, 0 si no se ha oprimido.
13	1 si se oprimió la tecla ALT, 0 si no se oprimió.
12	Usado por Windows
11	Usado por Windows
10	No se usa
9	No se usa
8	1 si oprimió una tecla de función, 0 de otra forma.
7-0	Código dependiente del fabricante de computadoras.

En este momento el único valor importante es **WParam**, ya que es el que tiene el valor de la tecla que se oprimió, pero es importante observar la gran cantidad de información que Windows maneja acerca del sistema en comparación con DOS. Claro que el programador puede utilizar sólo la que quiera o necesite.

El propósito del código dentro de **WMChar( )** es muy simple: desplegar el caracter de la tecla que se oprimió en la pantalla. Aparentemente es demasiado código para realizar ésta acción, pero hay dos razones para ésto, la primera es que, como Windows es Multitareas, no es posible desplegar un caracter si es que se está llevando al cabo otra tarea que tiene sobrepuesta una ventana del programa realizado. La segunda razón es al contrario, que otra parte del programa puede tener sobrepuesta una ventana ajena al mismo, para lo cual la aplicación debe pedir un permiso primero para llevar al cabo la tarea deseada; ésto es a través de **GetDC( )**, la cual obtiene el contexto de recursos en el que se está trabajando (el contexto de recursos tiene muchas variaciones por lo que será discutido a detalle más adelante). Una vez obtenido el contexto de recursos (o DC siglas de Device Context) es posible desplegar algo en la pantalla. Al final de la función, el contexto de recursos es liberado mediante **ReleaseDC( )**. El programa debe liberar el contexto de recursos cuando termina de utilizarlo o de lo contrario no podra ser utilizado por otra o la misma aplicación cuando sea llamado. **GetDC( )** y **ReleaseDC( )** son funciones del API y sus prototipos son los siguientes:

HDC GetDC(HWND *hWnd*);
int ReleaseDC(HWND *hWnd*, HDC *hDC*);

El tipo **HDC** es un manejador del contexto de recursos y el tipo **HWND** es un manejador de ventanas. Estos tipos son definidos en los archivos de cabecera que se incluyen automáticamente en OWL.H. **GetDC( )** retorna el contexto de recursos y **ReleaseDC( )** retorna verdadero si el contexto de recursos fue liberado y falso de otro modo.

La función para desplegar caracteres es **TextOut( )**, también del API, y su prototipo es:

BOOL TextOut(HDC *DC*, int *x*, int *y*, LPSTR *str*, int *conta*);

El tipo **BOOL** es un entero de 16 bits que retiene un valor verdadero o falso. La función **TextOut( )** despliega la cadena de caracteres apuntada por **str** en las coordenadas especificadas por *x* y *y*. La longitud de la cadena es especificada en *conta*. **TextOut( )** retorna verdadero si realizó su tarea satisfactoriamente o falso de otra forma.

En el programa cada caracter tecleado por el usuario es convertido en una cadena de longitud uno, mediante el uso del arreglo de entrada/salida (ver el apendice A), y desplegado con la función **TextOut( )** en las coordenadas 1,1. Las coordenadas de una ventana son siempre relativas al tamaño y localización de la misma ventana, no a la

pantalla. De cualquier manera, cuando entra un caracter, éste será desplegado en la esquina superior izquierda de la ventana.

El propósito para la primera llamada de **TextOut( )** dentro de **WMChar( )** es el de borrar cualquier caracter desplegado previamente en la misma coordenada. Como Windows es un sistema basado en gráficos, los caracteres pueden ser de diferentes tamaños y el sobreescribir un caracter en otro no asegura que el anterior sea borrado completamente. Por ejemplo, si se teclea una " w " seguida de una " i " parte de la " w " puede estar desplegada aún. También es importante aclarar que ninguna función de Windows permite que se despliegue fuera de los límites de las ventanas. Cualquier salida en la pantalla es automáticamente puesto dentro de la ventana apropiada, previniendo que se crucen las fronteras de las ventanas.

Windows tiene muy pocas funciones para desplegar texto o caracteres en el área de trabajo, ya que hace mucho más eficiente este trabajo mediante el uso de cuadros de diálogo y menús, como se verá más adelante.

El siguiente programa es la versión completa para responder a una tecla:

```
//   Prototipo de una aplicación para Windows que procesa el mensaje
//    WM_CHAR

#include <owl.h>
#include <string.h>
#include <strstream.h>

// Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HANDLE ThisInstance,
                  HANDLE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                                    Args, VidMode) {};
            virtual void InitMainWindow();
};

//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {};
      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMesagge Msg) = [M_FIRST + WM_CHAR];
      // Información adicional de la ventana
};

// Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
```

```
{
      MainWindow = new AppWindow(NULL, Name);
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20];

// Procesamiento al mensaje WM_CHAR
void AppWindow::WMChar(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      TextOut(DC, 1, 1, "    ", 3); // Borra un caracter previo
      ostr << (char) Msg.WParam << ends; // construye una cadena
                                         //  de caracteres
      TextOut(DC, 1, 1, s, strlen(s)); // Da salida a la cadena
      ReleaseDC( )(HWindow, DC);
}

int PASCAL WinMain(HANDLE ThisInstance,HANDLE PrevInstance,
                   LPSTR Args, int VidMode)
{
      AppName App("Respuesta al Mensaje WM_CHAR",ThisInstance,
                  PrevInstance, Args, VidMode);

      App.Run();  // Ejecuta una aplicación para Windows
      return App.Status;  // retorna el estatus de corrida
}
```

Para ejecutar la función **WMChar( )** es necesario obtener la prioridad para utilizar los recursos del sistema. Un recurso del sistema es el camino para las salidas de las aplicaciones para Windows, a través de un controlador apropiado de recursos, al área del cliente de la ventana. En otras palabras, el contexto de recursos define completamente el estado de los controladores de recursos.

Antes de que una aplicación pueda hacer alguna salida de información al área del cliente en la ventana, el contexto de recursos debe ser obtenido y hasta que esta operación no se complete no puede haber comunicación entre el programa y la ventana relativa a la salida que se desea realizar. Como se mencionó anteriormente, muchas cosas pueden ocurrir para que el estado de un recurso no sea obtenido, por ejemplo, que otra aplicación encime una ventana sobre la aplicación que solicita el contexto de recursos para hacer un desplegado, por lo que es necesario primero tener al frente esta aplicación. Así, el obtener el contexto de recursos se convierte en una regla para utilizar funciones de salida como **TextOut( )** entre otras.

Al correr el programa de respuesta de **WM_PAINT** el caracter en la ventana desaparece al cambiar el tamaño de la misma, moverla o al sobreponer otra, ésto es porque Windows no graba o mantiene una memoria de lo que contiene la ventana, para

hacer que no se borre el contenido de las ventanas es necesario enviar un mensaje a **WM_PAINT** con el que se redesplegara el contenido de la ventana después de cambiar su tamaño, moverla o sobreponerle otra ventana.

Mediante la función **WMPaint( )** ObjectWindows intercepta todos los mensajes de **WM_PAINT** , los procesa y toma las acciones adecuadas, como obtener el contexto de recursos. Con esto se hace innecesario responder directamente al mensaje de **WM_PAINT** y se puede rediseñar el programa anterior llamando a la función virtual **Paint( )** desde la clase **AppWindow**:

```
//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject Parent, LPSTR Title) :
            TWindow(Parent, Title) {};
      virtual void WMChar(RTMessage Msg)  =
            [WM_FIRST + WM_CHAR];
      virtual void Paint(HDC DC, PAINTSTRUCT &PI);
      // Información adicional de la ventana
};
```

Como se puede ver esta función tiene dos parámetros, el primero es un controlador de recursos que es obtenido automáticamente al interceptar el mensaje de **WM_PAINT**; el segundo es una referencia a una estructura (**struct**) que contiene información relativa al despliegue en la ventana. Esta estructura es la siguiente:

```
typedef struct tagPAINTSTRUCT {
      HDC hdc;                  // controlador de recursos
      BOOL fErase;              // Verdadero si el fondo a sido redesplegado
      RECT rcPaint;             // Coordenadas de la region de despliegue
      BOOL fRestore;            // reservado
      BOOL fIncUpdate;          // reservado
      BYTE rgbReserved[16]; // reservado
}PAINTSTRUCT;
```

El tipo **RECT** es una estructura que especifica las coordenadas del extremo derecho inferior y del izquierdo superior de la región rectangular en que se redesplegará el texto. Su prototipo es:

```
typedef struct tagRECT {
      int left, top;
      int right, bottom;
} RECT;
```

La función **Paint( )** puede ser redefinida para el programa anterior de la siguiente forma:

```
//Procesamiento del mensanje WM_PAINT
```

```
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
      TextOut(DC, 1, 1, s, strlen(s));  // vuelve a desplegar s
}
```

Como ObjectWindows realiza todos los servicios de arranque y terminación necesarios la función **Paint( )** lo único que hace es redesplegar el contenido de la ventana. El siguiente programa es el ejemplo completo para redesplegar el contenido de una ventana :

```
//   Prototipo de una aplicación para Windows que procesa el mensaje
//    WM_CHAR y WM_PAINT

#include <owl.h>
#include <string.h>
#include <strstream.h>

//  Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HANDLE ThisInstance,
                  HANDLE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                                    Args, VidMode) {};
            virtual void InitMainWindow();
};

//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject Parent, LPSTR Title) :
            TWindow(Parent, Title) {};
      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMessage Msg) = [WM_FIRST + WM_CHAR];
      virtual void Paint(HDC DC, PAINTSTRUCT &PI);
      // Información adicional de la ventana
};

//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
      MainWindow = new AppWindow(NULL, Name);
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20] = "Hello";

// Procesamiento al mensaje WM_CHAR
void AppWindow :: WMChar(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));
```

```
        DC = GetDC(HWindow);
        TextOut(DC, 1, 1, "   ", 3); // Borra un caracter previo
        ostr << (char) Msg.WParam << ends;  // construye una cadena
                                            // de caracteres
        TextOut(DC, 1, 1, s, strlen(s));   // Da salida a la cadena
        ReleaseDC(HWindow, DC);
}

//Procesamiento del mensanje WM_PAINT
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
        TextOut(DC, 1, 1, s, strlen(s));  // vuelve a desplegar s
}

// Parte principal de un programa para Windows
int PASCAL WinMain(HANDLE ThisInstance,HANDLE PrevInstance,
                        LPSTR Args, int VidMode)
{
        AppName App("Respuesta de Mensajes",ThisInstance,
                        PrevInstance, Args, VidMode);

        App.Run();  // Ejecuta una aplicación para Windows

        return App.Status;  // retorna el estatus de corrida
}
```

Aunque la función **Paint( )** del programa es muy simple, ejemplifica lo complejo que puede ser en un programa de verdadera aplicación, ya que normalmente las ventanas despliegan mucha más información. Como es responsabilidad del programa reestablecer una ventana si ésta es modificada en tamaño o lugar, es necesario siempre proveer mecanismos para hacerlo sobre todo en aplicaciones reales.

Mensajes del *mouse* o ratón

Ya que el mouse es básico dentro del ambiente operativo Windows, todas las aplicaciones deben responder a sus mensajes de los que, debido a su importancia, existen mucho tipos diferentes. Los más comúnes son **WM_LBUTTONDOWN** y **WM_RBUTTONDOWN**, los cuales son generados al oprimir el botón izquierdo y el botón derecho, respectivamente. Para responder a estos dos mensajes se tienen que agregar las funciones de respuesta adecuadas a la clase **AppWindow** y sus definiciones de la siguiente forma:

```
//   Prototipo de una aplicación para Windows que procesa los mensajes
//    WM_CHAR, WM_PAINT y del mouse

#include <owl.h>
#include <string.h>
#include <strstream.h>
```

```
//  Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HANDLE ThisInstance,
                  HANDLE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                                    Args, VidMode) {};
            virtual void InitMainWindow();
};

//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {};
      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMessage Msg) =
            [WM_FIRST + WM_CHAR];
      virtual void Paint(HDC DC, PAINTSTRUCT &PI);

      virtual void WMLButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_LBUTTONDOWN];            // respuesta al
botón izquierdo
      virtual void WMRButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_RBUTTONDOWN];            // respuesta al
botón derecho

      // Información adicional de la ventana
};

//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
      MainWindow = new AppWindow(NULL, Name);
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20] = "Hello";

// Procesamiento al mensaje WM_CHAR
void AppWindow :: WMChar(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      TextOut(DC, 1, 1, "   ", 3);       // Borra un caracter previo
      ostr << (char) Msg.WParam << ends; // construye una cadena
                                         // de caracteres
      TextOut(DC, 1, 1, s, strlen(s));    // Da salida a la cadena
      ReleaseDC(HWindow, DC);
}
```

```
//Procesamineto del mesanje WM_PAINT
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
      TextOut(DC, 1, 1, s, strlen(s));  // despliega s
}

// Procesamiento del botón izquierdo del mouse
void AppWindow :: WMLButtonDown(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      ostr << "Botón Izquierdo" << ends;
      TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
      ReleaseDC(HWindow, DC);
}

// Procesamiento del botón derecho del mouse
void AppWindow :: WMRButtonDown(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      ostr << "Botón Derecho" << ends;
      TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
      ReleaseDC(HWindow, DC);
}

// Parte principal de un programa para Windows
int PASCAL WinMain(HANDLE ThisInstance,HANDLE PrevInstance,
                        LPSTR Args, int VidMode)
{
      AppName App("Respuesta de Mensajes",ThisInstance,
                     PrevInstance, Args, VidMode);

      App.Run();  // Ejecuta una aplicación para Windows

      return App.Status;  // retorna el estatus de corrida
}
```

Cuando se oprime alguno de los botones, las coordenadas X,Y del mouse son especificadas en **Msg.LP.Lo** y **Msg.LP.Hi,** respectivamente. Las funciones de respuesta para el mouse utilizan estas coordenadas para desplegar la cadena de caracteres correspondiente al botón que se oprimió exactamente en el punto en que se hizo. (figura 5.3)

Cada vez que se generan los mensajes al oprimir los botones del mouse, también se genera mucha más información mediante el parámetro **WParam** que puede contener las combinaciones de los siguientes valores:

MK_CONTROL // Oprimir la tecla CTRL y algún botón del mouse
MK_SHIFT // Oprimir la tecla SHIFT y algún botón del mouse
MK_LBUTTON // Oprimir el botón derecho cuando está oprimido el izquierdo
MK_RBUTTON // Oprimir el botón izquierdo cuando está oprimido el derecho

Para ver como puede ser usada esta información extra se pueden sustituir las funciones de respuesta para el mouse del programa anterior por las siguientes funciones:

```
// Procesamiento del botón izquierdo del mouse
void AppWindow :: WMLButtonDown(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      ostr << "Botón Izquierdo" << ends;
      TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
      ostr.seekp(0, ios :: beg);
      if(Msg.WParam & MK_CONTROL) {
         ostr << "Botón Izquierdo + tecla CTRL" << ends;
         TextOut (DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
         ostr.seekp(0, ios :: beg);
      }
      if(Msg.WParam & MK_SHIFT) {
         ostr << "Botón Izquierdo + tecla SHIFT" << ends;
         TextOut (DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
         ostr.seekp(0, ios :: beg);
      }
      ReleaseDC(HWindow, DC);
}

// Procesamiento del botón derecho del mouse
void AppWindow :: WMRButtonDown(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      DC = GetDC(HWindow);
      ostr << "Botón Derecho" << ends;
      TextOut(DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
      ostr.seekp(0, ios :: beg);
      if(Msg.WParam & MK_CONTROL) {
         ostr << "Botón Derecho + tecla CTRL" << ends;
         TextOut (DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
         ostr.seekp(0, ios :: beg);
      }
      if(Msg.WParam & MK_SHIFT) {
         ostr << "Botón Derecho + tecla SHIFT" << ends;
         TextOut (DC, Msg.LP.Lo, Msg.LP.Hi, s, strlen(s));
         ostr.seekp(0, ios :: beg);
      }
```

```
        ReleaseDC(HWindow, DC);
}
```

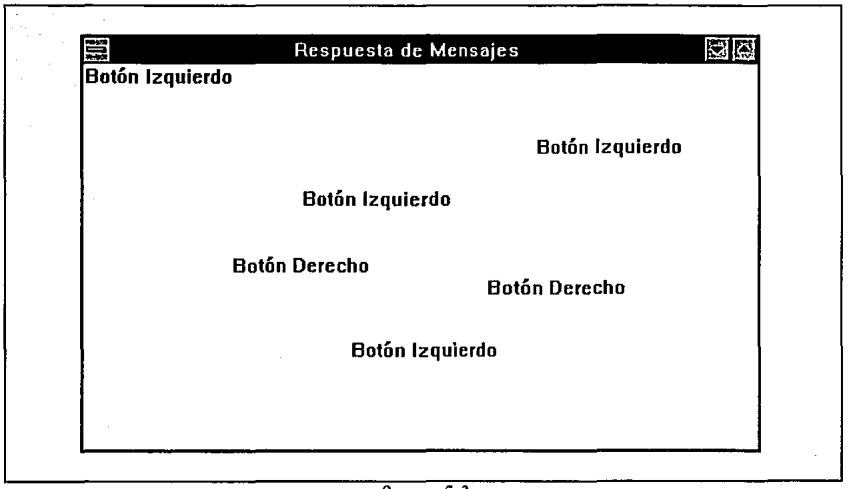


figura 5.3

Como generar un mensaje WM_PAINT

La importancia de generar mensajes de tipo WM_PAINT es el poder redesplegar la información contenida en una ventana que, como se mencionó anteriormente, se pierde al modificar la posición o el tamaño de la ventana. Debido a que Windows es un ambiente operativo multitarea lo mejor es que los programas le regresen el control lo más pronto posible, de este modo Windows decide cuando es el mejor momento para dar la salida de los programas enviando el mensaje **WM_PAINT**. Al utilizar este sistema, el programa simplemente retiene las salidas hasta que recibe el mensaje y posteriormente actualiza la ventana.

En el programa-ejemplo anterior el mensaje **WM_PAINT** se recibe únicamente cuando la ventana es modificada ya sea en tamaño o en posición, y si todas las salidas son retenidas hasta que se recibe el mensaje, entonces para realizar operaciones de entrada/salida en forma interactiva, debe existir alguna forma de decirle a Windows que es necesario que envie el mensaje **WM_PAINT** a la ventana cada vez que ésta tenga pendiente una salida. Así, cuando un programa tiene una salida en espera, envia una petición de que sea mandado el mensaje **WM_PAINT** cuando Windows esté listo para hacerlo.

Para que Windows mande el mensaje **WM_PAINT**, el programa debe llamar a la función del API **InvalidateRect( )**, de la cual su prototipo es:

void InvalidateRect(HWND *hWnd*, LPRECT *lpRect*, BOOL *bErase*);

Aquí, *hWnd* es el controlador de la ventana que se quiere enviar al mensaje **WM_PAINT**. El tipo **LPRECT** es un apuntador a la estructura **RECT**, la cual especifica las coordenadas en las cuales se debe hacer el redesplegado dentro de la ventana; si este valor es nulo, se especifica la ventana completa. Si el valor de *bErase* es verdadero o diferente de 0, el fondo de la ventana será borrado y no sufre cambios si es igual a cero.

Cuando la función **InvalidateRect( )** es llamada, se le dice a Windows que la ventana es invalida y que debe ser redesplegada, lo que ocasiona que Windows envie el mensaje **WM_PAINT** a la ventana.

Para ejemplificar mejor ésto, se le hacen algunas modificaciones al programa anterior para que utilice la función **InvalidateRect( )**:

```
/*   Prototipo de una aplicación para Windows que procesa los mensajes
     WM_CHAR, WM_PAINT y del mouse. Esta versión enruta todas las
     salidas a través de la función Paint()mediante la generación de un
     mensaje WM_CHAR usando InvalidateRect(). */

#include <owl.h>
#include <string.h>
#include <strstream.h>

//  Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HANDLE ThisInstance,
                  HANDLE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                              Args, VidMode) {};
            virtual void InitMainWindow();
};

//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {};
      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMessage Msg) =
            [WM_FIRST + WM_CHAR];
      virtual void Paint(HDC DC, PAINTSTRUCT &PI);

      virtual void WMLButtonDown(RTMessage Msg) =
```

```
            [WM_FIRST + WM_LBUTTONDOWN]; // respuesta al botón izquierdo
        virtual void WMRButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_RBUTTONDOWN];  // respuesta al botón derecho

        // Información adicional de la ventana
};

//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
        MainWindow = new AppWindow(NULL, Name);
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20] = "Hello";
int x,y;

// Procesamiento al mensaje WM_CHAR
void AppWindow :: WMChar(RTMessage Msg)
{
        HDC DC;
        ostrstream ostr(s, sizeof(s));

        ostr << (char) Msg.WParam << ends; // construye una cadena
        x = 1; y = 1;
        InvalidateRect(HWindow, NULL, 1); // actualiza la ventana
}

//Procesamineto del mesanje WM_PAINT
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
        TextOut(DC, x, y, s, strlen(s));  // despliega s
}

// Procesamiento del botón izquierdo del mouse
void AppWindow :: WMLButtonDown(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "Botón Izquierdo" << ends;
        x = Msg.LP.Lo;
        y = Msg.LP.Hi;
        InvalidateRect(HWindow, NULL, 1); // redespliega
}

// Procesamiento del botón derecho del mouse
void AppWindow :: WMRButtonDown(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "Botón Derecho" << ends;
        x = Msg.LP.Lo;
        y = Msg.LP.Hi;
        InvalidateRect(HWindow, NULL, 1); // redespliega
```

```
}

// Parte principal de un programa para Windows
int PASCAL WinMain(HANDLE ThisInstance,HANDLE PrevInstance,
                   LPSTR Args, int VidMode)
{
      AppName App("Respuesta de Mensajes",ThisInstance,
                  PrevInstance, Args, VidMode);

      App.Run();  // Ejecuta una aplicación para Windows

      return App.Status;  // retorna el estatus de corrida
}
```

Hay que notar que el programa tiene dos nuevas variables llamadas **x** y **y** para tener las coordenadas en las cuales se desplegará el texto al recibir el mensaje **WM_PAINT**. También se puede ver que mediante la canalización de las salidas por la función **Paint( )**, el programa será más pequeño y fácil de entender. De este modo, y como ya se mencionó, el programa permite a Windows decidir el mejor momento para actualizar la ventana.

Muchas aplicaciones enrutan todas o muchas de sus salidas a través de la función **Paint( )**, sin embargo ésta es sólo una técnica más y no siempre será la mejor para todos los propósitos.

5.4 Cuadros de mensajes, menús y cuadros de diálogo

Una vez que se sabe como hacer un programa básico para Windows, así como contestar y procesar sus mensajes, es necesario comenzar con los componentes de la interface con el usuario. Una vez que las aplicaciones cumplen con los principios de diseño general para Windows, necesitan comunicarse con el usuario mediante el uso de diferentes tipos de ventanas. Hay tres tipos básicos de ventanas de interface con el usuario: cuadros de mensaje, menús y cuadros de diálogo. Como se podra ver, los tipos básicos de cada una de estas ventanas está predefinido por Windows, sólo es necesario suplir la información necesaria para la aplicación. Hay que tener en mente que los cuadros de mensajes, los menús y los cuadros de diálogo son pequeñas ventanas de la aplicación original, es decir, son propiedad de la aplicación y dependen de ésta. Esta es la razón de la existencia de una ventana principal.

Cuadros de mensajes

Esta es la interface más sencilla que tiene Windows, los cuadros de mensajes simplemente despliegan un mensaje al usuario y esperan por una respuesta, para lo que es posible construir varias alternativas de respuesta dentro de un cuadro pero, en general, el

propósito de los cuadros de mensajes es informar al usuario que algún evento se está llevando al cabo.

Para crear un cuadro de mensaje se usa la función del API llamada **MessageBox( )** de la cual su prototipo es:

int MessageBox(HWND *HWindow*, LPSTR *lpTexto*, LPSTR *lpTitulo*, WORD *wCMTipo*);

Aquí, *HWindow* es el controlador de la ventana predecesora, *lpTexto* es un apuntador a una cadena de caracteres que aparecen dentro del cuadro, la cadena apuntada por *lpTitulo* es usada como título del cuadro y el valor de *wCMTipo* determina la forma exacta del cuadro de mensaje, incluyendo el tipo de botones a utilizar. Algunos de los valores más comúnes para esta función son los de la tabla 5-1. Estas macros son definidas en WINDOWS.H y se pueden usar dos o más al mismo tiempo, ya que no son exclusivas.

Valor	Efecto
MB_ABORTRETRYIGNORE	Despliega botones de abortar, reintentar e ignorar
MB_ICONEXCLAMATION	Despliega icono con signo de admiración
MB_ICONHAND	Despliega icono con signo de STOP
MB_ICONINFORMATION	Despliega un icono de información
MB_ICONQUESTION	Despliega un icono con signo de interrogación
MB_ICONSTOP	Lo mismo que MB_ICONHAND
MB_OKCANCEL	Despliega botones de OK y Cancel
MB_RETRYCANCEL	Despliega botones de Retry y Cancel
MB_YESNO	Despliega botones de Yes y No
MB_YESNOCANCEL	Despliega botones de Yes , No y Cancel

Tabla 5-1

MessageBox( ) retorna las respuestas del usuario al cuadro de mensajes para lo que la siguiente lista contiene las posibles respuestas y sus valores:

Botón oprimido	Valor de retorno
Abort	IDABORT
Retry	IDRETRY
Ignore	IDIGNORE
Cancel	IDCANCEL
No	IDNO
Yes	IDYES
OK	IDOK

Estas macros están definidas en WINDOWS.H y dependera del valor de wCMTipo la forma en que se desplieguen los botones.

Para desplegar un cuadro de mensaje, simplemente se hace el llamado a la función **MessageBox( )**, no es necesario obtener el contexto de recursos o generar un mensaje de **WM_PAINT** ya que ésta maneja todos estos detalles.

El siguiente ejemplo mustra como desplegar algunos cuadros de mensajes al responder a los mensajes del mouse :

```
//   Prototipo de una aplicación para Windows que procesa los mensajes
//    WM_CHAR, WM_PAINT y del mouse con cuadros de mensajes

#include <owl.h>
#include <string.h>
#include <strstream.h>

// Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HANDLE ThisInstance,
                  HANDLE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                                    Args, VidMode) {};
            virtual void InitMainWindow();
};


//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {};
      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMessage Msg) =
```

```
            [WM_FIRST + WM_CHAR];
      virtual void Paint(HDC DC, PAINTSTRUCT &PI);

      virtual void WMLButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_LBUTTONDOWN]; // respuesta al botón izquierdo
      virtual void WMRButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_RBUTTONDOWN]; // respuesta al botón derecho

      // Información adicional de la ventana
};

//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
      MainWindow = new AppWindow(NULL, Name);
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20] = "Hello";
int x=1, y=1;

// Procesamiento al mensaje WM_CHAR
void AppWindow :: WMChar(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      ostr << (char) Msg.WParam << ends; // construye una cadena
      x=1; y=1;
      InvalidateRect(HWindow, NULL, 1);  // Redespliega

}

//Procesamineto del mesanje WM_PAINT con un cuadro de mensaje
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
      //TextOut(DC, 1, 1, s, strlen(s));
      MessageBox(HWindow, s, "TECLA OPRIMIDA", MB_YESNO | MB_ICONSTOP);
}

// Procesamiento del botón izquierdo del mouse
void AppWindow :: WMLButtonDown(RTMessage Msg)
{
      MessageBox(HWindow, "Botón Izquierdo", "Izquierdo",
                  MB_OK | MB_ICONSTOP);
}

// Procesamiento del botón derecho del mouse
void AppWindow :: WMRButtonDown(RTMessage Msg)
{
      int resp;

      resp=MessageBox(HWindow, "Botón Derecho", "Derecho",
                  MB_ABORTRETRYIGNORE | MB_ICONHAND);
```

```
        switch(resp){

                case IDABORT : MessageBox(HWindow, "¿Desea Abortar?",
                               "Abortar", MB_YESNOCANCEL | MB_ICONQUESTION);
                break;
                case IDRETRY : MessageBox(HWindow, "Reintentando... ",
                               "Reintentar", MB_OK | MB_ICONEXCLAMATION);
                break;
                case IDIGNORE : MessageBox(HWindow, "Se ignora", "Ignorar",
                               MB_OK | MB_ICONINFORMATION);
                break;
        }
}

// Parte principal de un programa para Windows
int PASCAL WinMain(HANDLE ThisInstance,HANDLE PrevInstance,
                   LPSTR Args, int VidMode)
{
        AppName App("Cuadros de Mensajes",ThisInstance,
                    PrevInstance, Args, VidMode);

        App.Run();  // Ejecuta una aplicación para Windows

        return App.Status;  // retorna el estatus de corrida
}
```

Como en el programa anterior, en este ejemplo cada vez que se presiona un botón del mouse se despliega un cuadro de mensaje, indicando si fue el derecho o el izquierdo (figura 5.4), además de mostrar algunas variantes al oprimir una tecla (figura 5.5) y al oprimir el boton derecho del mouse (figura 5.6).

Al correr el ejemplo se puede observar que cuando se oprime el botón derecho del mouse se puede elegir entre Abort, Retry e Ignore para las cuales a su vez se crean otros cuadros de mensajes. De este modo se pueden utilizar los cuadros de mensajes para informar simplemente algo al usuario (objetivo principal de éstos) y tener una serie de cuadros mediante una respuesta a los mismos. (figura 5.7)

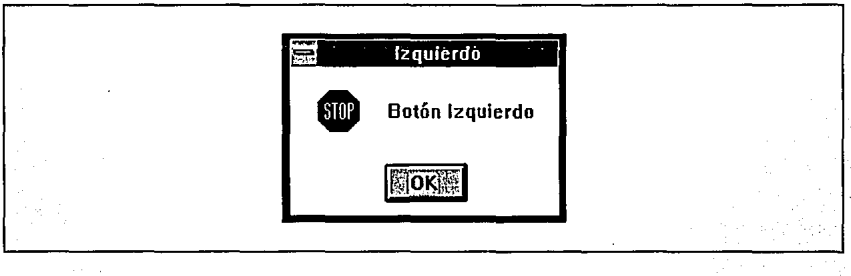


figura 5.4

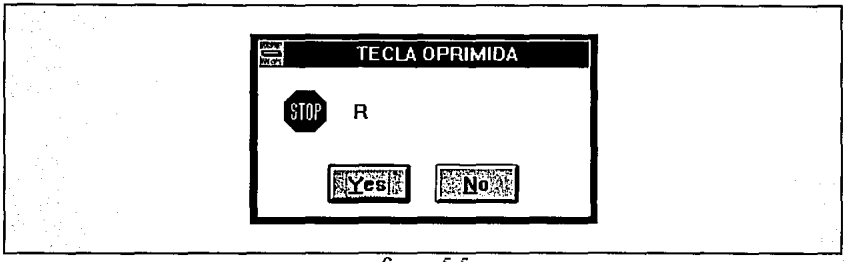


figura 5.5

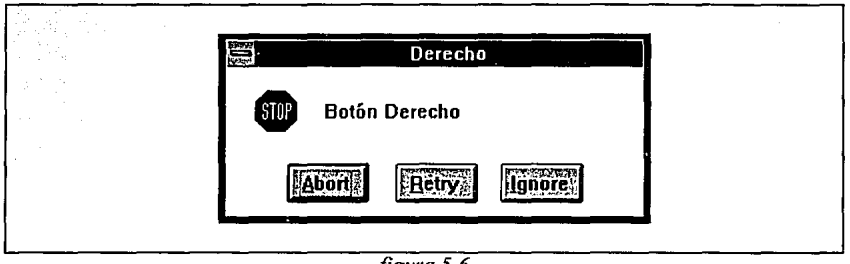


figura 5.6

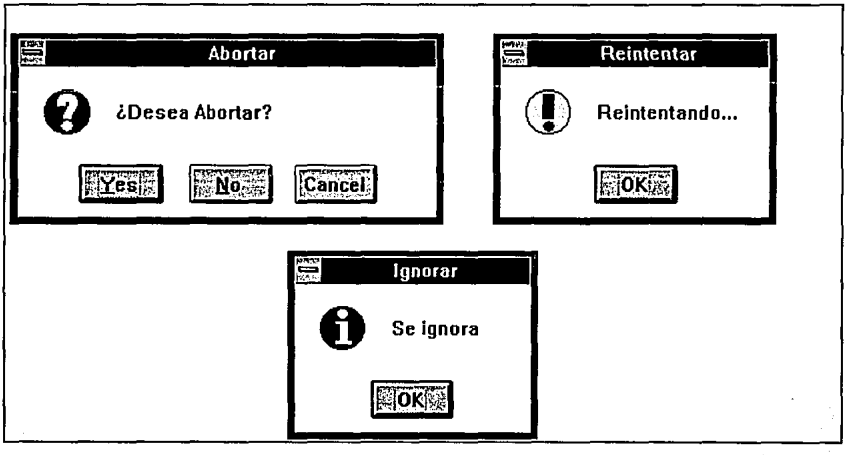


figura 5.7

Creando Menús

Virtualmente todas las ventanas principales tienen algún tipo de menú asociado con ella y debido a que son muy comúnes y de suma importancia en las aplicaciones, Windows tiene un substancial grupo de elementos para soportarlos y crearlos. Para agregar un menú a una ventana existen algunos pasos a seguir:

1. Se define la forma del menú en un archivo de recursos o *resource file.*
2. Cuando se crea la ventana principal se debe jalar o buscar el menú definido.
3. Procesar las opciones del menú.

En Windows, el nivel superior de un menú se despliega a través de la parte superior de la ventana y los submenús se depliegan en forma de pop-up, es decir aparecen de arriba hacia abajo en forma escalonada (figura 5.8)

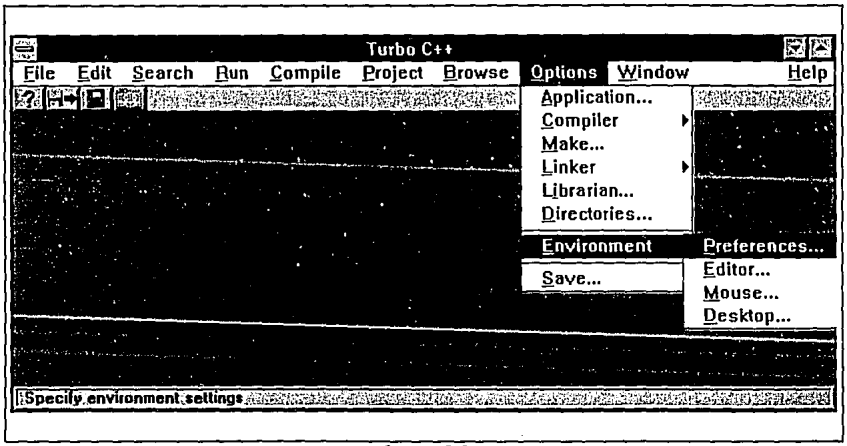


figura 5.8

Windows define muchos tipos de objetos como *recursos* para los cuales se incluyen menús, íconos, cuadros de diálogo y gráficas de tipo *bitmap*. Un recurso es creado de forma separada del código fuente del programa principal pero es agregado al archivo ejecutable (.EXE) cuando se hace el ligado o se corre el *linker*. Los recursos están contenidos en archivos de recursos, para los que se usa la extensión .RC y se crean en un editor de textos normal para archivos Windows. Una vez creado o editado el archivo de recursos debe ser compilado para generar un archivo con extensión .RES, que será ligado al programa principal. Para compilar un archivo de recurso se recomienda usar el

Resource Workshop (no explicado aquí) pero se puede usar el compilador del Turbo C++ para Windows.

Todas las definiciones de menús tienen la siguiente forma general :

NombreMenu MENÚ [*opciones*]
{
    tipos de menú
}

Aquí, *NombreMenu* es el nombre del menú, la palabra reservada **MENÚ** le dice al compilador de recursos que un menú será creado. Existen muchas opciones que pueden ser especificadas al crear un menú, y son:

Opción	**Significado**
DISCARDABLE	El menú puede ser removido de memoria cuando no se necesite
FIXED	El menú permanece fijo en memoria
LOADONCALL	El menú es buscado cuando se necesita (opción por omisión)
MOVEABLE	El menú puede ser movido en memoria (opción por omisión)
PRELOAD	El menú es llamado cuando se ejecuta el programa principal

Opción	Significado
CHECKED	Una marca de chequeo es desplegada seguida del nombre (no se aplica en el nivel superior)
GRAYED	El nombre es mostrado en gris y no puede ser seleccionado.
HELP	El nombre, normalmente "Help" o "Ayuda" es desplegado en la extrema derecha de la barra
INACTIVE	La opción no puede ser seleccionada.
MENUBARBREAK	Para los menús de barras, ocasiona que una barra vertical separe el submenú del anterior. Para los menús Popup causa que el submenú sea puesto en otra columna.
MENUBREAK	Lo mismo que MENUBARBREAK pero no usa barras separadoras.

Tabla 5-2

Para los tipos de menú existen dos opciones que pueden ser usadas para definir el menú: **MENUITEM** el cual especifica una selección final y **POPUP** para especificar un submenú que a su vez puede tener **MENUITEMs** y **POPUPs** definidos. La forma general para estos tipos son:

MENUITEM "*NombreTipo*", *MenuID* [,*opciones*]
POPUP "*NombrePopup*" [,*opciones*]

Aquí, *NombreTipo* es el nombre de la selección en el menú, como *File* (archivo) o *Help* (Ayuda); *MenuID* es un valor entero único asociado a un tipo de menú y que puede ser enviado a la aplicación principal cuando se hace una selección. Típicamente estos valores están definidos como macros dentro de los archivos de cabecera que se incluyen tanto en el programa principal como en el archivo de recursos. *NombrePopup* es el nombre para el menú de tipo popup. Para ambos casos los valores de las opciones se muestran en la tabla 5-2.

El siguiente programa es un ejemplo para crear un menú bastante sencillo:

```
; Ejemplo de un archivo de recursos
#include "menú.h"
MIMENU MENÚ
{
      POPUP "&Uno"
      {
            MENUITEM "&Alfa", IDM_ALFA
            MENUITEM "&Beta", IDM_BETA
      }
      POPUP "&Dos"
      {
            MENUITEM "&Gamma", IDM_GAMMA
            POPUP "&Delta"
            {
                  MENUITEM "&Epsilon", IDM_EPSILON
                  MENUITEM "&Zeta", IDM_ZETA
            }
            MENUITEM "&Teta", IDM_TETA
            MENUITEM "&Omega", IDM_OMEGA
      }
      MENUITEM "&Ayuda", IDM_AYUDA, HELP
}
```

Este menú, llamado MIMENU, contiene tres niveles superiores de opciones en la barra de menús: Uno, Dos y Ayuda; las opciones Uno y Dos contienen submenús de tipo pop-up, en las cuales la opción Delta acciona un nuevo submenú. Hay que notar que las opciones que activan submenús no tiene valores ID asociados con ellos, solamente las opciones del menú tienen números ID. En este ejemplo todos los valores ID están

especificados como macros comenzando con las iniciales **IDM,** definidos en el archivo de cabecera MENÚ.H, y los nombres que se les dan a los valores son arbitrarios.

El símbolo ampersan (&) ocasiona que la letra que le sigue sea la forma corta o rápida de accesar a ese menú mediante el oprimir la tecla que le corresponde. No tiene que ser la primera letra, pero es lo más recomendable a menos que exista una letra repetida.

El código para el archivo de cabecera MENÚ.H es

```
#define IDM_ALFA        100
#define IDM_BETA        101
#define IDM_GAMA        102
#define IDM_DELTA       103
#define IDM_EPSILON     104
#define IDM_ZETA        105
#define IDM_TETA        106
#define IDM_OMEGA       107
#define IDM_AYUDA       108
```

Este archivo define los valores ID que serán retornados cuando varias opciones del menú son seleccionadas y debe ser incluido en cada programa que use el menú. Es importante señalar que aunque los nombres y valores que se les da a las opciones del menú son arbitrarios, deben ser únicos.

Una vez creado el menú, se le debe llamar desde el programa mediante la función **AssignMenu( )** cuando se construye la ventana principal. La función **AssignMenu( )** es miembro de la clase **TWindow** y su prototipo es:

BOOL AssignMenu(LPSTR *NomMenu*);

donde *NomMenu* es el nombre del menú que se desea utilizar y retorna un valor de falso si el menú no pudo encontrarse y de verdadero de otra forma.

Cada vez que el usuario selecciona una opción, el programa envía un mensaje que es usado por el ObjectWindows para llamar a la función DDVT que se ha definido para ejecutar la tarea correspondiente. Las funciones de respuesta para los menús se definen de la misma manera que otra función de respuesta, dentro de la declaración de la clase de la aplicación, con la excepción de sumarle al valor ID la constante CM_FIRST en vez de WM_FIRST. A continuación se lista el ejemplo completo de la demostración de menús y su corrida (figura 5.9):

```
//    Prototipo de una aplicación para Windows que procesa utilizar
//    un menú creado como recurso. Responde a mensajes del mouse con
//    cuadros de mensaje.

#include <owl.h>
```

```
#include <string.h>
#include <strstream.h>
#include "menú.h"

//  Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HINSTANCE ThisInstance,
                  HINSTANCE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                                    Args, VidMode) {};
            virtual void InitMainWindow();
};

//   Define un tipo de ventana
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) : //Asignación de
Menú
            TWindow(WType, WTitle) {AssignMenu ("MIMENU");}
      // Procesamiento de la tecla oprimida
      virtual void WMChar(RTMessage Msg) =
            [WM_FIRST + WM_CHAR];
      virtual void Paint(HDC DC, PAINTSTRUCT &PI);

      virtual void WMLButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_LBUTTONDOWN];  //respuesta al botón izquierdo
      virtual void WMRButtonDown(RTMessage Msg) =
            [WM_FIRST + WM_RBUTTONDOWN];  //respuesta al botón derecho

      virtual void IDMAlfa(RTMessage Msg) = [CM_FIRST + IDM_ALFA];
      virtual void IDMBeta(RTMessage Msg) = [CM_FIRST + IDM_BETA];
      virtual void IDMGama(RTMessage Msg) = [CM_FIRST + IDM_GAMA];
      virtual void IDMEpsilon(RTMessage Msg) = [CM_FIRST + IDM_EPSILON];
      virtual void IDMDelta(RTMessage Msg) = [CM_FIRST + IDM_DELTA];
      virtual void IDMZeta(RTMessage Msg) = [CM_FIRST + IDM_ZETA];
      virtual void IDMOmega(RTMessage Msg) = [CM_FIRST + IDM_OMEGA];
      virtual void IDMAyuda(RTMessage Msg) = [CM_FIRST + IDM_AYUDA];

      // Información adicional de la ventana
};

//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
      MainWindow = new AppWindow(NULL, Name);
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20] = "Hello";
int x=1, y=1;
```

```
// Procesamiento al mensaje WM_CHAR
void AppWindow :: WMChar(RTMessage Msg)
{
      HDC DC;
      ostrstream ostr(s, sizeof(s));

      ostr << (char) Msg.WParam << ends; // construye una cadena
      x=1; y=1;
      InvalidateRect(HWindow, NULL, 1);  // Redespliega

}

//Procesamineto del mesanje WM_PAINT
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
      //TextOut(DC, 1, 1, s, strlen(s));  // despliega s
      MessageBox(HWindow, s, "TECLA OPRIMIDA", MB_YESNO | MB_ICONSTOP);
}

// Procesamiento del botón izquierdo del mouse
void AppWindow :: WMLButtonDown(RTMessage Msg)
{
      MessageBox(HWindow, "Botón Izquierdo", "Izquierdo",
                  MB_OK | MB_ICONSTOP);
}

// Procesamiento del botón derecho del mouse
void AppWindow :: WMRButtonDown(RTMessage Msg)
{
      int resp;

      resp=MessageBox(HWindow, "Botón Derecho", "Derecho",
                        MB_ABORTRETRYIGNORE | MB_ICONHAND);

      switch(resp){
            case IDABORT : MessageBox(HWindow, "¿Desea Abortar?",
                  "Abortar", MB_YESNOCANCEL | MB_ICONQUESTION);
            break;
            case IDRETRY : MessageBox(HWindow, "Reintentando... ",
                  "Reintentar", MB_OK | MB_ICONEXCLAMATION);
            break;
            case IDIGNORE : MessageBox(HWindow, "Se ignora", "Ignorar",
                  MB_OK | MB_ICONINFORMATION);
            break;
      }
}

// Procesamiento de la opción IDM_ALFA
void AppWindow :: IDMAlfa(RTMessage Msg)
{
      ostrstream ostr(s, sizeof(s));

      ostr << "ALFA ALFA" << ends;
      MessageBox(HWindow, "--Alfa--", "Selección Alfa", MB_OK);
```

```
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_BETA
void AppWindow :: IDMBeta(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "BETA BETA" << ends;
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_GAMA
void AppWindow :: IDMGama(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "GAMA GAMA" << ends;
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_DELTA
void AppWindow :: IDMDelta(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "DELTA DELTA" << ends;
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_EPSILON
void AppWindow :: IDMEpsilon(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "EPSILON EPSILON" << ends;
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_ZETA
void AppWindow :: IDMZeta(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "ZETA ZETA" << ends;
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_OMEGA
void AppWindow :: IDMOmega(RTMessage Msg)
{
        ostrstream ostr(s, sizeof(s));

        ostr << "OMEGA OMEGA" << ends;
```

```
        InvalidateRect(HWindow, NULL, 1);
}

// Procesamiento de la opción IDM_Ayuda
void AppWindow :: IDMAyuda(RTMessage Msg)
{
        MessageBox(HWindow, "Demostración de Menús", "Ayuda", MB_OK);
}

// Parte principal de un programa para Windows
int PASCAL WinMain(HINSTANCE ThisInstance,HINSTANCE PrevInstance,
                   LPSTR Args, int VidMode)
{
        AppName App("Demostración de Menús",ThisInstance,
                    PrevInstance, Args, VidMode);
        App.Run(); // Ejecuta una aplicación para Windows
        return App.Status; // retorna el estatus de corrida
}
```

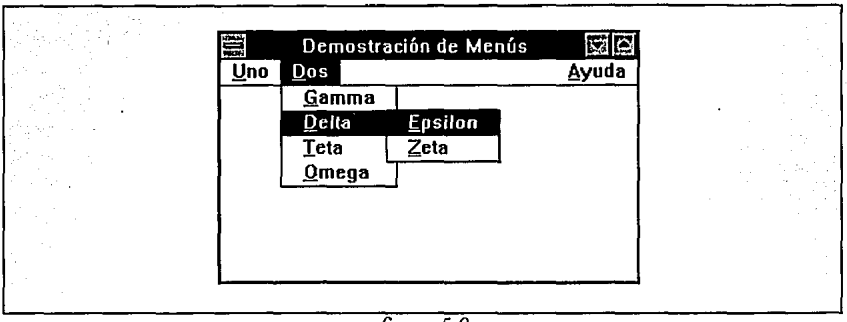


figura 5.9

Existe una característica adicional para los menús llamada tecla rápida, que se refiere a la asignación de una tecla para que la opción deseada se activada con sólo oprimirla sin importar a cuantos submenús tenga que brincarse. La idea de esta característica es que es mucho más rápido el oprimir una tecla para hacer una selección que entrar a cada menú que le anteceda.

Para definir una tecla rápida dentro de un menú se debe agregar una tabla de teclas rápidas al archivo de recursos. Las tablas de definición de teclas rápidas tienen la siguiente forma general:

NomMenu ACCELERATORS
{
    Tecla1, MenuID1, [,tipo] [opción]
    Tecla2, MenuID2, [,tipo] [opción]
    Tecla3, MenuID3, [,tipo] [opción]

TeclaN, MenuIDn, [,tipo] [opción]

}

Aquí, *NomMenu* es el nombre del menú al que se aplicaran las teclas rápidas y el nombre de la tabla de las teclas, *TeclaN* es la tecla que será la tecla rápida de selección y *MenuID* es el valor asociado con la opción deseada; *tipo* especifica si la tecla es standard o es tecla virtual (explicada más adelante); *opción* puede ser una de las siguientes macros: **NOINVERT, ALT, SHIFT** o **CONTROL. NOINVERT** causa que el nivel superior del menú no sea desplegado cuando se selecciona una opción del menú y las otras tres sólo se aplican en teclas virtuales, con una excepción tratada posteriormente.

El valor de *Tecla* puede ser un caracter puesto entre comillas, el valor ASCII correspondiente a la tecla o el código de la tecla virtual. Si se usa un caracter entre comillas el compilador asume que es un valor ASCII, si se usa un valor entero se le debe indicar explícitamente que es un caracter ASCII especificando *tipo* como **ASCII** . Si se usa una tecla virtual, *tipo* debe ser **VIRTKEY**. En el caso de que *Tecla* sea un caracter en mayúscula su opción correspondiente del menú será seleccionada si el usuario la oprime junto con la tecla SHIFT, si *Tecla* es un caracter en minúscula y **ALT** se especifica como *opción*, al oprimir ALT y el caracter se selecciona la opción del menú. Finalmente, si se desea que el usuario oprima la tecla CTRL o CONTROL con el caracter para seleccionar la opción del menú, se especifica la tecla en mayúscula precedida del símbolo ^.

Una tecla virtual es un código independiente para una variedad de teclas, en las que se incluyen las teclas de función F1 a F12, las teclas de flechas y varias teclas que no pertenecen al ASCII. Estas están definidas como macros en el archivo de cabecera WINDOWS.H y todas comienzan con **VK_**, por ejemplo las macros para teclas de función son **VK_F1** a **VK_F12**. Para usar las teclas virtuales como teclas rápidas simplemente se especifica su macro en *Tecla* y **VIRTKEY** en *tipo*, así como agregar **ALT, SHIFT** o **CONTROL** para realizar cualquier combinación. Los siguientes son algunos ejemplos de teclas rápidas:

```
"A", IDM_x              ; selección oprimiendo SHIFT-A
"a", IDM_x              ; selección oprimiendo a
"^A", IDM_x             ; selección oprimiendo CTRL-A
"a", IDM_x, ALT         ; selección oprimiendo ALT-A
VK_F2, IDM_x            ; selección oprimiendo F2
VK_F2, IDM_x, SHIFT     ; selección oprimiendo SHIFT-F2
```

En este ejemplo la declaración IDM_x es el nombre de la opción dentro del menú.

A continuación se lista el archivo de recursos MENÚ.RC que contiene la definición de las teclas rápidas para el menú del programa anterior:

```
; Ejemplo de un archivo de recursos
MIMENU MENÚ
{
      POPUP "&Uno"
      {
            MENUITEM "&Alfa\tF2", IDM_ALFA
            MENUITEM "&Beta\tF3", IDM_BETA
      }
      POPUP "&Dos"
      {
            MENUITEM "&Gamma\tSHIFT-G", IDM_GAMA
            POPUP "&Delta"
            {
                  MENUITEM "&Epsilon\tCntl-E", IDM_EPSILON
                  MENUITEM "&Zeta\tCntl-Z", IDM_ZETA
            }
            MENUITEM "&Teta\tCntl-F4", IDM_TETA
            MENUITEM "&Omega\tCntl-F5", IDM_OMEGA
      }
      MENUITEM "&Ayuda", IDM_AYUDA, HELP
}

;Definición del menú de teclas rápidas
MIMENU ACCELERATORS
{
      VK_F2, IDM_ALFA, VIRTKEY
      VK_F3, IDM_BETA, VIRTKEY
      "G",   IDM_GAMA
      "E",   IDM_EPSILON
      "Z",   IDM_ZETA
      VK_F4, IDM_TETA, VIRTKEY, CONTROL
      VK_F5, IDM_OMEGA, VIRTKEY
      VK_F1, IDM_AYUDA, VIRTKEY
}
```

Siempre que se utilicen teclas rápidas en los menús, se deben llamar de forma separada mediante la función **LoadAccelerators( )** del API y cuyo prototipo es:

HANDLE LoadAccelerators(HANDLE *ThisInstance*, LPSTR *Nombre*);

donde *ThisInstance* es el controlador de la aplicación y *Nombre* es el nombre de la tabla de teclas rápidas. El llamado a esta función debe hacerse dentro de la función **InitMainWindow( )** después de construir la ventana principal, como se muestra a continuación:

```
//  Crea e inicializa un ejemplo de ventana
void AppName::InitMainWindow( )
{
      MainWindow = new AppWindow(NULL, Name);
      HAccTable = LoadAccelerators(hInstance, "MIMENU");
```

}

La variable **HAccTable** es un controlador para la tabla de teclas rápidas y esta definida en la clase base **TApplication.** Para utilizar el menú con las teclas rápidas esta función debe usarse en vez de la del mismo nombre en el ejemplo anterior.

Cuadros de diálogo

Después de los menús no hay elemento más importante en la interface de Windows que los cuadros de diálogo. Un cuadro de diálogo es un tipo de ventana que tiene como finalidad una interacción más flexible entre el usuario y las aplicaciones para Windows, permiten, en general, seleccionar o introducir información que pueda ser muy complicado o imposible hacerlo mediante menús. El cuadro de diálogo es otro elemento en el archivo de recursos de la aplicación y aunque es posible especificar el contenido de los cuadros de diálogo en un archivo de texto normal, es mejor y más fácil hacerlo con el Resource Workshop del compilador.

Los cuadros de diálogo interactúan con el usuario a través de uno o más *controles*, es decir, de uno o más tipos especificos de ventanas de entrada y salida. Un control es propiedad de su ventana antecesora o padre, la cual será generalmente (como en los ejemplos que se presentan aquí) el cuadro de diálogo. Para los cuadros de diálogo, Windows soporta los siguientes controles: botones, cuadros de checado, botones de radio, controles de lista, controles de edición, cuadros de combinación, barras de movimiento y controles estáticos.

♦ Un *botón* es un elemento que el usuario tiene para oprimir una opción, ya sea mediante el mouse o tabulando para colocarse en él y oprimir la tecla enter, para activar alguna respuesta como ya se hizo con los cuadros de mensajes, por ejemplo el botón de OK. Puede haber uno o más botones en un cuadro de diálogo.

♦ Un *cuadro de chequeo* contiene uno o más elementos los cuales pueden ser checados o no, es decir, si un elemento es checado significa que fue seleccionado. Aquí, uno o más elementos pueden ser seleccionados.

♦ Un *control de lista* despliega un lista de elementos de la cual el usuario puede seleccionar uno o varios elementos. Estos son usados comúnmente para desplegar elementos como nombres de archivos.

♦ El *control de edición* permite al usuario introducir una cadena de caracteres ya que esta provista de todas las características necesarias en la edición de un texto. Este cuadro se despliega y espera hasta que el usuario haya terminado de teclear el texto.

♦ Un *cuadro combinado* es el que tiene un control de edición y un control de lista.

♦ La *barra de desplazamiento* es utilizada para desplazar texto en una ventana.

♦ Un *control estático* es usado para dar salida a texto o gráficas con alguna información para el usuario, pero no puede hacer ninguna entrada.

Es importante entender que los controles generan mensajes cuando son accesados por el usuario y reciben mensajes de la aplicación.

Para agregar un cuadro de diálogo a una aplicación se debe definir uno mediante la clase base **TDialog** del ObjectWindows. **TDialog** sirve como una clase base para los cuadros de diálogo de una aplicación y tiene como uno de sus constructores el siguiente prototipo:

TDialog(PTWindowsObject *Propietario*, LPSTR *DNombre*);

Aquí, *Propietario* recibe un apuntador al cuadro de diálogo padre y *DNombre* es el nombre del cuadro de diálogo que se especifica en el archivo de recursos. Cada cuadro de diálogo que se crea será una nueva clase derivada de **TDialog**, por ejemplo, el siguiente código deriva la clase **MiDialog**:

```
// Define un tipo de cuadro de diálogo
class MiDialog : public TDialog
{
public
      MiDialog(PTWindowsObject Propietario, LPSTR DNombre) :
            TDialog(Propietario, DNombre) { }

      // Proceso del cuadro de diálogo
};
```

Ya que un cuadro de diálogo es una ventana (de tipo especial), los eventos que le ocurren son enviados por medio de mensajes al programa del mismo modo que lo hace la ventana principal, por lo que es necesario hacer un conjunto propio de funciones de respuesta para los mensajes de los cuadros de diálogo. Para ésto se utilizan nuevamente las funciones DDVT, el índice de cada función será la combinación de la macro **ID_FIRST**, la cual especifica el punto inicial de todos los mensajes de los cuadros de diálogo, y el identificador ID específico de recurso del control en el cual el cuadro de diálogo genera el mensaje. En general cada control en un cuadro de diálogo dará su propio ID. Cada vez que el control es accesado por el usuario, un mensaje será enviado indicando el tipo de acción que el usuario ha tomado o elegido. El ObjectWindows usa el recurso ID para indicar en la tabla DDVT la llamada a la función asociada con ese control, la función decodifica el mensaje y lleva al cabo las acciones apropiadas.

Antes de que un cuadro de diálogo pueda ser accesado debe ser desplegado, para lo cual se usa la función **ExecDialog( )**, miembro de **TModule**, clase base de **TApplication**. Su prototipo es:
int ExecDialog(PTWindowsObject *DCuadro*);

donde **PTWindowsObject** es un apuntador a un objeto ventana.

Como **ExecDialog( )** es una función miembro (inderectamente) de **TApplication,** su llamado debe estar ligado a la aplicación actualmente activa. Para obtener la aplicación activa, se usa la función miembro **GetApplication( )** la cual retorna un apuntador a la aplicación y su prototipo es:

PTApplication GetApplication( );

Finalmente, para crear el cuadro de diálogo se usa la siguiente instrucción:

GetApplication( ) -> ExecDialog(new D-tipo(this, "D-nombre"));

donde D-tipo es el nombre de la clase de cuadro de diálogo a crear y D-nombre el la especificación del cuadro de diálogo en el archivo de recursos.

El siguiente ejemplo crea un cuadro de diálogo muy simple que contiene tres botones llamados Rojo, Verde y Cancel; cuando alguno de los botones Rojo o Verde es oprimido se activa un cuadro de mensaje indicando que opción se eligió y el cuadro será removido al oprimir Cancel. El programa tiene un menú de nivel superior con tres opciones Diálogo 1, Diálogo 2 y Ayuda. Solamente la opción Diálogo 1 tendrá cuadro de diálogo.

Este será el archivo de recursos para el programa, el cual define un menú para activar el cuadro de diálogo, teclas rápidas y cuadros de diálogo. Su nombre será MIDIALOG.RC

```
;Archivo de recursos para ejemplificar el uso de cuadros de diálogo y
;menús

#include "midialog.h"
MIMENU MENÚ
{
      MENUITEM "Diálogo &1", IDM_DIALOG1
      MENUITEM "Diálogo &2", IDM_DIALOG2
      MENUITEM "&Ayuda", IDM_AYUDA, HELP
}

MIMENU ACCELERATORS
{
      VK_F2, IDM_DIALOG1, VIRTKEY
      VK_F3, IDM_DIALOG2, VIRTKEY
      VK_F1, IDM_AYUDA, VIRTKEY
}

MICD DIALOG 18,18,142, 92
CAPTION "Prueba para Cuadros de Diálogo"
```

```
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
      DEFPUSHBUTTON "Rojo", IDD_ROJO, 32, 36, 28, 13,
                    WS_CHILD | WS_VISIBLE | WS_TABSTOP
      PUSHBUTTON "Verde", IDD_VERDE, 74, 36, 30, 13,
                    WS_CHILD | WS_VISIBLE | WS_TABSTOP
      PUSHBUTTON "Cancel", IDCANCEL, 52, 65, 37, 14,
                    WS_CHILD | WS_VISIBLE | WS_TABSTOP
}
```

El archivo de cabecera será el siguiente:

```
#define IDM_DIALOG1      100
#define IDM_DIALOG2      101
#define IDM_AYUDA        102

#define IDD_ROJO         103
#define IDD_VERDE        104
```

Hasta este punto el cuadro de diálogo requiere responder a dos botones: Rojo y Verde. No necesita responder al botón de Cancel porque ObjectWindows lo hace automáticamente. Las funciones de respuestas para estos botones deben estar en la definición de la clase de diálogo y pueden ser de la siguiente manera:

```
// Define un tipo de cuadro de diálogo
class MiDialog : public TDialog
{
public:
      MiDialog(PTWindowsObject Propietario, LPSTR DNombre) :
            TDialog(Propietario, DName) { }

// Respuesta a los botones Rojo y Verde
virtual void BRojo(RTMessage) = [ID_FIRST + IDD_ROJO];
virtual void BVerde(RTMessage) = [ID_FIRST + IDD_VERDE];
};
```

Las funciones de respuesta para este ejemplo son:

```
// Procesamiento para el mensaje IDD_ROJO
void MiDialog :: BRojo(RTMessage)
{
      MessageBox(HWindow, "Ha oprimido Rojo", "R O J O", MB_OK);
}

// Procesamiento para el mensaje IDD_VERDE
void MiDialog :: BVerde(RTMessage)
{
      MessageBox(HWindow, "Ha oprimido Verde", "V E R D E", MB_OK);
}
```

Cada vez que el usuaro hace un click en un botón, el mensaje asociado con éste es enviado al cuadro de diálogo, es decir que al oprimir el botón de Rojo se ejecuta la función **BRojo( )**.

Controles de lista

Otro control de los cuadros de diálogo es el control de lista, uno de los más comúnes después de los botones. Para su uso, lo primero es agregar la definición de la lista en el archivo de recursos MIDIALOG.RC en la parte de definición del cuadro de diálogo de la forma que se muestra a continuación :

```
MICD DIALOG 18,18,142, 92
CAPTION "Prueba para Cuadros de Diálogo"
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
{
      DEFPUSHBUTTON "Rojo", IDD_ROJO, 32, 36, 28, 13,
                        WS_CHILD | WS_VISIBLE | WS_TABSTOP
      PUSHBUTTON "Verde", IDD_VERDE, 74, 36, 30, 13,
                        WS_CHILD | WS_VISIBLE | WS_TABSTOP
      PUSHBUTTON "Cancel", IDCANCEL, 52, 65, 37, 14,
                        WS_CHILD | WS_VISIBLE | WS_TABSTOP
      CONTROL "Prueba de un control de lista", ID_CL1,
              "Control de lista", LBS_NOTIFY | WS_CHILD | WS_VISIBLE |
               WS_BORDER | WS_VSCROLL , 4, 12, 49, 30
}
```

Y agregar la siguente línea en el archivo de cabecera MIDIALOG.H

```
#define ID_CL1     105
```

donde ID_CL1 identifica el control de lista especificado en la definición del cuadro de diálogo en el archivo de recursos y utilizarlo como índice para las funciones de respuesta apropiadas.

Para responder a los mensajes del control de lista hay que agreagar la siguiente función en la declaración de la clase MiDialog:

```
// Respuesta al control de lista
virtual void CL1(RTMessage) = [ID_FIRST + ID_CL1];
```

Lo siguiente es definir la función de respuesta para responder a los eventos que ocurran dentro del control de lista. Los controles de lista generan varios mensajes y los que se usarán en este ejemplo son:

Macro	**Evento**
LBN_DBLCLK	El usuario ha hecho un doble click en un elemento de la lista o seleccionado mediante los comandos del teclado.
LBN_SETFOCUS	La lista ha adquirido un enfoque de entrada.

Estos mensajes se encuentran en **Msg.LP.Hi.**

El significado de **LBN_DBLCLK** es claro pero el de **LBN_SETFOCUS** es un poco más complejo. Este mensaje es generado cada vez que el control de lista obtiene capacidad para recibir entradas y ocurre sólo cuando el enfoque de entrada esta en todas las partes del cuadro de diálogo, como en los botones o los controles de edición, y después son movidos al control de lista.

A diferencia de un botón, los controles de lista son controles que generan y reciben mensajes. Es posible enviar 26 diferentes mensajes a los cuadros de diálogo, pero para el ejemplo en desarrollo se usarán tres:

Macro	**Propósito**
LB_ADDSTRING	Añade una cadena de caracteres (o selección) a la lista.
LB_GETCURSEL	Solicita el índice del elemento seleccionado.
LB_RESETCONTENT	Borra todos los elementos de la lista.

Para enviar un mensaje al control de lista, o cualquier otro control que lo permita, se usa la función SendDlgItemMsg( ) del ObjectWindows y su prototipo es:

DWORD SendDlgItemMsg(int *ID*, WORD *ID_Msg*, WORD *WParam*, DWORD *LParam*);

Esta función envia al control del cuadro de diálogo a cual ID especificado por ID se le envia que mensaje especificado por ID_Msg. Cualquier información adicional requerida por el mensaje se especifica en WParam y LParam, la cual puede variar de mensaje a mensaje. Si no existe información adicional los valores de estos parámetros deben ser cero.

La siguiente función es para responder al control de lista:

```
// Procesamiento de un cuadro de diálogo
void MiDialog :: CL1(RTMessage)
{
      DWORD i;
      char cadena[80];
      ostrstream ostr(cadena, sizeof(cadena));

      // El control de lista adquiere el enfoque para hacer entradas
      if(Msg.LP.Hi == LBN_SETFOCUS) {
         SendDlgItemMsg(ID_CL1, LB_RESETCONTENT, 0, 0L);

         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Manzana");
         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Naranja");
         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Pera");
```

```
        SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Platano");
    }

    // El usuario hace una selección
    if(Msg.LP.Hi == LBN_DBLCLK) {

        i = SendDlgItemMsg(ID_CL1, LB_GETCURSEL, 0, 0L);
        ostr << "El índice en la lista es: " << i << ends;
        MessageBox(HWindow, cadena, "Selección Hecha", MB_OK);
    }
}
```

En esta función, siempre que se tiene actividad o que el usuario utiliza el control de lista, el mensaje **LB1** es enviado por el cuadro de diálogo, ocasionando que la función **LB1( )** sea llamada. Cuando el control de lista recibe una entrada (por ejemplo un click en la lista), **Msg.LP,Hi** contiene el mensaje **LBN_SETFOCUS**. Cada vez que este mensaje es recibido la función **LB1( )** primero limpia cualquier contenido previo en el control de lista enviando el mensaje **LB_RESETCONTENT**, el cual no requiere información adicional. Ya con el cuadro en límpio, éste busca el conjunto de cadenas de caracteres que formarán los elementos de la lista de la que el usuario podra hacer una selección (por omisión la lista está vacía). Cada cadena de caracteres es agregada al control de lista mediante el llamado a la función **SendDlgItemMsg( )** con el mensaje **LB_ADDSTRING** en el orden en que son enviadas (aunque pueden ser desplegadas en orden alfabético). La cadena que se agregará esta apuntada por el parámetro *LParam* (el cual debe ser forzado al tipo **LONG**) y si el número de elementos de la lista excede el espacio visual del cuadro, una barra de desplazamiento vertical es añadida automáticamente.

Al hacer el usuario una selección de la lista, ya sea mediante el mouse o las teclas de flecha, se envia el mensaje **LBN_DBLCLK** y para determinar que elemento se seleccionó, la función manda el mensaje **LB_GETCURSEL** al control de lista, la cual retorna el índice del elemento.

Controles de edición

El último control que se añadira al ejemplo es un control de edición, los cuales son particularmente útiles ya que permiten al usuario introducir una cadena de caracteres de su propia creación. El primer paso es definir el control de edición en el archivo de recursos agregando las siguientes líneas (a MIDIALOG.RC para este ejemplo) :

```
CONTROL "Default", ID_CE1, "EDIT", ES_LEFT | ES_AUTOHSCROLL |
            WS_CHILD | WS_VISIBLE | WS_BORDER | WS_HSCROLL |
            WS_TABSTOP, 61, 8, 56,19
```

Esta definición crea un control de edición standard con una barra de desplazamiento horizontal. Lo siguente es añadir la siguiente línea al archivo de cabecera MIDIALOG.H:

```
#define ID_CE1          106
```

El siguiente paso es agregar el control de edición al cuadro de diálogo e incluir su función de respuesta en la declaración de la clase **MiDialog** como se muestra a continuación:

```
// Define un tipo de cuadro de diálogo
class MiDialog : public TDialog
{
public:
      MiDialog(PTWindowsObject Propietario, LPSTR DNombre) :
            TDialog(Propietario, DName) { }

// Respuesta a los botones Rojo y Verde
virtual void BRojo(RTMessage) = [ID_FIRST + IDD_ROJO];
virtual void BVerde(RTMessage) = [ID_FIRST + IDD_VERDE];

// Respuesta al control de lista
virtual void LB1(RTMessage) = [ID_FIRST + ID_CL1];

// Respuesta al control de edición
virtual void CE1(RTMessage) = [ID_FIRST + ID_CE1];
};
```

Los controles de edición reconocen muchos mensajes y generan los suyos, por lo que para el propósito de este ejemplo sólo se respondera a un mensaje **EN_KILLFOCUS**, el cual es generado cuando el control de edición ha perdido su enfoque o carácter de servir para entradas. Cuando este mensaje es recibido, la función envia el mensaje **EM_GETLINE** al control de edición lo que ocasiona que haga una copia de la cadena apuntada por el parámetro LParam para que finalmente la función **CE1( )** despliegue la cadena. La función **CE1( )** será la siguiente:

```
// Respuesta al mensaje CE1
void MiDialog :: CE1(RTMessage Msg)
{
  char cadena[80];

  if (Msg.LP.Hi == EN_KILLFOCUS) {
    cadena[SendDlgItemMsg(ID_CE1, EM_GETLINE, 0, (LONG) cadena)] = '\0';
    MessageBox(HWindow, cadena, cadena, MB_OK);
  }
}
```

Al comenzar la ejecución del ejemplo completo de los cuadros de diálogo sólo se despliega el nivel superior del menú en la barra de menús. Al seleccionar Diálogo 1, se desplegará el cuadro de diálogo, en el que a su vez, al oprimir un botón, se ejecutará la respuesta adecuada.(figuras 5.10 y 5.11):

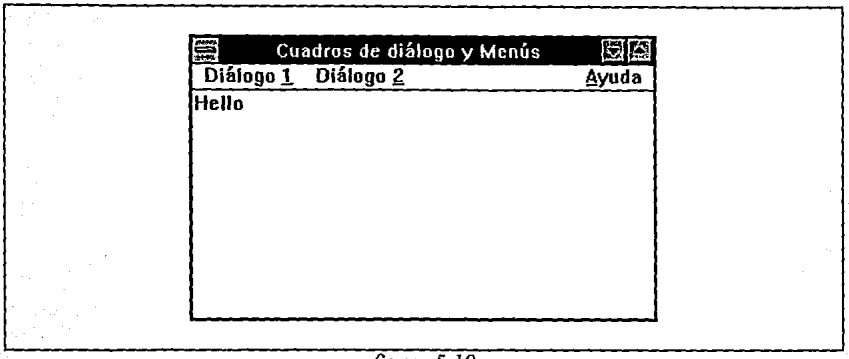


figura 5.10

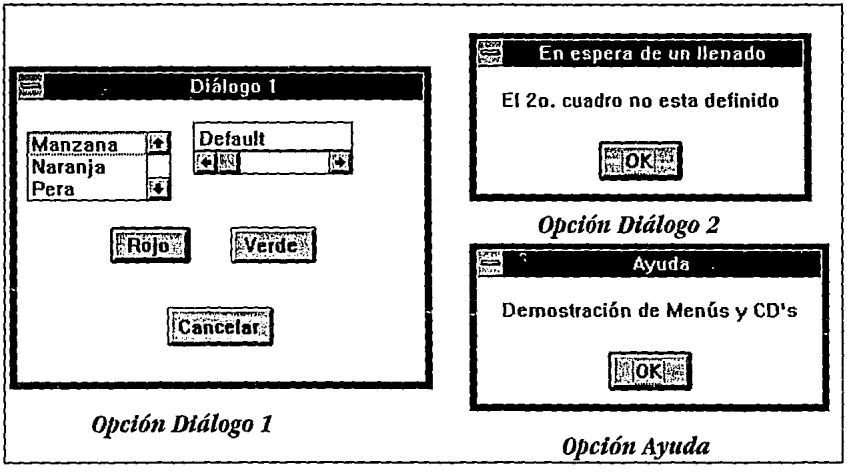


Figura 5.11

El listado completo para este programa se encuentra en el anexo.

CONCLUSIONES

No cabe la menor duda de que el término orientado a objetos está llamando la atención en todos los sectores del mundo de la computación, desde usuarios hasta programadores y fabricantes de paquetes. Tratándose de sistemas y en beneficio del largo plazo, las empresas ahora ven que, hasta el momento, uno de los mejores caminos a seguir es el orientado a objetos.

Hoy en día el término orientado a objetos ha rebasado su ámbito de origen en el área de lenguajes de programación y es usado de manera extensiva como sinónimo de lo novedoso, lo bueno y lo interesante. Aún así, el impacto de la programación orientada a objetos sólo puede compararse con la implantación de la programación estructurada en los años 70's.

La POO induce un cambio que hace más que incorporar nuevos mecanismos de programación; su impacto está en el cambio de perspectiva en el desarrollo de sistemas, que pasa de ser una actividad basada en una colección de subrutinas de uso específico, a un conjunto de componentes reutilizables. Con esto se convierte en la esperanza de la ingeniería de sistemas, pero su aplicación requiere de un enfoque totalmente distinto a lo tradicional ya que requiere de un esfuerzo de actualización y capacitación por parte de académicos, profesionistas y estudiantes.

Para obtener al máximo los beneficios de la POO es necesario un cambio de mentalidad, lo que implica, más que cambiar de lenguaje, ir hacia la nueva perspectiva en desarrollo de sistemas: la programación basada en componentes reutilizables o con prototipos. Sin embargo me he dado cuenta que no satisface cualquier necesidad, por lo que se debe de hacer un análisis completo de lo que se requiere antes de optar por un desarrollo basado en objetos.

No creo que deban desaparecer o sustituir las demás técnicas de programar, sino que la POO debe considerarce una forma más para resolver algunos problemas, sobre todo en la reutilización de componentes. Aún así, las aplicaciones desarrolladas en base a la POO no aseguran un "sello de calidad" como se maneja el área publicitaria de productos para desarrollo de programas.

Después de haber hecho un análisis de los términos y componentes de la POO, se puede concluir que C++ ofrece la mejor opción para comenzar a programar con orientación a objetos, ya que, por su naturaleza híbrida, nos permite crear programas muy semejantes a los que normalmente hacemos y en especial como técnicas de programación avanzada.

Un punto importante a mencionar es que el desarrollo de sistemas expertos, lenguajes de 4a. generación y los nuevos sistemas operativos tienen sus bases fundamentadas en el paradigma de la POO, que se caracterizan principalmente por tener interface gráfica y recursos reutilizables al momento de hacer modificaciones o mejoras en sus versiones. Tal es el caso de los sistemas operativos OS/2 de IBM, UNIX/V versión 4 (con Motif como interface gráfica), Windows de Microsoft, etc. que principalmente desarrollan sus librerías en lenguaje C y C++.

También cabe mencionar que C++ no es sólo un superconjunto de C, es más bien, un mejor lenguaje C. Los beneficios de un lenguaje de alto nivel y la programación orientada a objetos crea una combinación muy poderosa e interesante, más para los administradores de sistemas que para los desarrolladores o programadores. El incremento en la productividad, calidad y mantenimiento de las aplicaciones son aspectos a considerar por los programadores, pero los administradores de sistemas son los que los necesitan.

El desarrollo en C++ tiene la desventaja de llevarnos a utilizar la estructuración clasica en los programas, pero a base de práctica lo considero la mejor forma de emigrar a la POO ya que se conoce la sintaxis, el ambiente, etc. Una vez logrado ésto, es posible usar lenguajes orientados a objetos puros (C++ y ObjectPascal son híbridos) como SmallTalk o Eiffel con los que se desarrolla la POO de manera aún más robusta y eficiente.

Finalmente, considero de suma importancia el desarrollo de aplicaciones para plataformas gráficas y en especial para Windows, ya que es el sistema más comercial para microcomputadoras tipo PC y seguramente el sustituto de DOS. Sin embargo, como mencioné en la introducción, se requiere de mucha preparación para lograr un completo uso del API de Windows ya que consta de más de 600 funciones, por lo que la librería ObjectWindows de Borland es una herramienta muy útil para facilitar la programación para Windows.

BIBLIOGRAFIA

Turbo C++ for Windows 3.0 User's guide
Borland
California 1992

Turbo C++ for Windows 3.0 Programer's guide
Borland
California 1992

SCHILDT, Herbert; Turbo C/C++, the complete reference
Borland Osborne/McGraw-Hill
California 1991

SCHILDT, Herbert; Using Turbo C++
Borland Osborne/McGraw-Hill
California 1992

WEISKAMP,Keith; Object-Oriented Programing with Turbo C++
Wiley
Colorado 1991

VOSS, Greg; Object-Oriented Programing, an introduction
Osborne/McGraw-Hill
California 1991

MORRILL, Jane; State of art: Object Lessons
BYTE, octubre 1990
McGraw-Hill
Nueva York, E.U.

ANEXO

```
//    Prototipo de una aplicación para Windows con cuadros de diálogo
// Programa principal Midialog.cpp

#include <owl.h>
#include <dialog.h>
#include <string.h>
#include <strstream.h>

#include "dialog.h"

//  Define una aplicación
class AppName : public TApplication
{
      public:
            AppName(LPSTR Ap_Name, HINSTANCE ThisInstance,
                  HINSTANCE PrevInstance, LPSTR Args, int VidMode) :
                  TApplication (Ap_Name, ThisInstance, PrevInstance,
                        Args, VidMode) {};
            virtual void InitMainWindow();
};

//    Define un tipo de ventana
_CLASSDEF(AppWindow)
class AppWindow : public TWindow
{
public:
      AppWindow(PTWindowsObject WType, LPSTR WTitle) :
            TWindow(WType, WTitle) {AssignMenu("MIMENU");};

      virtual void Paint(HDC DC, PAINTSTRUCT &PI);

      virtual void IDMDIALOG1(RTMessage) = [CM_FIRST + IDM_DIALOG1];
      virtual void IDMDIALOG2(RTMessage) = [CM_FIRST + IDM_DIALOG2];
      virtual void IDMAYUDA(RTMessage) = [CM_FIRST + IDM_AYUDA];

      // Información adicional de la ventana
};

// Define un tipo de cuadro de diálogo
class MiDialog : public TDialog
{
public:
      MiDialog(PTWindowsObject Propietario, LPSTR DNombre) :
            TDialog(Propietario, DNombre) { }

// Respuesta a los botones Rojo y Verde
virtual void BRojo(RTMessage) = [ID_FIRST + IDD_ROJO];
virtual void BVerde(RTMessage) = [ID_FIRST + IDD_VERDE];

// Respuesta al cuadro de lista
virtual void CL1(RTMessage Msg) = [ID_FIRST + ID_CL1];
```

```
// Respuesta al cuadro de edición
virtual void CE1(RTMessage) = [ID_FIRST + ID_CE1];
};


//  Crea e inicializa una ventana
void AppName::InitMainWindow( )
{
      MainWindow = new AppWindow(NULL, Name);

     HAccTable = LoadAccelerators(hInstance, "MIMENU");
}

// Este arreglo es global porque será utilizado por varias funciones
char s[20] = "Hello";
int x=1, y=1;

// Procesamiento al mensaje WM_PAINT
void AppWindow :: Paint(HDC DC, PAINTSTRUCT &)
{
      TextOut(DC, x, y, s, strlen(s)); // Da salida a la cadena
}

//Procesamiento de la selección IDM_DIALOG1
void AppWindow :: IDMDIALOG1(RTMessage)
{
      GetApplication( ) -> ExecDialog(new MiDialog(this, "MICD"));
}

//Procesamiento de la selección IDM_DIALOG2
void AppWindow :: IDMDIALOG2(RTMessage)
{
      MessageBox(HWindow, "El 2o. cuadro no esta definido",
                  "En espera de un llenado", MB_OK);
}

//Procesamiento de la selección IDM_AYUDA
void AppWindow :: IDMAYUDA(RTMessage)
{
      MessageBox(HWindow, "Demostración de Menús y CD's",
                  "Ayuda", MB_OK);
}

// Procesamiento para el mensaje IDD_ROJO
void MiDialog :: BRojo(RTMessage)
{
      MessageBox(HWindow, "Ha oprimido Rojo", "R O J O", MB_OK);
}

// Procesamiento para el mensaje IDD_VERDE
void MiDialog :: BVerde(RTMessage)
{
      MessageBox(HWindow, "Ha oprimido Verde", "V E R D E", MB_OK);
```

```
}

// Procesamiento de un cuadro de lista
void MiDialog :: CL1(RTMessage Msg)
{
      DWORD i;
      char cadena[80];
      ostrstream ostr(cadena, sizeof(cadena));

      // El cuadro de lista adquiere el enfoque para hacer entradas
      if(Msg.LP.Hi == LBN_SETFOCUS) {
         SendDlgItemMsg(ID_CL1, LB_RESETCONTENT, 0, 0L);

         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Manzana");
         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Naranja");
         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Pera");
         SendDlgItemMsg(ID_CL1, LB_ADDSTRING, 0, (LONG) "Platano");
      }

      // El usuario hace una selección
      if(Msg.LP.Hi == LBN_DBLCLK) {

         i = SendDlgItemMsg(ID_CL1, LB_GETCURSEL, 0, 0L);
         ostr << "El índice en la lista es: " << i << ends;
         MessageBox(HWindow, cadena, "Selección Hecha", MB_OK);
      }
}


// Respuesta al mensaje CE1
void MiDialog :: CE1(RTMessage Msg)
{
  char cadena[80];

  if (Msg.LP.Hi == EN_KILLFOCUS) {
    cadena[SendDlgItemMsg(ID_CE1, EM_GETLINE, 0, (LONG) cadena)] = '\0';
    MessageBox(HWindow, cadena, cadena, MB_OK);
  }
}


int PASCAL WinMain(HINSTANCE ThisInstance,HINSTANCE PrevInstance,
                   LPSTR Args, int VidMode)
{
      AppName App("Cuadros de diálogo y Menús",ThisInstance,
                  PrevInstance, Args, VidMode);

      App.Run();  // Ejecuta una aplicación para Windows
      return App.Status;  // retorna el estatus de corrida
}
```

```
; Archivo de recursos para ejemplificar el uso de cuadros de diálogo y
; menús

#include "dialog.h"

MIMENU MENU
BEGIN
      MENUITEM "Diálogo &1", IDM_DIALOG1
      MENUITEM "Diálogo &2", IDM_DIALOG2
      MENUITEM "&Ayuda", IDM_AYUDA, HELP
END

MIMENU ACCELERATORS
BEGIN
      VK_F2, IDM_DIALOG1, VIRTKEY
      VK_F3, IDM_DIALOG2, VIRTKEY
      VK_F1, IDM_AYUDA, VIRTKEY
END

MICD DIALOG 18, 22, 142, 102
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Diálogo 1"
BEGIN
      DEFPUSHBUTTON "Rojo", IDD_ROJO, 32, 46, 28, 13, WS_CHILD |
      WS_VISIBLE | WS_TABSTOP
      PUSHBUTTON "Verde", IDD_VERDE, 74, 46, 30, 13, WS_CHILD |
      WS_VISIBLE | WS_TABSTOP
      PUSHBUTTON "Cancelar", IDCANCEL, 52, 75, 37, 14, WS_CHILD |
      WS_VISIBLE | WS_TABSTOP
      CONTROL "Cuadro de lista", ID_CL1, "LISTBOX", LBS_NOTIFY |
      WS_CHILD | WS_VISIBLE |
      WS_BORDER | WS_VSCROLL | WS_TABSTOP, 4, 12, 49, 30
      CONTROL "Default", ID_CE1, "EDIT", ES_LEFT | ES_AUTOHSCROLL |
      WS_CHILD |
      WS_VISIBLE | WS_BORDER | WS_HSCROLL |WS_TABSTOP, 61, 8, 56, 19
END


// Archivo Include Midialog.h para los índices

#define IDM_DIALOG1     100
#define IDM_DIALOG2     101
#define IDM_AYUDA       102

#define IDD_ROJO        103
#define IDD_VERDE       104

#define ID_CL1          105

#define ID_CE1          106
```