

TALLER DE PROGRAMACION I

Copyright (C) Gabriel Agustín Praino.

El presente material se encuentra registrado en el Registro Nacional de Propiedad Intelectual.

Prohibida la reproducción total y/o parcial sin autorización escrita, con excepción de las expresamente autorizadas en esta licencia.

Se autoriza el uso, distribución y/o reproducción del presente material, sin fines de lucro, como capacitación personal, sujeto a las condiciones mencionadas a continuación.

Se autoriza el uso, distribución y/o reproducción del presente material en toda entidad de enseñanza pública y gratuita, sujeta a las condiciones mencionadas a continuación.

Condiciones de distribución y reproducción

- No podrá cobrarse ningún importe ni servicio, bajo ningún concepto, por la distribución y/o reproducción de este material.
 - Toda reproducción de este material deberá realizarse en forma completa y sin modificaciones.
 - Toda reproducción de este material deberá mantener el presente copyright. Se prohíbe expresamente todo uso, distribución y/o reproducción de este material con fines comerciales, sin autorización escrita.
-

PARTE I - El lenguaje de programación C.....	7
Capítulo I - Comenzando.....	7
Introducción.....	7
Estructura de un Programa C.....	7
Primer programa en C.....	9
Turbo C / Borland C para D.O.S.....	9
Borland C para Windows.....	9
Microsoft C / Visual C para Windows.....	9
Agregando comentarios en C.....	11
Tipos de Datos.....	11
Definición y alcance de las Variables.....	12
La función printf().....	14
Algunos operadores.....	16
Ejemplos de programas en C.....	17
Resumen.....	18
Capítulo II - Funciones y prototipos.....	19
Definiciones de constantes.....	22
Declaraciones vs. definiciones.....	24
Resumen.....	27
Capítulo III - Sentencias de control.....	28
Sentencias de control de programa (if).....	28
Recursividad.....	30
Evaluación de condiciones.....	31
Sentencias de control de programa (switch-case).....	33
ifs anidados.....	36
Sentencias de control de programa (while y do-while).....	37
Sentencias de control de programa (for).....	38
Incluso pueden omitirse los tres parámetros de la sentencia for, escribiendo: for (;). En este caso, este código será equivalente a: while (1).....	39
Sentencias break y continue.....	39
Sentencia goto.....	41
Operador ?.....	42
Resumen.....	43
Capítulo IV - Asignaciones y operadores.....	43
Operadores.....	44
Operadores y asignaciones.....	46
Notación entera.....	47
Capítulo V - Definición de tipos y conversiones.....	49
Estructuras.....	49
Enumeraciones.....	50
Uniones.....	51
Definición de nuevos nombres para tipos de datos.....	53
Convirtiendo tipos de datos (cast).....	53
Capítulo VI - Vectores.....	56
Pasaje de vectores a funciones.....	60
Más sobre el pasaje de vectores a funciones.....	61
Capítulo VII - Matrices.....	65
Pasaje de matrices a funciones.....	67
Capítulo VIII - Punteros.....	70
Introducción.....	70

¿Qué es un puntero?	70
Primer ejemplo con punteros	73
Nota de sintaxis	74
Más acerca del pasaje de variables a funciones	76
Capítulo IX - Memoria dinámica y punteros a vectores	80
Pasaje de vectores a funciones	81
Aritmética de punteros	82
El modificador const y los punteros	85
Capítulo X - Punteros y estructuras de datos	88
Capítulo XI - Archivos	90
Streams	92
Entrada/salida standard	93
Entrada/salida sin streams	93
Capítulo XII - Temas varios de C	94
Variables static	94
Variables volatile	94
¿Cómo recibir parámetros de la línea de comandos?	95
Punteros a funciones	96
El preprocesador	97
Capítulo XIII - Estructuras de datos	101
Listas enlazadas	101
PARTE II - El lenguaje de programación C++	106
Capítulo XIV - Introducción a C++	106
Introducción	106
C vs C++	106
Capítulo XV - Comenzando a programar en C++	108
Algunas variantes de C++	108
Nuevos tipos de datos	112
Capítulo XVI - Programación orientada a objetos	113
PROGRAMACIÓN ORIENTADA A OBJETOS	113
Objetos	113
Polimorfismo	113
Herencia	113
LAS CLASES	114
Creación de la estructura Pila	115
Capítulo XVII - Parámetros por defecto y sobrecarga de funciones	125
PARÁMETROS POR DEFECTO	125
SOBRECARGA DE FUNCIONES	125
Capítulo XVIII - Herencia	128
HERENCIA	128
Introducción a la herencia múltiple	132
Distribución en memoria de atributos, en clases heredadas	133
Capítulo XIX - Casting de objetos	134
Casting de objetos	134
Nota	134
Tipos de casting entre objetos	135
Los constructores y la herencia	137
Capítulo XX - Sobrecarga de operadores y funciones friend	138
OPERADORES	138
Tabla 2: Operadores sobrecargables	142
PARTE III - Programación avanzada en C++	147
Capítulo XXI: Entrada/Salida en C++	147

Métodos constantes.....	148
Capítulo XXII - Templates.....	150
Clases template dentro de clases template.....	156
EL PREPROCESADOR.....	157
Capítulo XXIII - Un poco más sobre herencia - Herencia múltiple.....	159
Capítulo XXIV - Métodos virtuales y funciones puras y abstractas.....	163
LA HERENCIA VIRTUAL Y EL COMPILADOR.....	165
Capítulo XXV - Utilización de memoria dinámica en clases - Constructor de copia.....	167
La memoria dinámica y los objetos.....	167
Constructor de copia.....	170
Otro ejemplo.....	172
Objetos temporales.....	180
Resumen de puntos a tener en cuenta.....	182
Capítulo XXVI - Memoria compartida entre objetos.....	184
Capítulo XXVII - Sobrecarga de operadores new y delete.....	186
Alternativa 1:.....	188
Alternativa 2:.....	189
Capítulo XXVIII - Manejo de excepciones.....	191
APÉNDICE I - Palabras reservadas de C.....	196
Palabras reservadas de C.....	196
APÉNDICE II - Palabras reservadas de C++.....	196
Palabras reservadas de C++.....	196
Apéndice III - Caracteres reservados.....	198
Apéndice IV - EL PREPROCESADOR.....	199
Macros predefinidas.....	199
Directivas del preprocesador.....	199
Apéndice V - Mezclando código C y C++.....	200
Apéndice VI - Resumen de funciones especiales de C++.....	202
PARTE IV – WINDOWS API.....	203
Capítulo I - Introducción a la programación bajo Windows.....	203
Introducción.....	203
La filosofía de programación Windows.....	203
¿Qué es Windows?.....	204
Muy bien, pero ¿qué hay de nuevo?.....	204
Capítulo II - Comenzando.....	209
Comenzando.....	209
Primer Ejemplo de programa Windows.....	211
Archivo EJEMPLO1.C.....	211
Archivo RESOURCE.H.....	213
Archivo EJEMPLO1.RC.....	213
Analicemos el programa.....	216
Analicemos ahora el código C.....	217
¿Quién procesa los mensajes restantes?.....	221
Múltiples ventanas.....	222
Ventanas modales.....	223
Un poco más sobre ventanas.....	223
Ventanas STATIC:.....	224
Ventanas BUTTON:.....	224
Ventanas EDIT:.....	224
Ventanas SCROLLBAR:.....	225

Ventanas LISTBOX y COMBOBOX:	225
Dibujando en pantalla:	226
Ejemplo2.c	226
Ejemplo2.rc	227
resource.h	229
Redibujando la ventana	230
Capítulo III - Creación de una ventana	233
Paso 1: Registrar un modelo de ventana básico	233
Paso 2: Crear la ventana	234
El ciclo de mensajes y threads	236
La necesidad de una nueva estructura de programación	238
Capítulo IV - Hacia un nuevo modelo de programación Windows	245
La necesidad de un nuevo modelo de programación Windows	245
Borland - OWL (Object Windows Library)	245
Microsoft MFC	248
Paso 1: Creación del proyecto:	248
Paso2: Agregar una variable String y mostrarla	249
Paso3: Agregar un dialogo	251

PARTE I - El lenguaje de programación C

Capítulo I - Comenzando

Introducción

El C es un lenguaje de programación estructurado, también llamado funcional. Esto quiere decir que el mismo estará formado por un conjunto de variables y funciones, que se llamarán unas a otras.

A estas funciones y variables de un programa se las identifica mediante un nombre llamado **identificador**, que puede estar compuesto de caracteres alfanuméricos (comenzando en una letra), y se distinguen las mayúsculas de las minúsculas. Por ejemplo, las variables "Base", "BASE" y "base" serán diferentes. También son válidas variables como "registro_2", pero no "2_registro" ni "registro 2".

Si bien la longitud de los mismos es arbitraria, C define que tan sólo los primeros 32 caracteres serán considerados.

De todas las funciones que componen un programa, existe una especial llamada **main**. Esta función será llamada por el sistema operativo y será por lo tanto el punto de entrada al programa (el lugar donde empieza la ejecución). Cuando esta función termina, el programa se finaliza y se devuelve el control al sistema operativo.

Estructura de un Programa C

Un programa en C consta de las siguientes partes, si bien este orden no es estricto en absoluto:

- Archivos a incluir
- Declaraciones de tipos de datos, estructuras, constantes y macros
- Definiciones de variables globales (no utilizar indiscriminadamente)
- Prototipos de funciones
- Funciones del programa

Cada una de estas partes se irán viendo más adelante. Por ahora basta saber que estas funciones estarán formadas a su vez por llamadas a otras funciones o a si mismas, de sentencias de control y/o de asignaciones.

En C no hay distinción entre funciones y procedimientos como en otros lenguajes. De hecho, solamente existen funciones, si bien los procedimientos pueden ser fácilmente implementase mediante funciones. Las funciones en C pueden recibir un número arbitrario de parámetros (incluso variable) y pueden devolver un único valor.

Toda función se define como sigue:

```
Tipo_de_dato_devuelto nombre_funcion (lista_de_parametros)
{
    Cuerpo de la función
}
```

Si la función no devuelve ningún dato, o no toma argumentos, debe utilizarse la palabra reservada **void**.

En C toda instrucción o sentencia debe terminarse con punto y coma (;), y el cuerpo de una función (proposición) se encierra entre llaves.

Primer programa en C

Debido a la gran variedad de compiladores disponibles en el mercado, sería una tarea muy extensa explicar aquí la forma en que un programa puede ser compilado. Sin embargo, haremos algunos comentarios para los compiladores más usados:

Turbo C / Borland C para D.O.S

Puede escribirse el programa en un único archivo, y desde el menú compilar el programa. Puede también crearse un proyecto (project) que incluya varios archivos.

Borland C para Windows

Puede escribirse el programa en un único archivo y compilarlo desde el menú. Se generará un programa que correrá en una ventana Windows o DOS (dependiendo de la versión del compilador). Puede también crearse un proyecto (project) que incluya varios archivos.

Microsoft C / Visual C para Windows

Debe crearse un proyecto (WorkSpace), en el cual debe incluirse el archivo .c/.cpp a compilar.

Veamos nuestro primer programa en C



Ejemplo 1.1: Programa 'nada.c'

```
void main (void)
{
}
```

Este es el programa más sencillo que puede escribirse, y que no hace nada.

Simplemente se define una función de nombre **main**, que no toma ningún parámetro y no devuelve nada. Finalmente, ya que el cuerpo de la función es {}, la misma no hace nada.

El nombre de la función, **main**, no es casual. Todo programa debe tener una y tan sólo una función **main**, que es donde comienza la ejecución del programa. Es decir, al ejecutarse el programa, lo que hace el sistema operativo es llamar a la función **main**.



La función `main()` constituye el punto de entrada a un programa C.

Ahora bien, pasemos a algún programa que sí haga algo:

Ejemplo 1.2: Programa 'hola.c'

```
#include <stdio.h>

void main (void)
{
    printf ("Hola mundo\n");
}
```

La salida de este programa es la siguiente:

Hola mundo

La primer línea del programa, indica que debe incluirse un archivo, `stdio.h`, donde se define la entrada/salida standard de C, entre las cuales está la función ***printf()***. En nombre del archivo está escrito entre símbolos `<>`, lo que indica que debe buscársele en el directorio `include`. En algunos compiladores, este directorio se especifica mediante una instrucción 'SET include = ...' en el sistema operativo, otros mediante una opción en algún menú desplegable, y otros leen la variable `PATH` del sistema operativo. Si el nombre del archivo se escribiese entre comillas (""), el archivo a incluir se buscara en primer lugar en el directorio actual.

La línea que contiene la palabra `printf()` consiste en una llamada a la función `printf`, pasándole como parámetro el texto `Hola mundo\n`.

La función ***printf()*** (***print*** with ***format***) puede tomar un número cualquiera de parámetros, como se verá más adelante.

El carácter `\` (barra invertida) está reservado para imprimir caracteres especiales. El carácter `%` también está reservado, pero su uso se verá más adelante. Los principales caracteres especiales son:

<code>\n</code>	salto de línea
<code>\t</code>	tabulación horizontal
<code>\r</code>	retorno de carro
<code>\\</code>	barra invertida
<code>\0</code>	byte 0

Veamos otro programa en C

Ejemplo 1.3: Más acerca de la salida standard

```
#include <stdio.h>

void main (void)
{
    printf ("Hola mundo del C\nEste es mi ");
    printf ("primer programa en C.\n");
    printf ("Ahora dejaré una línea en blanco\n\ny seguiré escribiendo.\n");
}
```

La salida de este programa es la siguiente:

```
Hola mundo del C
Este es mi primer programa en C.
Ahora dejaré una línea en blanco

y seguiré escribiendo
```

Notar que pueden insertarse saltos de línea (\n) en medio del texto, y no es necesario terminar una línea en \n, como tampoco es obligatorio escribir una línea en un único printf(). Sin embargo, en muchos sistemas operativos el texto no será impreso en pantalla hasta no encontrar un \n.

Agregando comentarios en C

Los comentarios en C se escriben entre símbolos /* y */. Los mismos pueden contener dentro, cualquier tipo de símbolo o carácter ascii, incluso saltos de línea.

Agreguémosle comentarios al programa anterior:

Ejemplo 1.4: Comentarios en un programa

```
/* *****\
 * Programa de prueba *
 \*****/
#include <stdio.h>

/* Aqui comienza el programa */
void main (void)
{
    /* *****/
    printf ("Hola mundo\n"); /* Impresión en pantalla */
}
```

Todo texto encerrado entre símbolos /* y */ será interpretado como un comentario y no será compilado

Tipos de Datos

C soporta los siguientes tipos de datos básicos, definidos por el compilador. Notar que puede variar el significado de cada uno de ellos según la plataforma sobre la que se esté trabajando.

Tipo de dato	Significado	Longitud habitual (en bits)
char	Caracter / Entero (-128 a 127)	8
unsigned char	Caracter / Entero (0 a 255)	8
short int	Entero reducido (-127 a 128 ó -32768 a 32767)	8 ó 16
unsigned short int	Entero reducido (0 a 255 ó 0 a 65535)	8 ó 16
int	Entero con signo	16 ó 32

unsigned int (ó unsigned)	Entero sin signo	16 ó 32
long int	Entero largo con signo	32
unsigned long int (ó unsigned long)	Entero largo sin signo	32
long long ó long64 ó _int64	Entero largo. No forma parte del standard pero varios compiladores lo implementan.	64
float	Punto flotante IEEE	32
double	Punto flotante doble precisión	64
long double	Punto flotante doble precisión	80 ó 96
void	Sin valor	indeterminado

El tipo de dato puede variar de un sistema a otro. Por ejemplo, un entero (int), se compone de 16 bits en sistemas operativos de 16 bits, y de 32 bits en sistemas.

Si bien tantos tipos de datos pueden parecer difícil de recordar, basta saber que existen básicamente dos tipos de datos: enteros y de punto flotante (reales), llamados int y float/double respectivamente, y que existen modificadores (prefijos) unsigned, long, etc, para controlar el uso de signo y su longitud. Existe también el prefijo signed, opuesto a unsigned, pero rara vez se lo usa ya que, por defecto las variables son con signo (puede especificarse al compilador que no sea así, pero por convención nunca se lo hace)

Debe hacerse una aclaración con el tipo de dato char:



El tipo de dato **char** debe ser interpretado como un número entero de 8 bits, no como un carácter, si bien en la práctica se lo suele usar para este fin. Incluso, como todo entero, el mismo puede ser definido con o sin signo. Más aún, los sistemas operativos avanzados utilizan el standard lenguaje UNICODE en lugar de ASCII, donde cada carácter se representa con un entero de 16 bits (short).

Definición y alcance de las Variables

Las variables se definen escribiendo su nombre (identificador), que puede constar de hasta 32 caracteres alfanuméricos, precedidos por el tipo de variable. Toda definición de variable debe necesariamente terminar en punto y coma (;). Veamos un ejemplo:



Las variables tienen un alcance local al bloque donde fueron definidas.



Ejemplo 1.5: Definición de variables

```
void main (void)
{
    /* Defino tres variables */
    int a;
    unsigned int hola;
    float radio;
}
```

En este caso se están definiendo tres variables. La primera llamada **a**, capaz de almacenar valores enteros con signo, la segunda llamada **hola**, capaz de almacenar enteros sin signo, y la tercera llamada **radio** permitirá almacenar valores reales, también llamados de punto flotante.

Es posible definir varias variables en un único renglón, separadas por coma.

Ejemplo 1.6: Definición de variables

```
void main (void)
{
    /* Defino las variables a, b y c de tipo entero */
    int a, b, c;
}
```

Las variables definidas en cualquier función o bloque de código, es decir, en cualquier bloque delimitado por llaves {}, tiene un alcance local relativo a ese mismo bloque. Esto quiere decir que al salir del bloque donde fueron definidas dejarán de existir.

La única excepción son las variables definidas fuera de cualquier función, que tienen alcance global.

En cualquier parte de una función, al comienzo de un bloque delimitado por llaves, se pueden declarar variables, las cuales tendrán el alcance antes mencionado.

No puede definirse ninguna variable en un bloque, luego de ejecutarse alguna instrucción o sentencia.

Puede definirse varias variables del mismo tipo en una única línea, separando estas por comas. Ej: **int a,b,c,d;**

Es posible asignar un valor a una variable utilizando el operador igual (=).

Ejemplo 1.7: Asignación de variables

```
void main (void)
{
    /* Defino tres variables */
    int a, b;    /* Defino dos variables de tipo entero, llamadas a y b */

    a = 5;      /* Cargo el valor 5 en la variable a */
    b = a;      /* Copio el valor de a en b */
}
```

También es posible asignar valores iniciales a las variables en el momento de crearlas. Veamos un ejemplo:

Ejemplo 1.8: Asignación de valores iniciales

```
void main (void)
```

```
{
    /* Defino tres variables */
    /* Defino dos variables de tipo entero, llamadas a y b*/
    int a = 5, b = 2;
    /* y una de tipo 'punto flotante' */
    float f = 4.5;
}
```

La función printf()

La función **printf()** (**print** with **format**) permite la impresión en pantalla, en una forma muy versátil y poderosa, si bien un tanto compleja. Esta función está declarada en el archivo **stdio.h**, y recibe una cantidad aleatoria de parámetros, pero el primero especifica la forma y cantidad de parámetros que se reciben. Hay dos caracteres especiales cuyo uso está reservado. Ellos son \ (barra invertida) y % (porcentaje).

El carácter \ (barra invertida) está reservado para imprimir caracteres especiales. Los fundamentales son:

\n	salto de línea
\r	retorno de carro
\t	tabulación horizontal
\\	barra invertida
\"	comillas
\0	byte 0.

El carácter % está reservado para imprimir campos, y el siguiente o siguientes caracteres especifican el tipo de parámetro que se pasará y la forma en que debe imprimirse. Las principales opciones son:

%%	símbolo %
%d	int
%u	unsigned
%ld	long
%lu	unsigned long
%c	char
%s	char* (strings) (se verá más adelante)
%f	float

En el primer parámetro de la función printf se define la forma en que se va a imprimir el texto en pantalla, escribiendo un símbolo % (porcentaje) donde deba imprimirse un valor pasado como parámetro. Estos parámetros se pasan a continuación. Veamos algunos ejemplos:

Ejemplo 1.9: La función printf - Impresión de parámetros

```
#include <stdio.h>

void main (void)
{
    int a, b;
    a = 10;
    b = 5;
    printf ("El valor de a es %d y el valor de b es %d.\n", a, b);
}
```

Esto imprimirá en pantalla:

El valor de a es 10 y el valor de b es 5.

Veamos otro ejemplo:

Ejemplo 1.10: La función printf - Impresión de parámetros (2)

```
/* Este programa es incorrecto */
#include <stdio.h>

void main(void)
{
    float a;
    a = 4.5;
    printf ("El valor de a es %d.\n", a);
}
```

El programa anterior es incorrecto, si bien será compilado normalmente y no se reportará ningún error. Lo que ocurre es que la variable a es de tipo real, y en el primer parámetro de la función printf se dice que se pasará un parámetro entero. La forma correcta de llamar a la función printf en este caso sería:

```
printf ("El valor de a es %f.\n", a);
```

Debe recordarse que el tipo de dato char almacena un entero de 8 bits, y por lo tanto es posible imprimir su valor numérico, al igual que todo entero. Esto se hace utilizando la opción %d de la función printf. También es posible imprimir el carácter ascii representado por un entero, ya sea este char, short int o int, con o sin signo. Veamos un ejemplo:

Ejemplo 1.11: Impresión de caracteres

```
void main(void)
{
    char a;
    int b;

    a = 'A';
    b = 97;

    printf ("El valor de a es %d\n", a);
    printf ("El valor de b es %d\n", b);
    printf ("El caracter representado por a es '%c'\n", a);
    printf ("El caracter representado por b es '%c'\n", b);
}
```

Este programa producirá la siguiente salida:

```
El valor de a es 65
El valor de b es 97
El caracter representado por a es 'A'
El caracter representado por b es 'a'
```

La función `printf()` tiene muchas opciones y formatos de impresión que no se analizarán aquí. Se da simplemente el siguiente programa como ejemplo:

Ejemplo 1.12: Impresión con formato

```
void main(void)
{
    int a;
    int b;
    float f;

    a = -32;
    b = 97;
    f = 10.0/3.0;
    printf ("El valor de b es: %05d\n", b);
    printf ("El valor de a es: %-6u\n", b);
    printf ("El valor de f es: %5.2f\n", f);
}
```

El primer `printf()` especifica que el primer parámetro será de tipo entero (`%05d`), y deberá imprimírselo utilizando 5 espacios (`%05d`) completando los restantes con 0 (`%05d`), con justificado derecho.

El segundo `printf()` indica que se imprimirá un entero sin signo (`%-6u`), con formato izquierdo (`%-6u`), utilizando 6 espacios (`%-6u`).

El tercer `printf()` indica que se imprimirá un número de punto flotante (`%5.2f`), utilizando 5 espacios (`%5.2f`), dos de los cuales se destinarán a la parte decimal (`%5.2f`).

La salida de este programa será:

```
El valor de b es: 00097
El valor de a es: -32
El valor de f es:  0.33
```

Algunos operadores

El lenguaje C define varios operadores, los cuales se verán más adelante. Un operador es un símbolo que permite realizar operaciones matemáticas, lógicas o binarias entre dos valores. Existen entre otros los operadores:

+	suma
-	resta
*	multiplicación
/	división
%	resto de una división (sólo para números enteros).

Ejemplo 1.13: Los operadores

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



```
void main (void)
{
    float radio = 2;
    float pi = 3.14;

    printf ("La circunferencia de un círculo de radio %f es %f\n",
           radio, 2 * pi * radio);
}
```

La salida de este programa será:

La circunferencia de un círculo de radio 2 es 12.56

Notar cómo la llamada a la función printf() se escribió en dos renglones. Ya que el C no tiene en cuenta los saltos de línea, tabulaciones o estilos de programación, es posible escribir una sentencia en varios renglones, para que la misma resulte más comprensible.

Ejemplos de programas en C

Ejemplo 1.14: Operaciones aritméticas

```
#include <stdio.h>

void main (void)
{
    int a, b, c;
    int prom;

    a = 1;
    b = 2;
    c = 3;

    prom = (a + b + c) / 3;
    printf ("El promedio de a, b y c es: %d\n", prom);

    prom = a/3 + b/3 + c/3;
    printf ("El promedio de a, b y c es: %d\n", prom);
}
```

El programa produce la siguiente salida:

```
El promedio de a, b y c es: 2
El promedio de a, b y c es: 1
```

El programa anterior calcula el promedio de los números a, b y c. Notar que se obtuvieron dos resultados diferentes para un mismo cálculo, realizado de formas diferentes pero matemáticamente equivalentes.

En el primer caso se sumo los números y luego los divido por 3.

$(1+2+3)/3$

6/3

2

En el segundo, si bien matemáticamente la operación es la misma, debe tenerse en cuenta que se está trabajando con números enteros, y por lo tanto la parte decimal se truncará, en cada una de las operaciones.

La cuenta que el programa hará es la siguiente:

$1/3 + 2/3 + 3/3$

$0 + 0 + 1$

1

Con lo que la variable `prom` quedará en 1.

Resumen

Un programa en C se compone principalmente de una serie de funciones. Estas funciones reciben parámetros por valor (no existe el pasaje de parámetros por referencia como en otros lenguajes), es decir, que las funciones reciben copias de los valores de las variables. Las funciones pueden o no devolver un valor.

Entre todas las funciones posibles, existe una llamada ***main()***, que es la función por donde comienza la ejecución del programa. Al terminar esta función, el programa termina.

Los compiladores C ya traen una serie de bibliotecas, que pueden ser incluidas en el programa mediante la directiva ***#include*** entre las cuales se encuentra la librería ***stdio***, que define funciones de entrada/salida, como la función ***printf()***.

Las funciones se componen de sentencias y llamadas a otras funciones. Dentro de cada función pueden definirse variables, y su alcance, conocido como ***scope***, está limitado a la función o bloque donde se definió la variable. Pueden definirse variables globales, y las mismas estarán disponibles en todo el programa.

Salvo las variables globales, ninguna variable es inicializada, y sus valores iniciales no están definidos.

Capítulo II - Funciones y prototipos

Hasta ahora hemos escrito todo nuestro programa en una única función **main**. Está claro que en cuanto el programa comience a crecer, será incomodo trabajar dentro de la misma función. El problema se complica aún más cuando deben realizarse operaciones largas en forma repetida. Por ejemplo, imaginemos tener que calcular la superficie de varias figuras en forma reiterada. Sería muy cómodo poder abstraerse de la fórmula concreta en cada caso, al realizar el ciclo principal del programa.

Al llamar a una función con parámetros se crearán copias de los mismos, y estos serán pasados a la función. Esto quiere decir que toda función recibirá una **copia** de la variable original, y por lo tanto la alteración de este valor no modificará el valor original.



En C todas las variables se pasan por valor. No existe el pasaje de parámetros por referencia, si bien se lo puede implementar.

Veamos un ejemplo.



Ejemplo 2.1: Definición de funciones

```
#include <stdio.h>

float pi = 3.14;

float SuperficieCirculo (float radio)
{
    return 2 * pi * radio;
}

float VolumenCilindro (float radio, float altura)
{
    float vol;
    vol = SuperficieCirculo (radio) * altura;
    return vol;
}

void main(void)
{
    float radio = 3.0;
    float altura = 4.5;
    printf ("El volumen de un cilindro de radio %f y altura %f es %f",
           radio, altura, VolumenCilindro (radio, altura));
}
```

Este programa imprimirá el siguiente texto:

El volumen de un cilindro de radio 3 y altura 4.5 es 42.39

Notar como se definió la variable global pi, y se le dio un valor inicial 3,14. Esta variable conservará este valor en tanto no se la modifique. Sería muy grave que por un error de programación se modifique el valor de esta variable, ya que se producirían errores en todos los resultados. Para evitar que esto suceda, puede

utilizarse la palabra reservada **const**. Simplemente debe anteponerse esta palabra a la declaración de la variable, y esta indicará que la misma deberá conservar el valor durante todo el programa o la ejecución de la función, e impedirá que la misma sea modificada. Por ejemplo, en nuestro caso deberíamos haber escrito:

```
const float pi = 3.14;
```

Se definieron también dos funciones, llamadas *SuperficieCirculo* y *VolumenCilindro*, cada una de las cuales puede tomar uno y dos parámetros respectivamente de tipo real, y devuelve un valor de tipo real. La devolución de un valor se hace utilizando la palabra reservada **return**. El valor que esté a continuación de esta palabra será devuelto a la función original.

Ya que la función devuelve un valor, es posible copiar el mismo en una variable, como se hace al llamar a la función *SuperficieCirculo* desde *VolumenCilindro*, o pasarlo como parámetro a otra función como en el caso de la llamada a *VolumenCilindro* en la función *printf*.

Sólo se puede llamar a una función que ha sido previamente declarada. Por ejemplo, en nuestro ejemplo anterior no sería posible escribir la función *main* antes de la función *VolumenCilindro*, ya que la misma no habría sido aún declarada. De idéntica forma, tampoco podría definirse la función *VolumenCilindro* antes de la función *SuperficieCirculo*.

Esto es, NO se puede llamar a una función que aún no ha sido declarada, aunque esta se encuentre en forma inmediata a continuación. Esto obligaría a que todas las funciones deban ordenarse cuidadosamente, y nunca se podría llamar de una función a otra y de esta otra a la anterior.

Para solucionar este inconveniente, debe informársele al compilador la existencia de una o más funciones, antes de que estas sean escritas. Esto se conoce con el nombre de **declaración de función**, y a esta declaración de la suele llamar **prototipo de la función**. Veamos nuevamente nuestro ejemplo anterior, pero declarando las funciones.

Ejemplo 2.2: Prototipos de funciones

```
#include <stdio.h>

const float pi = 3.14;

/***** Declaro las funciones del programa *****/
/* Prototipo de la función VolumenCilindro */
float VolumenCilindro (float radio, float altura);
/* Prototipo de la función SuperficieCirculo */
float SuperficieCirculo (float radio);

/***** Defino las funciones del programa *****/

void main(void)
{
    float radio = 3.0;
    float altura = 4.5;
    printf ("El volumen de un cilindro de radio %f y altura %f es %f",
           radio, altura, VolumenCilindro (radio, altura));
}
```

```
float VolumenCilindro (float radio, float altura)
{
    float vol;
    vol = SuperficieCirculo (radio) * altura;
    return vol;
}

float SuperficieCirculo (float radio)
{
    return 2 * pi * radio;
}
```

Definiciones de constantes

Existe una segunda forma de definir constantes (la que generalmente se utiliza), mediante **macros** o **definiciones**. Estas se definen utilizando la cláusula `#define`, seguida del nombre y luego el valor, **SIN PUNTO Y COMA** al final. Por ejemplo, podría escribirse:

```
#define PI 3.1416
#define IVA 21
#define MAX_PATH 128 /* Máxima cantidad de caracteres en el nombre de un archivo
*/
```

En este caso, no se está definiendo variables, sino macros que serán reemplazadas por el preprocesador **antes de compilar** el código. Veamos un ejemplo completo:

Ejemplo 2.3: Prototipos de funciones (2)

```
#include <stdio.h>

#define PI 3.14

/* Prototipos de las funciones del programa */
float SuperficieCuadrado (float lado);
float SuperficieRectangulo (float ancho, float alto);
float SuperficieCirculo (float radio);
float SuperficieElipse (float RadioMenor, float RadioMayor);

/* Bloque principal del programa */
void main (void)
{
    float lado;
    float superf;

    lado = 10.2;

    printf ("La superficie de un cuadrado de lado %f es %f\n",
        lado, SuperficieCuadrado (lado));

    superf = SuperficieRectangulo (5,6);
    printf ("La superficie de un rectangulo de %f x %f es %f\n", 5, 6, superf);

    printf ("La superficie de un circulo de radio %f es %f\n", 8,
        SuperficieCirculo (8));
}

/* Funciones para calcular la superficie de las figuras */
float SuperficieCuadrado (float lado)
{
    return SuperficieRectangulo (lado, lado);
}

float SuperficieRectangulo (float ancho, float alto)
{
```

```
    return ancho * alto;
}

float SuperficieCirculo (float radio)
{
    return SuperficieElipse (radio, radio);
}

float SuperficieElipse (float RadioMenor, float RadioMayor)
{
    return PI * RadioMenor * RadioMayor;
}
```

Notar que en este ejemplo, la constante PI no se definió utilizando la palabra reservada **const**, sino **#define**. Esta es una **directiva** o **directriz** que indica al precompilador que reemplace todas las palabras PI por 3.14. Este es el caso más simple de la definición de una macro. Las **macro** constituyen una herramienta muy poderosa en C, y se las verán con más detalle más adelante. Sólo se reemplazan palabras completas. Por ejemplo, en la palabra: ESPIA, PI no será reemplazado.

¿Qué diferencia hay entre const y #define?

Rta: Si bien muchas veces puede resultar indistinto utilizar **const** o **#define**, su significado es muy diferente. **const** es un modificador que especifica que el valor de una variable permanecerá constante, en tanto que **#define** define un texto que deberá ser reemplazado por el precompilador antes de compilar el programa.

Veamos otro ejemplo. Escribamos ahora una función que intercambie el contenido de dos variables:

Ejemplo 2.4: Modificación de parámetros de las funciones

```
#include <stdio.h>

void Intercambiar (int a, int b)
{
    int aux;

    aux = a;
    a = b;
    b = aux;

    printf ("En Intercambiar: a=%u, b=%u\n", a, b);
}

void main(void)
{
    int a, b;
    a = 4;
    b = 5;

    printf ("Originalmente: a=%u, b=%u\n", a, b);
    Intercambiar (a, b);
    printf ("De vuelta en main: a=%u, b=%u\n", a, b);
}
```

La salida de este programa es la siguiente:

Originalmente: a=4, b=5
En Intercambiar: a=5, b=4
De vuelta en main: a=4, b=5

Las variables a y b fueron efectivamente intercambiadas dentro de la función Intercambiar(), pero al volver a la función original las variables seguían teniendo el valor original. ¿Qué ha sucedido?

Rta: como ya dijimos, en C sólo existe pasaje de parámetros por valor, es decir que las funciones recibirán **copias** de los parámetros con que fueron llamadas. No hay forma en que una función pueda modificar los parámetros con que fue llamada.

¿Implica esto que es imposible escribir una función para intercambiar el contenido de dos variables?

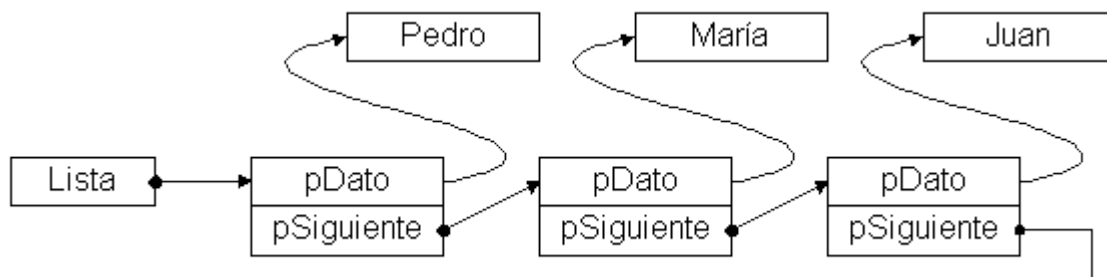
Rta: No, no es así. Existe una forma de hacerlo, pero esto se verá más adelante, en el tema punteros.

Declaraciones vs. definiciones

Hasta ahora hemos visto cómo escribir algunos programas básicos, y hemos utilizado las palabras **declaración** y **definición** sin precisar qué significa cada una de ellas.

El lenguaje de programación C fue desarrollado para escribir programas de gran envergadura, también denominados proyectos, en los cuales comúnmente trabajarán muchas personas, y donde hablar de 50 mil o 100 mil líneas de código es habitual. Es por eso ilógico pensar que todo el programa esté en un único archivo.

Para solucionar este inconveniente, los proyectos se subdividen en partes más pequeñas (código fuente), que se compilan por separado para formar código objeto o librerías, que luego se unirán (en inglés link) para formar un único ejecutable. El segundo paso aquí mencionado (link) suele estar automatizado, con lo cual es transparente al programador, no así el primero (compilación). Este proceso se muestra esquemáticamente como sigue:



Debe señalarse que el proceso de compilación es, por lo general, mucho más costoso que el de link. Cuando un programa se subdivide en varios archivos, sólo se compilarán aquellos archivos que hayan sufrido modificaciones desde la última compilación. Esto hace que, la subdivisión del proyecto en varios archivos disminuye los tiempos de compilación.

Ya que el programa se subdividirá en muchos archivos, es necesario informar a cada una de estas partes el contenido de las demás. Por ejemplo, si en un módulo se llamará a la función Factorial(), es necesario informarle al mismo que tipos de parámetros recibe y devuelve, para poder llamarla.

De idéntica forma, si tengo una variable global en uno de los módulos, debo informarle al resto de los módulos a cerca de su existencia y de su tipo.

Este concepto de informar al compilador la existencia de un tipo de variable o de función se conoce con el nombre de declaración. En este caso se informa al compilador como debe ser utilizada la misma, pero este no reservará memoria para la variable ni sabrá cómo ejecutar la función.

Por el contrario, al definirse una variable, el compilador reservará memoria para la misma o, en el caso de una función, se dará la forma precisa su contenido.

En el caso se funciones la declaración se realiza poniendo el título completo de la función, seguido de punto y coma. El nombre de los parámetros es opcional. Ej:

```
unsigned Factorial (unsigned n);
```

ó

```
unsigned Factorial (unsigned);
```

A esto se lo conoce también con el nombre de prototipos de funciones.

En el caso de las variables esto se hace anteponiendo la palabra reservada extern. Ej:

```
extern int Salir;  
extern unsigned VarGlobal;
```

Toda variable o función debe estar declarada o definida antes de que puede ser utilizada. Es por esto que generalmente todas las declaraciones se almacenan en archivos .h, en tanto que las definiciones se almacenan en archivos .c.

De esta forma, todo programa que incluya un archivo .h sabrá de la existencia de variables globales y funciones en otros archivos.



Toda variable o función debe haber sido declarada antes de poder ser utilizada.



Las declaraciones a compartir entre varios archivos deben escribirse en archivos .h.

Si bien es posible, por regla general nunca debe incluirse un archivo .C dentro de otro archivo .C. Sí es posible que archivos .H incluyan a su vez otros archivos .H. Se verá este tema más adelante.

Ejemplo 2.5: Declaración vs. definición

Archivo arch1.c

```
#include <stdio.h>

/* Incluyo el archivo matem.h que contiene las declaraciones de la función
   VolumenCilindro. Uso comillas ya que el archivo matem.h no estará en el
   directorio include sino en el mismo directorio que este archivo.
*/
#include "matem.h"

void main (void)
{
    float radio = 3.0; /* Notar que escribo 3.0 en vez de 3, para
                        que quede claro que se trata de un número real */
    float altura = 4.5;
    printf ("El volumen de un cilindro de radio %f y altura %f es %f",
           radio, altura, VolumenCilindro (radio, altura));
}
```

Archivo matem.h

```
#ifndef __MATEM_INCLUDED__
#define __MATEM_INCLUDED__

/* Declaro la variable pi */
extern float pi;

/* Declaro las funciones VolumenCilindro y SuperficieCirculo mediante sus
   respectivos prototipos */
float VolumenCilindro (float radio, float altura);
float SuperficieCirculo (float radio);

#endif
```

Archivo matem.c

```
/* Defino la variable pi */
float pi = 3.14;

/* Defino las funciones VolumenCilindro y SuperficieCirculo dando el código
   completo de las mismas */
float VolumenCilindro (float radio, float altura)
{
    float vol;
    vol = SuperficieCirculo (radio) * altura;
    return vol;
}

float SuperficieCirculo (float radio)
{
    return 2*pi*radio;
}
```

Notar en la segunda línea del archivo matem.h la definición de la constante `__MATEM_INCLUDED__`. Ya que un mismo archivo .h puede ser incluído desde varios otros archivos, puede ocurrir que un mismo archivo sea incluído varias veces. Incluso, podría ocurrir que dos archivos .h se incluyan uno al otro, lo cual se conoce con el nombre de referencias cruzadas (cross references). Para evitar que en un mismo archivo .c se incluya un archivo .h varias veces, la técnica utilizada consiste en preguntar si una constante ya fue definida, y en caso negativo, definirla e incluir el código correspondiente.

Las palabras `#ifndef`, `#define` y `#endif` son directivas o directrices, y son interpretadas por el preprocesador, antes de compilar el código. Volveremos sobre este tema más adelante. Por ahora basta saber que todo archivo .h debe utilizar esta técnica para evitar ser incluído varias veces, y que línea `#ifxxx` se cierra siempre en una línea `#endif`. El formato es siempre el siguiente:

Archivo .h

```
#ifndef CTE
#define CTE
...
#endif
```

Resumen

Hemos visto que un proyecto grande puede ser subdividido en varios archivos, cada uno de los cuales será compilado por separado. Para que en todos los archivos .C se conozcan las funciones existentes en otros archivos o módulos del proyecto, los mismos deben incluir archivos .H, utilizando la directiva `#include`. Estos archivos tendrán las declaraciones o prototipos de las funciones y variables existentes en otros módulos.

Capítulo III - Sentencias de control

Las sentencias de control de programas son la esencia de cualquier lenguaje de programación, ya que gobiernan el flujo de ejecución del programa.

Las sentencias de control de programa pueden agruparse en tres grupos. La primera está formada por las instrucciones condicionales **if** y **switch**. La segunda por las sentencias de control de bucles **while**, **do-while** y **for**. La tercera es la instrucción de ramificación incondicional **goto**, que por ir en contra de la programación estructurada, se recomienda no utilizar.

Sentencias de control de programa (if)

La sentencia **if**, ejecutará la sentencia o sentencias que se encuentren a continuación si y sólo si se cumple la condición especificada. El formato es el siguiente:

```
if (condición)
    sentencia;
else
    sentencia;
```

Si hubiese que ejecutar varias sentencias debería encerrárselas entre llaves.

```
if (condición)
{
    sentencia;
    ...
    sentencia;
}
else
{
    sentencia;
    ...
    sentencia;
}
```

Un ejemplo de esto es el siguiente:

Ejemplo 3.1: Sentencias de control if

```
/* Función división */
int Division(int a, int b)
{
    int r;

    /* Notar que la comparación se realiza con el doble signo = (==) */
    if (b==0)
    {
        printf ("Error\n");
    }
}
```

```
    return 0;    /* Es obligatorio devolver un valor */  
}  
  
r = a / b; /* En tanto que la asignación se realiza tan sólo con = */  
return r;  
}
```

Recursividad

Ya que el C es un lenguaje de programación funcional, nada impide llamar a una función dentro de sí misma, incluso tantas veces como uno quiera (teniendo en cuenta limitaciones de memoria). A esto se lo conoce con el nombre de **recursividad**, y consiste en resolver un problema complejo, utilizando para ello la misma herramienta varias veces. Esto puede resultar muy útil para resolver ciertos problemas, como por ejemplo el cálculo del factorial de un número.

Ejemplo 3.2: Recursividad

```
#include <stdio.h>

int factorial (int n)
{
    if (n == 0)
        return 1;
    return n * factorial(n-1);
}

void main(void)
{
    int n = 5;
    printf ("El factorial de %u es %u\n", n, factorial (n));
}
```

¿Qué hubiese sucedido si hubiésemos llamado a la función con el número -3?

Pues bien, la función se hubiese seguido llamando a sí misma indefinidamente con números negativos (-4, -5,...), hasta llegar al máximo número negativo que puede representarse. Cuando a este número se le reste uno, se producirá un error conocido como **overflow**, y el número resultante será el máximo entero **positivo** que puede representarse. De esta forma se continuará hasta volver llegar al número 0.

No es importante comprender en este momento en forma detallada la forma en que esto ocurre, sino tan sólo que esto constituye un error de programación, y que ni el programa ni el compilador informarán nunca de este error, ya que el programa es sintácticamente correcto.

Lo que ocurre es que la función factorial está definida únicamente para números positivos (más el cero). Esto se conoce con el nombre de **dominio de una función**. Toda función debería validar que los parámetros recibidos son correctos. Ya que esta función sólo puede recibir números positivos, lo correcto es que reciba números sin signo:

```
unsigned factorial (unsigned n)
{
    if (n == 0)
        return 1;
    return n * factorial (n - 1);
}
```

De todas formas si quisiésemos calcular el factorial de 1000, trabajando en un sistema operativo de 16 bits, basta notar que $1000 \cdot 999 = 999000$, que no es un número representable con 16 bits (el máximo es $2^{16}-1 = 65535$). Nuevamente esto no producirá ningún error, pero el resultado será incorrecto. Concretamente la forma en que se hará esto es multiplicando estos dos números, pero se conservarán sólo los 16 bits más bajos del resultado.

En forma similar podemos calcular números de la secuencia de fibonacci, que se define como:

```
Fib (0) = 1
Fib (1) = 1
Fib (x) = Fib (x-1) + Fib (x-2), para x>=2
```

Ejemplo 3.3: Más sobre recursividad

```
#include <stdio.h>

unsigned Fib(unsigned n)
{
    if (n < 2)
        return 1;
    return Fib (n-1)+Fib(n-2);
}

void main(void)
{
    unsigned n;
    n = 10;
    printf ("El número %u de Fibonacci es %u\n", n, Fib (n));
}
```

Evaluación de condiciones

Vimos en los programas anteriores que las comparaciones se efectúan con == (dos símbolos =).

Los restantes símbolos de comparación son:

==	igual
!=	diferente
<	menor
>	mayor
<=	menor igual
>=	mayor igual

Si se debe efectuar una comparación de varios parámetros, deben usarse && ó ||.

&&	Y
	O

Implementemos una función para saber si un día es navidad.

Ejemplo 3.4: if-else

```
int Navidad (int dia, int mes)
{
    if (dia == 25 && mes == 12)
        return 1;
    else
        return 0;
}
```

```
int Navidad (int dia, int mes)
{
    if (dia == 25 && mes == 12)
        return 1;
    else
        return 0;
}
```

Notar que el **else** no es indispensable, ya que si la comparación es verdadera, se saldrá de la función. Es decir que podría escribirse:

```
int Navidad (int dia, int mes)
{
    if (dia == 25 && mes == 12)
        return 1;
    return 0;
}
```

Esta función también podría escribirse:

```
int Navidad (int dia, int mes)
{
    if (dia != 25 || mes != 12)
        return 0;
    return 1;
}
```

Otro ejemplo sería implementar una división de números enteros positivos:

```
int Division (int a, int b)
{
    if (a >= 0 && b > 0)
        return a / b;
    printf ("Error");
    return 0; /* es obligatorio devolver un dato */
}
```

Notar que el **else** no es indispensable, ya que si la comparación es verdadera, se saldrá de la función. Es decir que podría escribirse:

```
int Navidad (int dia, int mes)
{
    if (dia == 25 && mes == 12)
        return 1;
    return 0;
}
```


Esta función también podría escribirse como sigue:

```
int Navidad (int dia, int mes)
{
    if (dia != 25 || mes != 12)
        return 0;
    return 1;
}
```

Otro ejemplo sería implementar una división de números enteros positivos:

```
int Division (int a, int b)
{
    if (a >= 0 && b > 0)
        return a / b;

    printf ("Error");
    return 0;    /* es obligatorio devolver un dato */
}
```

En C toda condición se evalúa por verdadero o por falso. Un valor igual a cero será falso, y todo valor distinto de cero será considerado verdadero. Veamos un ejemplo:

Ejemplo 3.5: Evaluación de condiciones

```
void imprimir (int a)
{
    if (a)
        printf ("a = %u", a);
}
```

La función imprimir imprimirá el valor de a sólo y sólo si a es distinta de cero.

Sentencias de control de programa (switch-case)

La sentencia **switch-case** es un **if** múltiple, donde una variable se compara contra múltiples constantes:

```
switch (variable)
{
    case constante_1: /* Ejecutar todas las sentencias hasta el break */
        sentencia;
        ...
        break;
    case constante_2:
        sentencia;
        ...
        break;
    ...
    case constante_n:
        sentencia;
        ...
}
```

```
        break;
    default:          /* Ejecutar si no se entra en las anteriores */
        sentencia;
        ...
}
```

esto sería equivalente a:

```
if (variable == constante_1)
{

}
else if (variable == constante_2)
{

}
...
else if (variable == constante_n)
{

}
else
{

}
```

Notar que cada case termina con una sentencia **break**; Este **break** es necesario para que no se continúe ejecutando el **case** inferior. Si no se lo escribiese, la ejecución continuaría con el case inferior, y así sucesivamente hasta encontrar una sentencia **break**. La sentencia **default**: se ejecutará si ninguno de los case anteriores se cumple. Veamos un ejemplo:

```
int Cantidad_de_dias_del_mes (int mes, int anio)
{
    int dias;

    switch (mes)
    {
        case 1:
        case 3:
        case 5:
        case 7:
        case 8:
        case 10:
        case 12:
            dias = 31;
            break;
        case 2:
            if (anio %4 == 0)
                dias = 29;
            else
                dias = 28;
            break;
        case 4:
        case 6:
        case 9:
        case 11:
            dias = 30;
```

```
        break;
    default:
        printf ("El mes especificado es invalido.\n");
        dias = 0;      /* Es obligatorio devolver un dato */
    }
    return dias;
}
```

Esta función podría escribirse también como:

```
int Cantidad_de_dias_del_mes (int mes, int anio)
{
    int dias;

    if (mes == 1 || mes == 3 || mes == 7 || mes == 8 || mes == 10 || mes == 12)
        dias = 31;
    else if (mes == 2)
    {
        if (anio%4 == 0)
            dias = 29;
        else
            dias = 28;
    }
    else if (mes == 4 || mes == 6 || mes == 9 || mes == 11)
        dias = 30;
    else
    {
        printf ("El mes especificado es invalido.\n");
        dias = 0;      /* Es obligatorio devolver un dato */
    }
    return dias;
}
```

Puede parecer más práctico utilizar *if-else*, en vez de *switch-case*. Sin embargo, en muchos casos resulta mucho más práctico e intuitivo esta última sentencia, por ejemplo cuando la aplicación debe responder a una gran cantidad de mensajes, de formas diferentes. Es común el uso por ejemplo al procesar mensajes de error. Por ejemplo:

```
switch (error)
{
    case 1:
        printf ("División por cero\n");
        break;
    case 2:
        printf ("Overflow\n");
        break;
    case 3:
        printf ("Parametros incorrectos\n");
        break;
    case 4:
        printf ("Error fatal. Debe terminarse el programa inmediatamente.\n");
        exit (1);
        /* exit() es una función definida en la librería stdlib.h, que interrumpe
el
        programa inmediatamente y retorna el control al sistema operativo */
    default:
        printf ("Error desconocido\n");
}
```

ifs anidados

Ejemplo 3.6: ifs anidados (1)

```
/* Devuelve 1 si el día indicado es Navidad, y 0 si no lo es */
int Navidad (int mes, int dia)
{
    if (mes == 12)
        if (dia == 25)
            return 1;
    return 0;
}
```

Debe tenerse mucho cuidado, y por lo general conviene poner llaves cuando la situación se vuelve confusa. Por ejemplo, la siguiente función es incorrecta:

```
int Navidad (int mes, int dia)
{
    int ret;

    if (mes == 12)
        if (dia == 25)
            ret = 1;
    else
        ret = 0;

    return ret;
}
```

ya que el else no corresponde al primer if, sino al segundo. En este caso, el segundo if debería haberse encerrado entre llaves.

También es posible escribir sentencias if como sentencia de un else.

Ejemplo 3.7: if como sentencia de un else (1)

```
/* Devuelve 1 si se trata de un día feriado y 0 en caso contrario */
int Feriado (int dia, int mes)
{
    int Ret;

    if (dia==25 && mes==5)
        Ret=1;
    else if (dia==9 && mes==7)
        Ret=1;
    else if (dia==12 && mes==10)
        Ret=1;
    else if (dia==25 && mes==12)
        Ret=1;
    else
        Ret=0;
}
```

```
    return Ret;
}
```

Este código es similar a un switch-case, pero por requerirse la evaluación de dos valores no se lo puede implementar de esta forma. Gramaticalmente lo correcto sería escribir este código debería escribirse como sigue:

```
int Feriado (int dia, int mes)
{
    int Ret;

    if (dia==25 && mes==5)
        Ret=1;
    else
        if (dia==9 && mes==7)
            Ret=1;
        else
            if (dia==12 && mes==10)
                Ret=1;
            else
                if (dia==25 && mes==12)
                    Ret=1;
                else
                    Ret=0;

    return Ret;
}
```

Sin embargo, lejos de resultar más claro, este código resulta más confuso, razón por la cual esta forma de anidar if-else no se utiliza.

Sentencias de control de programa (while y do-while)

La sentencia while ejecuta la sentencia o sentencias que están a continuación, si y sólo si se cumple la condición especificada. Un ejemplo de esto es el siguiente:

```
while (condición)
    sentencia;
```

De idéntica forma, puede utilizarse la dupla do-while si se quiere evaluar la condición al final del ciclo:

```
do
{
    sentencia;
    ...
    sentencia;
} while (condición);
```

Ambas sentencias funcionan en una forma similar. La única diferencia consiste en que while evalúa la condición al comenzar el ciclo, y por lo tanto el mismo puede no ejecutarse, y **do-while** al finalizar el mismo, y por lo tanto el mismo se ejecuta al menos una vez. Notar que **do-while** puede fácilmente implementarse con **while**.

Reescribamos ahora la función factorial, pero en forma iterativa.

Ejemplo 3.8: while

```
unsigned Factorial (unsigned n)
{
    int ret;

    ret = 1;

    while (n > 1)
    {
        ret = ret * n;
        n = n - 1;
    }
    return ret;
}
```

Sentencias de control de programa (for)

La sentencia **for** se usa para implementar ciclos controlados por un contador. La forma de esta sentencia es la siguiente:

```
for (sentencia_inicial; condición_de_continuidad; sentencia_de_paso)
    sentencia_del_lazo;
```

Un ejemplo de esto es:

Ejemplo 3.9: Ciclo for

```
for (a = 10; a < 20; a = a + 1)
{
    printf ("a = %d\n", a);
}
```

Veamos otro ejemplo. Escribamos una función que imprima la tabla de multiplicar.

Ejemplo 3.10: Ciclos for anidados

```
void ImprimirTabla (void)
{
    int i, j;

    for (i = 0; i <= 10; i = i + 1)
        for (j = 0; j < 10; j = j + 1)
            printf ("%d por %d es igual a %d\n", i, j, i * j);
}
```

En realidad la sentencia **for** es mucho más que una sentencia para implementar contadores. En realidad, la sentencia **for** como la siguiente:

```
for (sentencia_inicial; condicion_de_continuidad; sentencia_de_paso)
    sentencia_del_lazo;
```

puede traducirse como sigue:

```
sentencia_inicial;

while (condición_de_continuidad)
{
    sentencia_del_lazo;
    ...
    ...
    sentencia_de_paso;
}
```

Por lo cual, sería válido escribir cosas como:

```
for (a = 0; a < 10 && b > 30; a = a * 2)
{
}

}
```

Pero por claridad, cuando se requiere hacer cosas así, generalmente se usa un ciclo **while** y no **for**, ya que resulta confuso para el programador.

Los tres parámetros de la sentencia **for** son opcionales y pueden omitirse. Incluso puede escribirse más de una sentencia en cada uno de los parámetros, separadas por coma.

```
int i, j;
i = j = 0;
for (; i < 10; ++i, j+=2)
{
    .....printf ("El producto de %d x %d es %d\n", i, j, i * j);
}
```

Incluso pueden omitirse los tres parámetros de la sentencia **for**, escribiendo: **for (;).** En este caso, este código será equivalente a: **while (1).**

Sentencias **break** y **continue**

Las sentencias **break** y **continue** pueden utilizarse en cualquier ciclo de los anteriormente vistos: **while**, **do-while**, **for** y su objetivo es modificar el normal flujo del ciclo. La primera de ellas (**break**) interrumpe el flujo del ciclo, y el programa continuará su ejecución en la instrucción inmediatamente posterior. Veamos un ejemplo:

Ejemplo 3.11: **break**

```
#include <stdio.h>

void main(void)
```

```
{
    int i;
    for (i=0; i<10; i=i+1)
    {
        if (i == 4)
            break;
        printf ("El valor de i es %d\n", i);
    }
    printf ("Fin\n")
}
```

La salida del programa es la siguiente:

```
El valor de i es 0
El valor de i es 1
El valor de i es 2
El valor de i es 3
Fin
```

¿Qué sentido tiene el uso de la sentencia *break*? En este programa no tiene mucho sentido, ya que simplemente se podría haber comparado *i* con 5 en vez de 10. Un uso muy común de la sentencia *break* es interrumpir un ciclo cuando se produce un error en el programa.

La sentencia *continue*, salta ejecuta el siguiente ciclo del lazo, evaluando previamente la condición de continuidad. Escribamos por ejemplo un programa que imprima los números impares del 1 al 10.

Ejemplo 3.12: continue

```
#include <stdio.h>

void main(void)
{
    int i;
    for (i = 1; i < 10; i = i + 1)
    {
        if (i % 2 == 0)
            continue;
        printf ("El valor %d es impar\n", i);
    }
    printf ("Fin\n");
}
```

La salida del programa es la siguiente:

```
El valor 1 es impar
El valor 3 es impar
El valor 5 es impar
El valor 7 es impar
El valor 9 es impar
```


Nuevamente, este programa podría haberse escrito más fácilmente incrementando *i* de a 2. continue se utiliza en condiciones en las cuales deben saltarse algunos casos particulares y no periódicos, como podría ser imprimir los números no primos.

Sentencia goto

La sentencia de salto incondicional **goto**, permite realizar un salto en la ejecución del programa. Esto es contrario a la programación estructurada, y su uso puede llevar a códigos difíciles de leer. Por esta razón sólo debe ser utilizada en casos muy limitados.

La sintaxis es la siguiente:

```
goto label;
label:
```

Veamos un ejemplo:

Ejemplo 3.13: goto

```
int FuncionCompleja ()
{
    for (...) {
        while (...) {
            for (...) {
                ...
                if (b == 0)
                    goto error;
                r = a / b;
                ...
            }
        }
    }
    return r;

error:
    printf ("Se ha producido un error\n");
    return -1;
}
```

La sentencia **goto** puede resultar útil para interrumpir funciones sumamente complejas, como la de arriba, en donde se deben interrumpir varios ciclos anidados al producirse un error. Resulta más simple utilizar la sentencia **goto** que utilizar variables auxiliares para interrumpir todos los ciclos. Sin embargo, esto puede solucionarse reescribiendo las funciones correctamente. Por ejemplo, el programa anterior podría ser reescrito como sigue:

```
int Calculo ()
{
    for (...) {
        while(...) {
            for(...) {
                ...
            }
        }
    }
}
```

```
        if (b == 0)
            return -1;
        r = a / b;
        ...
    }
}
return r;
}

int FuncionCompleja()
{
    int r = Calculo();

    if (r < 0)
        printf ("Se ha producido un error\n");
    return r;
}
```

La recomendación de evitar el uso de la sentencia **goto**, no quiere decir que no existan casos en los que su uso simplifica un programa, pero es más común que programadores inexpertos la utilicen sin criterio resultando en programas complicados y difíciles de mantener, razón por la cual se recomienda que sólo sea utilizada por programadores con experiencia.

Operador ?

Existe una variante más a la función **if**, que puede escribirse como una única sentencia que devuelve un valor. Esta variante se escribe utilizando el operador **?**, y es similar a **if-else**, salvo por el hecho de que puede retornar un valor. Veamos un ejemplo:

Ejemplo 3.14: Operador ?

```
if (a > b)
    r = a;
else
    r = b;
```

Este ejemplo puede ser reescrito como sigue:

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

aunque generalmente se lo escribe en un único renglón, como sigue:

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

Si ya existe **if-else**, ¿qué sentido tiene esta segunda variante? La respuesta es que esta segunda variante retorna un valor, y por ende permite escribir códigos como:

```
max = a > b ? a : b;
```

ó

```
printf ("el máximo es:%u\n", a > b ? a : b );
```

Resumen

Se han visto aquí tres tipos de sentencias de control; los saltos condicionales, los incondicionales y los ciclos o lazos.

Los saltos condicionales, que se definen utilizando la palabra reservada **if** y **switch-case**, cuyas estructuras son:

```
if (condicion) sentencia;                y

switch (variable)
{
    case valor: sentencia;
    ...
    default: sentencia;
}
```

En el caso de la sentencia **switch**, debe escribirse una sentencia **break** al finalizar cada **case**, para evitar que se ejecute el siguiente **case**.

La sentencia puede ser reemplazada por una serie de sentencias entre llaves.

Los ciclos, pueden ser instrumentados utilizando las palabras reservadas **while**, **do-while** y **for**. En este caso la sintaxis es:

```
while (condición) sentencia;
do { sentencia; } while (condición);
for (sentencia inicial; condición; paso) sentencia;
```

Existen dos sentencias, **break** y **continue**, que permiten alterar la secuencia de ejecución de un ciclo.

Finalmente, aunque muy poco utilizada, existe la sentencia de salto incondicional, **goto**. La sintaxis es:

```
goto label;
label:
```

Capítulo IV - Asignaciones y operadores

Operadores

En se C definen los siguientes operadores, que pueden aplicarse a cualquier tipo de variable numérica, tanto enteros como reales:

=	Asignación
*	Multipliación
/	División
+	Suma
-	Resta

Se definen también, únicamente para los números enteros, los siguientes operadores

%	Módulo o resto de división entera.
>>	Corrimiento de bits hacia la derecha
<<	Corrimiento de bits hacia la izquierda
&	Y (and) binario
	Ó (or) binario
^	Ó excluyente (xor) binaria
&&	Y lógica
	Ó lógica

Operadores de comparación:

<	menor
>	mayor
<=	menor o igual
>=	mayor o igual (el símbolo => es incorrecto)
==	igual
!=	diferente

Como ya se dijo antes, las condiciones se evalúan en C por verdaderas o falsas. ¿qué ocurre entonces al evaluar una condición? Esta devuelve un valor de verdad o falsedad. Por ejemplo:

```
{
    if (a==3)
        printf ("a = 3");
}
```

La comparación `a==3` devolverá 1 si `a` es igual a 3 y 0 en caso contrario. Es decir que este código también podría haber sido escrito como sigue:

```
{
    int tmp;

    tmp = (a == 3);
    if (tmp)
        printf ("a = 3\n");
}
```

Esta forma de trabajar del C resulta muy útil, pero también algo confusa para quien recién comienza a programar. Veamos el siguiente ejemplo:

Ejemplo 4.1: Operadores y comparaciones

```
void Imprimir (int a, int b)
{
    if (a && b)
        printf ("a = %d    b = %d\n", a, b);
}
```

Imprimirá el contenido de a y b si y sólo si a y b son distintas de 0. ¿Qué pasaría si modificásemos la comparación como sigue?

Ejemplo 4.2: Operadores y comparaciones

```
void Imprimir (int a, int b)
{
    if (a & b)
        printf ("a = %d    b = %d\n", a, b);
}
```

Pués bien se imprimirían a y b si y sólo si a y b tienen algún bit en común en 1, ya que lo que se está haciendo es un "y" bit a bit, y este es el valor que se pasa al if.

El uso de los operadores binarios es muy común para indicar el estado de un proceso o de un registro, comúnmente llamados flags o banderas. Por ejemplo, es común que las funciones retornen un código de error almacenado en un entero. Veamos un ejemplo:

Ejemplo 4.3: Flags

```
/*
bit 0: 0 = OK          1 = Error
bit 1: 0 = Error del programa  1 = Error del Sistema operativo
bit 2:  0 = Reintentar      1 = Cancelar
bit 3:  0 = Continuar       1 = Interrumpir el programa
*/

#define ERROR          1  /* OK */
#define OS_ERROR       2  /* El programa produjo un error */
#define RETRY          4  /* Reintentar la operación que produjo el error */
#define END_PROGRAM    8  /* Error del programa*/

void ProcesarError (unsigned ErrCode)
{
    if ((ErrCode & ERROR) == 0)
        return;

    /* Reporto el causante del error */
    if (ErrCode & OS_ERROR)
        printf ("Error de sistema operativo\n");
    else
        printf ("Error del programa\n");
}
```

```
/* Reporto si se va a reintentar la operación o no */
if (ErrCode & RETRY)
{
    printf ("Se reintentará la operación\n");
    return;
}
else
    printf ("La operación no se intentará nuevamente\n");

/* Reporto si se va a reintentar la operación o no */
if (ErrCode & END_PROGRAM)
{
    printf ("Error fatal\n");
    exit (1);          /* Exit es una función que interrumpe el programa */
}
}
```

Esta función podría ser llamada por ejemplo como sigue:

```
ProcesarError (ERROR | RETRY);
```

Operadores y asignaciones

Como ya se vió, una asignación de variables se realiza utilizando el símbolo = (igual). Es decir que si yo quisiese incrementar el valor de una variable, debería efectuar una operación como:

```
var = var + 1;
```

Sin embargo, es muy común que a una variable se la modifique simplemente sumándole, restándole, multiplicándola, dividiéndola, etc, por un valor. Por esta razón, C permite utilizar una notación abreviada, que consiste en escribir la operación a realizar antes del símbolo igual. Por ejemplo, la operación anterior se escribiría:

```
var += 1;
```

En forma similar se pueden escribir:

```
var = var * 2;    como var *= 2;
var = var % 3;    como var %= 3;
```

Aún más, en C es muy común realizar operaciones donde a la variable sea incrementada o decrementada de a 1. Por esta razón, C provee una forma aún más simplificada de escribir esta operación (sólo válida para enteros), y define a tal fin los operadores ++ y --. Estos operadores incrementan o decrementan el valor de una variable de tipo entero. Por ejemplo, en el caso anterior podría hacerse:

```
++var;
```

ó

```
var++;
```

En forma similar también puede escribirse:

```
--var;
```

ó

```
var--;
```

para decrementar su valor en 1.

La única diferencia entre poner el operador antes o después es que, si el operador está antes, la variable se modifica en forma inmediata, en tanto que si el operador está al final, la variable se modifica luego de que se finalice la sentencia. Por ejemplo:

```
{
    int var;
    var = 10;
    printf ("var = %d\n", ++a);
}
```

imprimirá el valor 11, en tanto que

```
{
    int var;
    var = 10;
    printf ("var = %d\n", a++);
}
```

imprimirá el valor 10. En ambos casos, al finalizarse el llamado a **printf()**, el valor de la variable *var* será 11.

Notación entera

No siempre resulta práctico escribir números en base decimal. Es por ello que el C permite escribir números en otras bases diferentes de la decimal, estas son la base octal (8) y hexadecimal (16). Todo número que comience con un cero, seguido de un dígito decimal se entenderá que está escrito en base octal, en tanto que todo número que comience con dígito 0 seguido del carácter **x** (equis) se interpretará como hexadecimal.



Ejemplo 4.4: Notaciones no decimales

```
void main (void)
{
    int a, b, c;

    a = 10;          /* Cargo el valor 10 decimal */
    b = 010;         /* Cargo el valor 10 octal (8 decimal) */
}
```

```
c = 0x10;    /* Cargo el valor 10 hexadecimal (16 decimal) */  
printf ("a=%d\nb=%d\nc=%d\n", a, b, c);  
}
```

Este programa producirá la siguiente salida:

```
a = 10  
b = 8  
c = 16
```


Capítulo V - Definición de tipos y conversiones

Estructuras

El C soporta varias formas de definir nuevos tipos de datos, que permiten agrupar información en varias formas. La más conocida y utilizada es la estructura, que permite agrupar varias variables en un único grupo.

Una estructura se define mediante la palabra reservada **struct**, encerrando entre llaves el tipo de dato

```
struct nombre_estructura
{
    definición de variables
}; /* Notar el punto y coma al final */
```

Ejemplo:

```
struct Registro {
    int a;
    float b;
    unsigned int c;
    unsigned int d;
}; /* Notar el punto y coma al final de la llave */
```

Esto permite agrupar un conjunto de variables en una única **estructura**. Para crear una variable de este tipo, bastará con hacer referencia a esta estructura. Las variables o campos de esta estructura se referencian utilizando un punto.

Por ejemplo, podría utilizarse la estructura anterior en la siguiente forma:

```
void main (void)
{
    struct Registro Reg;

    Reg.a = -10;
    Reg.b = 20.3;
    Reg.c = 15;
    Reg.d = Registro.c * 2;
}
```

Es posible cargar una estructura con un valor inicial. Por ejemplo, en el caso anterior podría escribirse:

```
void main (void)
{
    struct Registro Reg = {-10, 20.3, 15, 15*2};
    struct Registro Reg2;

    Reg2 = Reg; /* Es posible copiar registros como cualquier otra variable */
}
```

No existe ninguna función que imprima el contenido de un registro. Para ello debe definirse una función que lo haga.

Enumeraciones

El tipo de dato enumeración consiste en un conjunto de constantes numéricas, llamadas enumeradores ('enumerators' en inglés). Dichas constantes pueden tomar cualquier valor arbitrario (entero). Si no se lo especifica, se entiende cero para el primer valor, y uno más que el anterior para los restantes. Por ejemplo:

```
enum DiasSemana
{
    DOMINGO,
    LUNES,
    MARTES,
    MIERCOLES,
    JUEVES,
    VIERNES,
    SABADO
};
```

Para definir una variable de este tipo deberá utilizarse:

```
enum DiasSemana Dia;
/* En C++ está permitido omitir la palabra enum al
   definir una variable, en C no */
Dia = LUNES;
```

En este caso, *DOMINGO* tendrá el valor 0, *LUNES* el 1 y así sucesivamente.

Es posible asignar un valor específico a cada elemento, incluso valores repetidos:

```
enum DiasSemana
{
    DOMINGO = 10,
    LUNES,      /* Se utilizará el valor del elemento de arriba + 1 */
    MARTES,
    MIERCOLES = 15,
    JUEVES,
    VIERNES,
    SABADO = 10 /* Repito un valor ya utilizado */
};
```

En este caso, tanto *SABADO* como *DOMINGO* tendrán el valor 10, *LUNES* tendrá el 11, *MARTES* el 12, *MIERCOLES* el 15, *JUEVES* el 16 y *VIERNES* 17.

Si bien una variable de tipo enum almacenará un entero no está permitido asignarle directamente valores de este tipo. Por ejemplo, el código:

```
Dia = 15;
```

sería incorrecto, aún cuando MIERCOLES tenga el valor 15.

Sí es posible realizar esta operación utilizando algo llamado *cast*, que se verá más adelante, en este mismo capítulo. Por ahora simplemente diremos que es posible realizar esta operación de la siguiente forma.

```
Dia = (enum DiasSemana) 15;
```

También es sintácticamente posible asignar un valor inválido, como en el siguiente ejemplo:

```
Dia = (enum DiasSemana) 4257;
```

pero en este caso los resultados no están definidos.

Uniones

Una unión es una agrupación de variables bajo una única denominación, pero en la cual todas ellas comparten una misma zona de memoria. Según como se la acceda, dicha zona de memoria será utilizada para almacenar un tipo de dato diferente. Veamos un ejemplo:

Ejemplo 5.1: Uniones

```
union MiUnion{
    char c;
    int n;
    float f;
    double d;
};

void main(void)
{
    MiUnion u;

    u.d = 10.0 / 3.0;

    printf ("u como double = %f\n", u.d);
    printf ("u como float = %f\n", u.f);
    printf ("u como int = %d\n", u.n);
    printf ("u como char = %d\n", u.c);
}
```

Este ejemplo es correcto. Sin embargo no debe leerse nunca una variable de una forma diferente a como fue escrita, por ejemplo, si *u* se escribe como *int* no debe leerse nunca como *double*. Las uniones son utilizadas principalmente para almacenar datos particulares de un tipo de dato más general. En C++ este problema se resuelve de una forma diferente, con el concepto de herencia que se verá más adelante, razón por la cual las uniones son rara vez utilizadas.

Para terminar con el tema, veamos un par de ejemplos, el los que se utilicen estructuras, enumeraciones y uniones:

Ejemplo 5.2: Estructuras complejas (1)

```
/* estructura para almacenar información sobre una figura,
   a dibujar en pantalla */
enum TipoFigura {CUADRADO, RECTANGULO,CIRCULO};

struct TPosicion
{
    double x0, y0;
};

struct TFigura{
    enum TipoFigura;
    unsigned Color;
    TPosicion Pos;
    union Datos
    {
        double Lado;
        struct Lados
        {
            double Ancho;
            double Alto;
        }
        double Radio;
    };
};
```

Ejemplo 5.3: Estructuras complejas (2)

```
enum TipoMisil {TIERRATIERRA, TIERRAAIRE, AIRETIERRA, AIREAIRE};
enum TipoDureza {DEBIL, MEDIA, FUERTE}

/* Estructura para almacenar los datos de un misil */
struct TMisil
{
    double x,y,z;    /* Posición del misil */
    double vx, vy, vz; /* Velocidad */
    enum TipoMisil;
    union Blanco
    {
        struct Avion
        {
            double x,y,z;
            double vx,vy,vz;
        };
        struct Edificio
        {
            double x,y;
            TipoDureza Dureza;
        }
    };
};
```

Esta estructura permite almacenar la información de un misil. En este caso, el blanco u objetivo del misil nunca será un avión y un edificio al mismo tiempo. Por ello se utiliza una unión para almacenar los datos del objetivo. Ambas estructuras utilizarán la misma área de memoria, y el programador deberá saber como tratarla.

Definición de nuevos nombres para tipos de datos

Pueden definirse tipos de datos o nuevos nombres para los ya existentes usando la palabra reservada **typedef**. Esta se utiliza de la siguiente forma:

```
typedef definición_del_tipo nuevo_nombre_del_tipo;
```

Por ejemplo, si se quisiese llamar **byte** al tipo de dato **unsigned char**, debería escribir:

```
typedef unsigned char byte;
```

También podría utilizar **typedef** para abreviar el nombre de una estructura. Por ejemplo, en ejemplo de la sección anterior, donde se definía el tipo de dato **struct Registro**, podría abreviar estas dos palabras en una sola escribiendo:

```
struct Registro {  
    int a;  
    float b;  
    unsigned int c;  
    unsigned int d;  
};  
typedef struct Registro TRegistro;
```

Estas líneas pueden escribirse en forma más resumida como sigue:

```
typedef struct Registro {  
    int a;  
    float b;  
    unsigned int c;  
    unsigned int d;  
} TRegistro;
```

con lo cual, para crear este tipo de dato podría escribir:

```
TRegistro Reg;
```

Convirtiendo tipos de datos (cast)

En C está permitido realizar conversiones de un tipo de dato a otro, sin mayor inconveniente. Por ejemplo, podría escribirse:

 **Ejemplo 5.4:** Conversión de tipos (cast) (1)

```
void main(void)
{
    int a;
    unsigned int b;

    a = 10;
    b = a;
}
```

¿Qué valor tomará *b*? La respuesta es simple, 10. ¿Pero que ocurriría en el caso siguiente?:

```
void main (void)
{
    int a;
    unsigned int b;

    a = -1;
    b = a;
}
```

Para responder esta pregunta es necesario analizar la representación de -1 en binario. Un valor entero -1 tiene la siguiente representación binaria, en un sistema de 16 bits: 11111111 11111111. Si tratamos de leer este valor en decimal, considerándolo un número positivo obtendremos en valor 65535, es decir que *b* recibirá este valor.

Para evitar problemas de este tipo, el compilador suele advertir con un **warning** cuando se realizan conversiones de datos no especificadas.

Pueden realizarse también operaciones entre números enteros y de punto flotante. En este caso el compilador se encarga de la conversión. Por ejemplo:

```
{
    float a;
    int b;

    a = 20 / 3;
    b = a;
}
```

En este caso *a* recibirá en valor 6.66666, y *b* este valor, pero con la parte decimal truncada a 0, es decir, 6.

La mayoría de los compiladores emiten mensajes de advertencia cuando se realizan operaciones entre datos de distinto tipo, y lo correcto es no hacer esto. Para realizar operaciones entre datos de distinto tipo, debe realizarse una conversión de tipo. Esto se hace encerrando entre paréntesis el tipo de dato al que se quiere convertir. Esto se conoce con el nombre de **cast**, y la operación de realizar esta conversión con el nombre de **casting**. Por ejemplo, en los casos anteriores se tendría:

```
{
    int a;
    unsigned int b;
```

```
float c;

a = -1;
b = (unsigned int) a;
c = (float) a;
}
```

Ejemplo 5.5: Casting

```
typedef unsigned char byte;

byte prom (byte a, byte b, byte c)
{
    byte result;

    result = (byte) (((unsigned)a + (unsigned)b + (unsigned)c) / 3);

    return result;
}
```

La conversión de tipos puede realizarse aún sin que se cambie el tipo de variable. Supongamos que se requiera calcular el promedio de 3 números enteros de 8 bits, sin signo, llamados **a**, **b** y **c**.

Desde ya que la función promedio retornará un entero de 8 bits, ya que el resultado estará acotado a los valores recibidos. Vimos anteriormente que este cálculo no lo podemos realizar en la forma: $a/3+b/3+c/3$, ya que al realizarse la división se trunca la parte decimal. Por ejemplo, el promedio de 2, 2, 2 sería 0.

La segunda alternativa consiste en realizar la suma primero, y luego la división $(a+b+c)/3$.

¿Pero qué ocurriría si la suma superase el número 255, que es el máximo entero representable con 8 bits?

Rta: El resultado de la suma se trugaría quedándose únicamente con los 8 bits menos significativos, y el resultado sería incorrecto.

¿Cómo solucionar este problema? La respuesta consiste en realizar este cálculo con números de mayor precisión. Veamos el ejemplo:

```
typedef unsigned char byte;

byte prom (byte a, byte b, byte c)
{
    byte result;

    result = (byte) (((unsigned)a + (unsigned)b + (unsigned)c) / 3);

    return result;
}
```

Notar la forma en que se realizó la operación. En primer lugar se convirtió cada una de las tres variables a un entero de mayor precisión, luego se efectuó la suma de las tres y la división por 3. Finalmente se convirtió el resultado en un entero de 8 bits y se lo copió en la variable *result*.

Capítulo VI - Vectores

Los vectores (arrays) en C se definen encerrando entre corchetes la cantidad de elementos que el mismo almacenará, a continuación del nombre del mismo.

Por ejemplo, un vector de 10 elementos de tipo entero se definiría como:

```
int vect[10];
```

y cada uno de estos 10 elementos se direccionará escribiendo

```
vect[i]
```

con $0 \leq i \leq 9$.

Todo vector de n elementos tendrá un índice que irá entre 0 y n-1.

Es importante destacar que el lenguaje C no realiza ningún tipo de verificación del índice con el cual se direcciona el vector. Si se utilizase un índice que excediese el tamaño del arreglo, pueden producirse las siguientes situaciones:

- Se lee basura de la memoria.
- Se sobrescriben otros datos del programa -> Esto conducirá a que en otras partes de programa, cuando se requieran los datos sobrescritos, se trabaje con basura.
- Se sobrescribe el programa.
Si no se trabaja con un sistema operativo en modo protegido (ej: D.O.S.) los resultados son impredecibles. Generalmente, cuando se cuando se intente ejecutar la parte del programa que fue sobrescrita el programa producirá un error y se "colgará" el programa y la computadora.
En un sistema operativo que trabaje en modo protegido (Windows, UNIX), el mismo impedirá que se efectúe la modificación del programa, y se producirá algo conocido como interrupción o excepción, que se verá más adelante. El resultado es la interrupción del programa (salvo que se especifique lo contrario). En Windows se emitirá un mensaje "el programa ha efectuado una operación no válida y se cerrará". En UNIX se produce un mensaje conocido como coredump.
- Se intenta acceder (leer o escribir) memoria que no pertenece al programa. En un sistema operativo que no funcione en modo protegido, la operación se llevará a cabo y los resultados son impredecibles. En un sistema operativo que funcione en modo protegido se cerrará la aplicación (salvo que se especifique lo contrario) y se notificará al usuario. En Windows se imprimirá un cartel "Fallo de protección general" ó ("General Protection Fault"). En UNIX el error es conocido como coredump.

Es por esto que debe tenerse mucho cuidado al trabajar con vectores o, como se verá más adelante, con punteros. Es deber del programador verificar todas las condiciones que pueden llevar a cometer este error, y el lenguaje C no ofrece absolutamente ningún tipo de protección ni validación.

Veamos un par de ejemplos con vectores:

Ejemplo 6.1: Vectores

```
void main (void)
{
    int vec[10]; /* Declaro un vector de 10 elementos de tipo entero */
    int i;

    for (i=0;i<10;++i)
        vec[i] = i*2; /* Cargo el vector */

    for (i=0;i<10;++i)
        printf ("La posición %d del vector contiene el numero %d\n", i, vec[i]);
}
```

Este ejemplo imprimirá la siguiente salida:

```
La posición 0 del vector contiene el numero 0
La posición 1 del vector contiene el numero 2
La posición 2 del vector contiene el numero 4
La posición 3 del vector contiene el numero 6
La posición 4 del vector contiene el numero 8
La posición 5 del vector contiene el numero 10
La posición 6 del vector contiene el numero 12
La posición 7 del vector contiene el numero 14
La posición 8 del vector contiene el numero 16
La posición 9 del vector contiene el numero 18
```

Notar que ya que el vector tiene 10 elementos, el índice va de 0 a 9.

Si se declaran vectores muy grandes se puede exceder el tamaño de la pila, con lo cual la computadora se puede colgar. Por ejemplo, DOS no funcionará correctamente si se declaran cosas como:

```
long double Vect[50000];
```

En DOS y Windows 16 bits, el tamaño de la pila es de algunos KBytes en tanto que en sistemas operativos de 32 bits (Windows 95/98/NT, o UNIX actuales) es de algunos MBytes. Algunos compiladores permiten verificar esto activando la opción "Test stack overflow".

El C no define ningún tipo de dato para almacenar string, como en otros lenguajes. Por el contrario, los strings se almacenan en cadenas de caracteres o vectores (arrays) terminados en el byte 0, o lo que es lo mismo, el carácter '\0'. Por ejemplo, si quisiésemos almacenar la palabra 'HOLA' necesitaremos 5 posiciones del vector. De esta forma, todo string se almacenará en un vector de caracteres, terminado en el byte 0.

Debe recordarse sin embargo que la palabra reservada `char` no debe entenderse como "caracter" sino como entero (con o sin signo) de 8 bits, si bien en la práctica se la usa para almacenar caracteres.

Puede utilizarse la opción `%s` de la función `printf` para imprimir strings.

Ejemplo 6.2: Impresión de strings

```
#include <stdio.h>

void main (void)
{
    char Str[10];

    Str[0] = 'H';
    Str[1] = 'O';
    Str[2] = 'L';
    Str[3] = 'A';
    Str[4] = 0;

    printf ("La cadena Str contiene '%s'\n", Str);
}
```

La salida de este programa sería:

La cadena Str contiene 'Hola'

Al igual que cualquier variable, es posible darle a un vector un valor inicial:

Ejemplo 6.3: Valores iniciales en los vectores

```
#include <stdio.h>

void main (void)
{
    char Str[10] = "HOLA";

    printf ("La cadena Str contiene %s\n", Str);
}
```

Sin embargo no es posible realizar copia de vectores enteros. Sólo está permitido acceder a los elementos individuales de un vector. Es decir, no pueden efectuarse operaciones tales como:

```
void main (void)
{
    char Str[10] = "HOLA";

    char Vec[10];
```

```
Vec = Str;    /* Esto es incorrecto */  
}
```

Escribamos entonces el código necesario para copiar un vector de caracteres en otro.

Ejemplo 6.4: Copiando strings

```
/* Este programa es incorrecto */  
  
#include <stdio.h>  
  
void main (void)  
{  
    char Str[10] = "HOLA";  
    char Nuevo[10];  
    int i;  
  
    /* Copio Str en Nuevo */  
    i = 0;  
  
    while (Str[i] != 0)  
    {  
        /* Copio uno a uno los caracteres, hasta encontrar el 0 */  
        Nuevo[i] = Str[i];  
        ++i;  
    }  
    printf ("La cadena Nuevo contiene %s\n", Nuevo);  
}
```

¿qué está mal en el programa anterior? Pues bien, toda cadena de caracteres debe terminar en el byte 0, sin embargo en el ejemplo anterior el 0 no sería copiado. Es un error pensar que una variable no inicializada contendrá ceros. Esto sólo es válido para variables globales, salvo que se indique lo contrario. Veamos entonces nuestro ejemplo corregido.

Ejemplo 6.5: Copiando strings

```
#include <stdio.h>  
  
void main (void)  
{  
    char Str[10] = "HOLA";  
    char Nuevo[10];  
    int i;  
  
    /* Copio Str en Nuevo */  
    i = 0;  
  
    do{  
        Nuevo[i] = Str[i];  
        ++i;  
    }while (Str[i-1] != 0);  
    /* Copio uno a uno los caracteres, hasta encontrar el 0 */
```

```
printf ("La cadena Nuevo contiene %s\n", Nuevo);  
}
```

La salida de este programa será:

La cadena Nuevo contiene HOLA

Pasaje de vectores a funciones

Nada impide pasar un vector a una función, como cualquier otro tipo de dato.

Veamos otro ejemplo de strings. Escribamos una función capaz de separar las palabras que componen una oración

Ejemplo 6.6: Recorriendo un string

```
#include <stdio.h>  
  
void SepararPalabras (char Entrada[100])  
{  
    int Inicio; /* Señalará la posición de comienzo de una palabra */  
    int Pos;    /* Usaré esta variable para ir recorriendo el vector */  
  
    Pos = Inicio = 0; /* Comienzo en el principio del vector */  
  
    do{  
        /* Si encuentro un caracter espacio punto o fin de texto, llegué al  
        final de una palabra */  
        if (Entrada[Pos] == ' ' || Entrada[Pos] == '.' || Entrada[Pos] == 0)  
        {  
            /* Me fijo que la palabra no empiece y termine en la misma posición, lo  
            que ocurriría si el string estuviese vacío o si hubiesen varios  
            espacios juntos */  
            if (Inicio != Pos)  
            {  
                /* Imprimo uno a uno los caracteres de la palabra */  
                while (Inicio < Pos)  
                {  
                    printf ("%c", Entrada[Inicio]);  
                    ++Inicio;  
                }  
                /* Salto al siguiente renglón */  
                printf ("\n");  
            }  
            /* Señalo el comienzo de la siguiente palabra */  
            Inicio = Pos+1;  
        }  
        ++Pos;  
    }while (Entrada[Pos-1] != 0);  
}  
  
void main(void)
```

```
{
    char Texto[100] = "el gato esta arriba de la mesa.";

    printf ("Las palabras que componen la oracion '%s' son:\n", Texto);
    SepararPalabras (Texto);
}
```

El programa imprimirá:

```
Las palabras que componen la oracion 'el gato esta arriba de la mesa' son:
el
gato
esta
arriba
de
la
mesa
```

En la función `SepararPalabras`, utilizo dos variables de tipo entero, llamadas `Inicio` y `Pos`. La primera de las variables contendrá la posición de comienzo de la palabra, en tanto que la segunda se utiliza para ir recorriendo la palabra, hasta encontrar el fin de la misma.

Notar que cuando se encuentra el final de una palabra, se la imprime carácter a carácter, y al final se agrega el salto de línea. Este programa muestra claramente la forma en que se guardan las cadenas de caracteres. En C los string son simplemente vectores que contienen bytes, es decir, el código ascii de cada letra, terminadas con el byte 0.

La función `printf()` provee la forma de imprimir strings, utilizando `%s`.

El lenguaje C provee una librería para manejo de string, llamada `string.h`. En ella se definen entre otras las siguientes funciones:

```
strcmp (s1, s2);
```

compara los string `s1` con `s2`. Devuelve 0 si son iguales, o -1 o +1 si son diferentes (-1 si `s1` está alfabéticamente antes de `s2`, y 1 en caso contrario).

```
strcpy (s1, s2);
```

Copia en string `s2` en `s1`.

```
strcat (s1, s2);
```

Copia (concatena) el string `s2` al final de `s1`.

Más sobre el pasaje de vectores a funciones

Veamos un ejemplo de un programa en C:

Ejemplo 6.7: Pasaje de vectores a funciones

```
#include <stdio.h>

void Funcion (char s[10])
{
    s[0] = 'C';
    s[1] = 'h';
    s[2] = 'a';
    s[3] = 'u';
    s[4] = '\0';
}

void main(void)
{
    char Str[10] = "Hola";

    printf ("%s\n", Str);
    Funcion (Str);
    printf ("%s\n", Str);
}
```

Este programa producirá la siguiente salida:

```
Hola
Chau
```

En C todo el pasaje de parámetros a las funciones se realiza por valor, es decir, no existe el pasaje por referencia. Veamos ahora el programa:

Al comenzar el programa, defino una variable Str, local a la función main().

A continuación llamo a la función Funcion(), que carga el texto "Chau" en la variable s, local a la función Funcion(), y que es una copia de la variable Str de la función main().

El siguiente printf() muestra que la variable Str contiene el texto Chau.

Pero si sólo existe el pasaje por valor, ¿cómo es posible que esta función haya modificado esta variable?

La respuesta a esta pregunta exige presentar un nuevo tema; los punteros, que se verá más adelante. Sin embargo vamos a dar una respuesta rápida sin entrar en detalles, ya que el análisis exhaustivo de este tema se verá en el capítulo punteros.

La variable s no contiene en realidad un vector, sino la dirección de memoria donde se almacena dicho vector.

Es por ello que cuando se llama a una función, no se pasa una copia del vector, sino una copia de la posición de memoria donde se encuentra dicho vector, y por lo tanto cualquier modificación que se realice sobre el vector se hará sobre el vector original.

Viéndolo desde otro punto de vista, se podría decir que el pasaje de vectores a funciones "se comporta en forma idéntica" al pasaje de parámetros por referencia.

Ninguna función puede devolver un vector. La devolución de vectores no está soportada por C, pero se puede hacer algo parecido, lo cual se verá en el tema punteros.

Existe una biblioteca de funciones standard para trabajar con strings, declarada en el archivo string.h. Veamos un ejemplo:

Ejemplo 6.8: función strcpy ()

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

void Funcion (char s[])
{
    strcpy (s, "Chau");
}

void main(void)
{
    char Str[10] = "Hola";

    printf ("%s\n", Str);
    Funcion (Str);
    printf ("%s\n", Str);
}
```

Este programa producirá la siguiente salida:

```
Hola
Chau
```

Notar que en la función Función, no se especificó el tamaño del vector s. Esto tiene sentido, ya que la función Funcion no recibe una copia de los elementos, sino tan sólo de la dirección en memoria de los mismos. Tampoco se realizará ningún chequeo de validez de los índices, razón por la cual esta función sólo necesita saber que recibirá como parámetro un vector, pero no le interesa la cantidad de elementos que el mismo contenga.

La función strcpy() copia un string en otro. Desde ya que el C no verifica que el string destino sea suficientemente grande. Es decir algo como:

```
char s[5];
strcpy (s, "Estoy tratando de cargar un string muy grande en un string muy chico.");
```

sería un error muy grave, que llevaría a que el programa no funcione correctamente, pero ni el compilador ni el programa emitirán ningún mensaje de error ni de advertencia.

Otras funciones definidas en esta biblioteca son:

strcmp()

compara dos strings. Devuelve 0 si son iguales y -1 ó +1 si son diferentes (-1 si el primer string es alfabéticamente anterior al segundo y +1 si no lo es).

strcat()

Concatena dos strings, es decir, copia el segundo string a continuación del primero.

toupper ()

Devuelve el caracter en mayúsculas.

tolower ()

Devuelve el caracter en minúsculas.

en stdio.h se definen también:

atoi()

Convierte un string en un entero.

atof()

Convierte un string en un número de punto flotante.

Capítulo VII - Matrices

Las matrices son similares a los vectores. Pueden definirse matrices n-dimensionales, en forma similar a una matriz bidimensional, pudiendo las mismas contener cualquier tipo de datos. Por ejemplo, una matriz de enteros de 2x3 se define como:

```
int Mat[2][3];
```

Al igual que con cualquier tipo de variable, es posible cargar valores iniciales en los vectores y matrices, encerrando cada vector entre llaves. Por ejemplo, podría darse valores iniciales a la matriz anterior en la forma que sigue:

```
int Mat[2][3] = {{0, 1, 2}, {3, 4, 5}};
```

En este ejemplo, los índices de la matriz irán de [0-1][0-2], pero como ya vimos, el lenguaje C no realiza ningún tipo de verificación, por lo que nada nos impediría hacer referencia al elemento `Mat[0][5]`. En este caso esto no es un error, como podría suponerse, ya que este elemento no está fuera de la matriz sino que se tratará del elemento `Mat[1][3]`. Los elementos de una matriz se ordenan en forma contigua en memoria, uno detrás de otro, es decir que, en este ejemplo, el orden sería:

```
[0][0] ; [0][1] ; [0][2] ; [1][0] ; [1][1] ; [1][2]
```

Notar que en este ejemplo, el número almacenado en cada posición de la matriz es el orden en el cual se almacenarán en la memoria.

Una matriz n-dimensional se definiría como sigue:

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

Recordar que los saltos de línea son opcionales, se utilizan sólo para dar mayor claridad al texto.

Veamos ahora el ejemplo de un programa traductor. Nota: La función *TraducirOración()* es una variación de *SepararPalabras()*, vista en el capítulo anterior, razón por la cual no se la analizará nuevamente.

Ejemplo 7.1: Matrices - programa traductor

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

void TraducirPalabra (char Palabra[])
{
    int i;

    char Diccionario [20][10] = {
        "el",      "the",
        "gato",     "cat",
        "perro",    "dog",
        "esta",     "is",
```

```
    "arriba",    "on",
    "la",        "the",
    "de",        "",
    "un",        "a",
    "mesa",      "table",
    "silla",     "chair"};

for (i = 0; i < 20; i += 2)
{
    if (strcmp (Diccionario[i], Palabra) == 0)
    {
        printf (Diccionario[i+1]);
        printf (" ");
        return;
    }
}
printf ("%s? ", Palabra);
}

void TraducirOracion (char Entrada[])
{
    int Inicio;
    int Pos;
    int PosPalabra;
    char Palabra[10];

    Pos = Inicio = 0;

    do {
        if (Entrada[Pos] == ' ' || Entrada[Pos] == '.' || Entrada[Pos] == 0)
        {
            if (Inicio != Pos)
            {
                {
                    PosPalabra = 0;
                    while (Inicio < Pos)
                    {
                        Palabra [PosPalabra] = Entrada[Inicio];
                        ++PosPalabra;
                        ++Inicio;
                    }
                    Palabra[PosPalabra] = 0;
                    TraducirPalabra (Palabra);
                }
                Inicio = Pos+1;
            }
            ++Pos;
        }while (Entrada[Pos-1] != 0);
    }

void main(void)
{
    char Texto[100] = "el gato esta arriba de la mesa.";

    printf ("La traduccion de la oracion '%s' es:\n", Texto);
    TraducirOracion (Texto);
    printf("\n");
}
```

La salida del programa será la siguiente:

La traducción de la oración 'el gato esta arriba de la mesa.' es:
the cat is on the table

Notar que en este ejemplo, en la función traducir se define una matriz Diccionario, que mediante el segundo índice recorrerá los distintos caracteres, y con el primero las palabras. También podría pensársela como un vector que en cada posición contiene un string, (que no son otra cosa que vectores de caracteres). En las posiciones pares del primer índice estará la palabra en castellano, y en las posiciones impares las palabras en inglés. La función **strcmp()** está definida en string.h, y sirve para comparar strings, es decir, cadenas de caracteres terminadas en 0. Esta función devuelve 0 si las palabras son iguales, y -1/+1 si son diferentes (-1 si la primera palabra es alfabéticamente anterior a la segunda y +1 en caso contrario).

Podría haberse definido una matriz 3-dimensional en la forma siguiente:

```
void TraducirPalabra (char Palabra[10])
{
    int i;

    char Diccionario [10][2][10] = {
        {"el",      "the"},
        {"gato",    "cat"},
        {"perro",   "dog"},
        {"esta",    "is"},
        {"arriba",  "on"},
        {"la",      "the"},
        {"de",      ""},
        {"un",      "a"},
        {"mesa",    "table"},
        {"silla",   "chair"}};

    for (i = 0; i < 10; ++i)
    {
        if (strcmp (Diccionario[i][0], Palabra) == 0)
        {
            printf (Diccionario[i][1]);
            printf (" ");
            return;
        }
    }
    printf ("%s?", Palabra);
}
```

Pasaje de matrices a funciones

El pasaje de matrices a funciones es idéntico al pasaje de vectores. Incluso en el ejemplo anterior podríamos haber pasado el diccionario como parámetro:

Ejemplo 7.2: Matrices - programa traductor

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

void Traducir (char Palabra[10], char Diccionario[][2][10])
{
```

```
int i;

for (i = 0; i < 10; ++i)
{
    if (strcmp (Diccionario[i][0], Palabra) == 0)
    {
        printf (Diccionario[i][1]);
        printf (" ");
        return;
    }
}
printf ("%s? ", Palabra);
}

void TraducirOracion (char Entrada[], char Diccionario[][2][10])
{
    int Inicio;
    int Pos;
    int PosPalabra;
    char Palabra[10];

    Pos = Inicio = 0;

    do{
        if (Entrada[Pos] == ' ' || Entrada[Pos] == '.' || Entrada[Pos] == 0)
        {
            if (Inicio != Pos)
            {
                PosPalabra = 0;
                while (Inicio < Pos)
                {
                    Palabra [PosPalabra] = Entrada[Inicio];
                    ++PosPalabra;
                    ++Inicio;
                }
                Palabra[PosPalabra] = 0;
                TraducirPalabra (Palabra, Diccionario);
            }
            Inicio = Pos+1;
        }
        ++Pos;
    }while (Entrada[Pos-1] != 0);
}

void main(void)
{
    char Texto[100] = "el gato esta arriba de la mesa.";
    char DicCastellanoIngles [10][2][10] = {
        {"el",      "the"},
        {"gato",    "cat"},
        {"perro",   "dog"},
        {"esta",    "is"},
        {"arriba",  "on"},
        {"la",      "the"},
        {"de",      ""},
        {"un",      "a"},
        {"mesa",    "table"},
        {"silla",   "chair"}};
```

```
printf ("La traduccion de la oracion '%s' es:\n", Texto);  
TraducirOracion (Texto, DicCastellanoIngles);  
printf("\n");  
}
```

Notar que al especificar los parámetros que recibe la función, no se especifica el primer tamaño pero sí los restantes. En C es obligatorio especificar las dimensiones de cada uno de los vectores de un array n-dimensional, con excepción del primero.

Capítulo VIII - Punteros

Introducción

Un puntero es un tipo de variable como cualquier otra, cuyo contenido es una dirección de memoria. El concepto en sí es muy simple, y su uso es igual a los demás tipos de variables del lenguaje C. Sin embargo el uso de punteros suele resultar algo complicado y confuso para quién recién comienza a programar en lenguaje C, y es un tipo de dato que no está presente en otros lenguajes de programación y por lo tanto puede resultar nuevo.

¿Qué es un puntero?



Un puntero es simplemente una variable cuyo contenido es una dirección de memoria.

En la práctica, los punteros suelen usarse para direccionar un tipo de dato en particular. Por esta razón, y también para saber como direccionar el dato, en C se suele asociar al puntero el tipo de dato que se va a direccionar.

Un puntero se define simplemente anteponiendo un * (asterisco) al nombre de la variable, en su definición (Ej: `int *a;`).

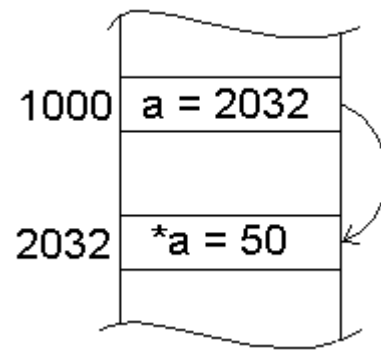
Es importante recordar que en C, ninguna variable es inicializada por defecto, salvo que se indique lo contrario. De la misma forma, un puntero, al ser creado, contendrá una dirección de memoria cualquiera. Si se esta programando en una computadora sin modo protegido (D.O.S., por ejemplo), toda la memoria es direccionable, es decir que se puede leer/sobreescribir toda la memoria de la PC, incluyendo el sistema operativo. En modo protegido, direccionar una posición de memoria inválida producirá un mensaje de error, conocido en Windows como '**General Protection Fault**'.

Veamos un par de ejemplos:

```
int *a;
```

a es una variable que almacenará una dirección de memoria, en tanto que ***a** será el entero contenido de dicha dirección de memoria.

Por ejemplo, supongamos que a tenga el valor de la dirección de memoria 2032, y ***a** tenga el valor 50:



El operador &, antepuesto a una variable devuelve la dirección de dicha variable, y se lo llama operador de indirección. En nuestro ejemplo tendremos:

Variable	Tipo de variable	Tipo de contenido	contenido
puntero a entero	(int *)	dirección de memoria	2032
a	entero(int)	entero	50
a	puntero a un puntero a entero (int **)	dirección de memoria	1000

Veamos otro ejemplo:

Ejemplo 8.1: Punteros

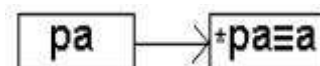
```

01  #include <stdio.h>
02
03  void main(void)
04  {
05      /* Definición de las variables */
06      int a;
07      int *pa;
08
09      a = 4;
10      pa = &a;
11      *pa = 2;
12
13      printf ("%d", a);
14  }
```

Este programa imprimirá el valor 2

En este ejemplo, **pa** es un puntero a una variable de tipo entero, es decir, una variable que contendrá una dirección de memoria en donde se almacenará un entero.

En la línea 6 se define una variable **a** que guardará una variable de tipo entero, y a la cual le asigno el valor 4, en la línea 9 del código. **&a** será la dirección de dicha variable. Así, lo que estoy haciendo en la línea 10 del ejemplo es cargar en **pa** la dirección de la variable **a**. Lo que se tiene en este momento es lo siguiente:

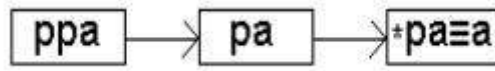


Cuando a continuación hago referencia a ***pa**, me estoy refiriendo a la dirección apuntada por **pa**, que en este caso será **a**.

¿Puedo preguntar ahora que es **&pa**? Pues bien, es la dirección de memoria donde está almacenada la variable **pa**, que a su vez es la dirección de memoria de un entero, es decir, **&pa** es un puntero a un puntero a una variable de tipo entero.

En este caso podría escribirse:

```
int **ppa;  
ppa = &pa;
```



es decir que pueden definirse punteros a punteros a un tipo de variable, y así sucesivamente, aunque en la práctica sólo se usan a lo sumo doble punteros.

Es común utilizar un caracter 'p' como primer letra del nombre de una variable de tipo puntero, para tener en claro que se trata de un puntero. De idéntica forma, se suele usar pp al trabajar con doble punteros, y así sucesivamente.

¿Tendría sentido preguntarse por **&&a**?

Pues no, de ninguna manera. **a** es una variable que está almacenada en una posición de memoria, conocida por el programa. Con **&a** puedo conocer dicha posición, pero **&&a** no tiene ningún sentido.

Veamos un ejemplo de una función que intercambia el contenido de dos variables de tipo entero. Llamaremos a esta función swap.

Primer ejemplo con punteros

Ejemplo 8.2: Función swap (versión incorrecta)

```
/* Este ejemplo no funciona correctamente */  
void swap (int a, int b);  
  
void main (void)  
{  
    int a, b;  
  
    swap (a, b);  
}  
  
/* Función que intercambia los dos enteros */  
void swap (int a, int b)  
{  
    int c;  
  
    c = a;  
    a = b;  
    b = c;
```

```
}
```

¿Funcionará este ejemplo correctamente? Claramente no, ya que la función `swap()` recibirá una **copia** de las variables `a` y `b`, las cuales locales a esta función, y que serán efectivamente intercambiadas. Pero las variables originales no serán modificadas.

Para poder intercambiar el contenido de estas dos variables en la función que la llamó, deberé pasar las direcciones de las mismas, de tal forma que la función pueda acceder directamente a las variables usadas por quien la llamó.



Toda función recibe como parámetros copias de los datos especificados (pasaje por valor), nunca datos mismos (pasaje por referencia).



Ejemplo 8.3: Función swap (versión correcta)

```
void swap (int *a, int *b);

void main (void)
{
    int a, b;

    swap (&a, &b);
}

/* Función que intercambia los dos enteros */
void swap (int *a, int *b)
{
    int c;

    c = *a;
    *a = *b;
    *b = c;
}
```

Nota de sintaxis

A veces se ve en algunas definiciones de punteros, el `*` del lado de la definición del tipo de dato en vez del de la variable. Por ejemplo:

```
int* a;
```

en vez de

```
int *a;
```

Esto no es un error de programación, pero si puede llevar a errores de comprensión, por ejemplo en el siguiente caso:

```
int* a, b;
```

En este caso, **a** será un puntero, pero **b** no, ya que el compilador lo interpreta como:

```
int *a, b;
```

que es la notación más adecuada..

Por otro lado, cuando una función devuelve un puntero, esto puede escribirse como:

```
int* funcion();
```

ó

```
int *funcion();
```

En este caso, si bien ambas declaraciones son correctas, la segunda parecería ser un puntero a una función que devuelve un entero (un tema que se verá más adelante), por lo que la sintaxis más adecuada es la primera.

Veamos un par de ejemplos más para comprender el tema de punteros.

Ejemplo 8.4: Punteros

```
/* Este programa es incorrecto */
void main(void)
{
    int *pa;

    *pa = 5;
}
```

En la función *main()* defino una variable llamada **pa**. A continuación cargo el valor 5 en la posición "apuntada" por dicha variable.

El programa es sintácticamente correcto. Entonces ¿qué está mal en este programa?

Rta: El valor de **pa** no fue nunca definido, o dicho de otra forma, la variable **pa** no fue inicializada. A este error se lo conoce como "puntero no inicializado". Todo puntero no inicializado tendrá un valor impredecible, y por lo tanto el valor 5 se estará cargando en una posición cualquiera de la memoria.

Escribamos ahora una función que devuelva el máximo y el mínimo de dos números:

Ejemplo 8.5: Punteros

```
void MinMax (int a, int b, int *min, int *max)
{
    if (a < b)
    {
        *min = a;
        *max = b;
    }
}
```

```
    }
    else
    {
        *min = a;
        *max = b;
    }
}

void main(void)
{
    int a, b, m, M;

    a = 5;
    b = 2;

    MinMax (a, b, &m, &M);

    printf ("El máximo entre %d y %d es %d\n", a, b, M);
}
```

La salida de este programa será:

El máximo entre 5 y 2 es 5

Más acerca del pasaje de variables a funciones

Veamos el siguiente ejemplo:

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

int DevolverEntero (void)
{
    int n;
    printf ("Ingrese el numero:\n");
    scanf ("%d", &n);

    return n;
}

void CargarEntero (int *n)
{
    printf ("Ingrese el numero:\n");
    scanf ("%d", n);
}

void MostrarEntero (int n)
{
    printf ("Numero = %d\n", n);
}

void main (void)
{
    int i,j;

    printf ("Repaso de punteros\n");
}
```

```
i = DevolverEntero ();
CargarEntero (&j);

MostrarEntero (i);
MostrarEntero (j);
}
```

Notar las dos funciones que se definieron para cargar enteros. La primera de ellas (*DevolverEntero()*), más simple de comprender, solicita al usuario el ingreso de un entero. La segunda de las funciones (*CargarEntero()*) recibe un puntero a la posición de memoria donde debe cargar el entero ingresado por el usuario.

La función *scanf()* está definida en *stdio.h*, y su funcionamiento es similar y opuesto al de *printf()*. Esta solicita al usuario el ingreso de un parámetro por la entrada standard (generalmente teclado). Utilizar esta función con un único parámetro. La principal diferencia entre *printf()* y *scanf()* en la forma en que son usadas, es que la primera no requiere modificar los parámetros recibidos, y por lo tanto recibe copias de los mismos, en tanto que la segunda sí debe hacerlo, por lo que recibe el puntero a los datos.

Finalmente la función *MostrarEntero()* es una función como cualquier otra. La misma recibe una **copia** del entero, y lo muestra en pantalla. Podría escribirse una versión similar que reciba un puntero al entero a mostrar.

¿Todo puntero debe estar inicializado?

Rta: Si bien no es obligatorio, todo puntero debe necesariamente ser inicializado para poder ser utilizado. En caso contrario se producirá un error en tiempo de ejecución al acceder a la posición especificada. Pero esto no obliga a inicializar todos los punteros de un programa. Los punteros son utilizados con fines muy variados, y es común que en muchos valores estos no contengan ningún valor válido. Para dejar el claro que el puntero no contiene ningún valor útil, se puede cargar en los mismos el valor 0, al que se llama habitualmente puntero nulo o **NULL**. Se supone que la dirección de memoria se reserva para indicar una posición de memoria inválida. Ej:

```
int *pa = NULL; /* Creo el puntero pa y le cargo el valor NULL, definido en
stdio.h como 0 */
```

Veamos otro ejemplo: Escriba un programa que cargue el valor NULL en un puntero a entero, utilizando una función:

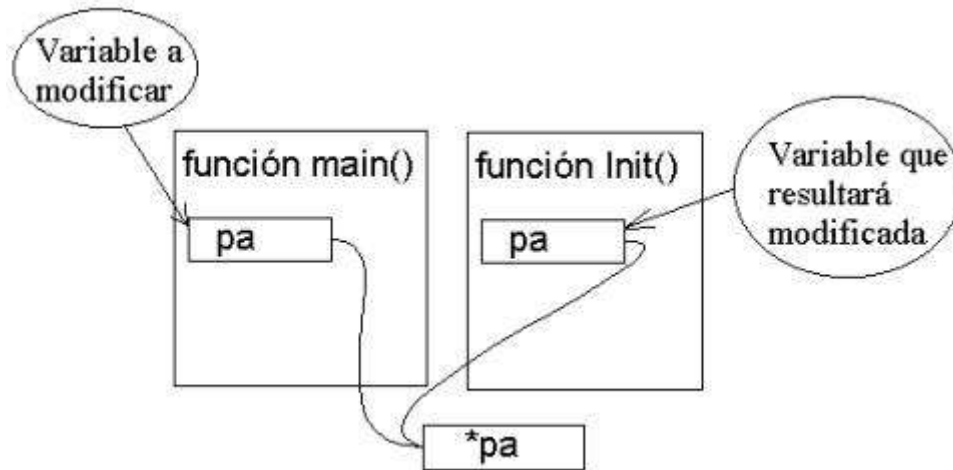
Ejemplo 8.6: Punteros a punteros (versión incorrecta)

```
/* Programa incorrecto */
#include <stdio.h>

void Init (int *pa)
{
    pa = NULL;
}

void main (void)
{
    int *pa;
    Init (pa);
}
```

Nuevamente, el programa es sintácticamente correcto. ¿Qué está mal entonces? Al llamar a la función *Init()* se pasa como parámetro una copia de la variable **pa**. Esta copia se inicializa en **NULL**, pero el valor original no fue modificado.



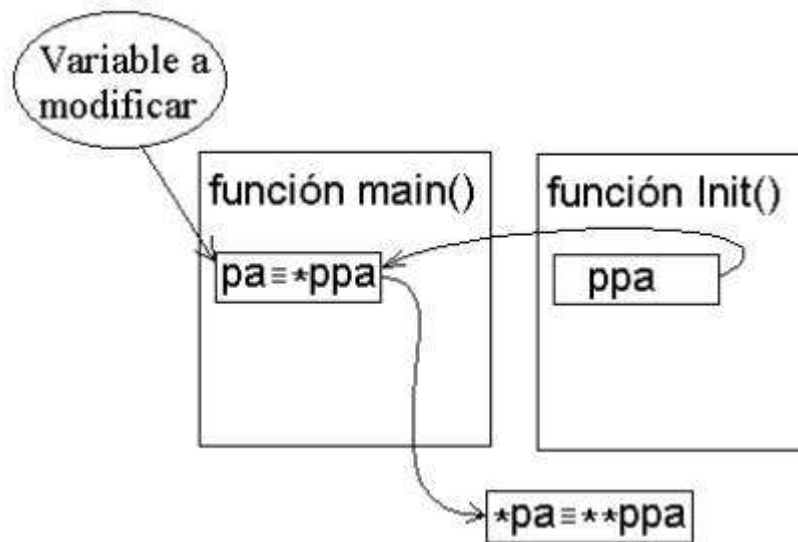
Como puede verse en el dibujo, desde la función *Init()* simplemente no hay forma de modificar la variable **pa** de la función *main()*, ya que no se tiene acceso a ella. Para solucionar esto, debe modificarse tanto la función como la llamada a la misma.

Veamos ahora la versión corregida:

```
#include <stdio.h>

void Init (int **ppx)
{
    *ppx = NULL;
}

void main (void)
{
    int *a;
    Init (&a);
}
```



Notar como en la función *Init()* debió utilizarse un puntero a puntero a entero. Pueden definirse de esta forma, punteros a punteros, indefinidamente. Sin embargo no hay ninguna razón útil para definir más de dos niveles de punteros.

Capítulo IX - Memoria dinámica y punteros a vectores

Veamos otro ejemplo similar al del capítulo anterior, pero ahora con string:

Ejemplo 9.1: Punteros y vectores

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

#define MAX_STRING 1000

void CargarString (char *s)
{
    printf ("Ingrese el texto:\n");
    scanf ("%s", s);
}

void MostrarString (char * s)
{
    printf ("String=%s\n", s);
}

void main (void)
{
    char *s1;

    s1 = (char *)malloc (MAX_STRING);
    CargarString (s1);
    MostrarString (s1);

    free (s1);
}
```

s1 se define como un puntero a un caracter (recordemos que 1 caracter ocupa 1 byte en memoria). Sin embargo en la segunda línea de la función *main()* reservaron 1000 bytes para dicha variable, es decir que estoy reservando memoria para 1000 caracteres. Esta información no se tiene en el puntero. Este simplemente apunta al primer carácter del arreglo (array) o vector, pero no conoce el tamaño del mismo.

La función **malloc()** es la encargada de solicitar un bloque de memoria al sistema operativo. Esta función devuelve un puntero genérico (**void ***), que es un puntero que no apunta a ningún tipo de dato en particular o, dicho de otra forma, es una dirección de memoria en donde no se ha especificado qué tipo de dato hay. Para cargar este valor en la variable *s1*, que es un puntero a caracter (**char ***), debe utilizarse algo llamado cast o casting, esto es transformación de un tipo de variable a otro (ya visto anteriormente), anteponiendo al valor devuelto por **malloc()** la conversión (**char ***).

La función **free()** sirve para devolver la memoria solicitada al sistema operativo. Al terminar todo programa, el sistema operativo automáticamente libera toda la memoria que el mismo haya solicitado y que no haya sido liberada, razón por la cual el último free no es necesario, pero es conveniente ponerlo por claridad, o por si en un futuro que quiere transformar el programa en una función de un programa mayor.

Ambas funciones, *malloc()* y *free()*, están definidas en *stdlib.h*.

Luego llamo a la función *CargarString()*, que recibe una copia del puntero a la cadena de caracteres, y llama a la función *scanf()* también pasándole una copia de dicho puntero.

Finalmente se llama a la función *MostrarString()*, a la cual se le pasa una copia del puntero a los datos.



Los vectores inicializados de esta forma (*malloc - free*) se los conoce como vectores o arrays dinámicos, también llamados memoria dinámica, en tanto que los vectores utilizados anteriormente, que se almacenan en la pila del sistema, se conocen como vectores o arrays estáticos, también llamados memoria estática.

Pasaje de vectores a funciones

Veamos otro ejemplo similar, del mismo programa. Es conveniente que se vaya comparando el código con el ejemplo anterior.



Ejemplo 9.2: Pasaje de vectores a funciones

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

#define MAX_STRING 1000

void CargarString (char s[])
{
    printf ("Ingrese el texto:\n");
    scanf ("%s", s);
}

void MostrarString (char s[])
{
    printf ("String=%s\n", s);
}

void main (void)
{
    char s1[MAX_STRING];
    CargarString (s1);
    MostrarString (s1);
}
```

El programa es idéntico, salvo por el hecho de que ahora el string está definido de otra forma. Notar que en la función *main* se define a *s1* como:

```
char s1[MAX_STRING];
```

Esto hace que el vector esté definido en forma "**estática**", es decir al ejecutarse la función *main()* se reservan automáticamente *MAX_STRING* caracteres para almacenar el dato, y los liberará automáticamente al salir de la función.

¿Dónde se almacenarán los datos? Rta: en la pila del sistema. Si se trabaja con sistemas operativos de 16 bits (D.O.S. y Windows 3.1) la pila del sistema es muy escasa (algunos Kbytes), en tanto que en sistemas operativos de 32 bits (Windows 95, 98, NT, UNIX), la misma suele ser de algunos MegaBytes, lo cual es bastante pero no excesivo. Por esta razón conviene definir de esta forma sólo variables pequeñas.

¿Qué pasa cuando se llama a las funciones? No se ha definido que se pase un puntero. ¿Se pasa una copia completa del vector?. La respuesta ya vista anteriormente es NO. **Siempre que en C una función reciba un vector, recibirá tan solo el puntero a los datos**, es decir que es lo mismo definir una función como:

```
MiFuncion (TipoDato Dato[])
```

o como

```
MiFuncion (TipoDato *Dato)
```

lo cual es más claro, y por eso generalmente se utiliza esta notación.

Aritmética de punteros

Pero sí es equivalente definir ***TipoDato Dato[]*** ó ***TipoDato *Dato***, también debería ser equivalente definir ***Dato[0]*** ó ****Dato***. Y la respuesta es que efectivamente lo es.

Más aún, también es equivalente escribir

```
Dato[n]    ó    *(Dato + n)
```

Esto está claro al trabajar con vectores de caracteres, como en el siguiente ejemplo:

Ejemplo 9.3: Aritmética de punteros

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

void main (void)
{
    char *string;
    string = (char*)malloc (1000);

    string[1] = 0;        /* Cargo el la posición 1 del vector el valor 0 */
    *(string+1) = 0;    /* Cargo el la posición 1 del vector el valor 0 */
}
```

En este ejemplo, las dos últimas líneas del programa son equivalentes.

Si string tuviese, por ejemplo, el valor 5200 en la posición de memoria 5200 estará el primer elemento del vector. Pues bien, string+1 será 5201, con lo cual al hablar de *(string+1) se estará hablando del contenido de la posición 5201, o lo que es lo mismo, el segundo elemento (posición 1) del vector.

¿Qué hubiese pasado si se tenía un vector de enteros? Veámoslo.

Ejemplo 9.4: Aritmética de punteros

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

void main (void)
{
    int *string;
    string = (char*)malloc (1000 * sizeof (int));

    string[1] = 0;      /* Cargo el la posición 1 del vector el valor 0 */
    *(string+1) = 0;    /* Cargo el la posición 1 del vector el valor 0 */
}
```

Supondré en este análisis que se trabaja en un sistema operativo de 32 bits, en donde el tamaño del entero es de 32 bits (4 byte).

Notar que en este caso , en la función *malloc()* se reservó memoria para 1000 enteros, no 1000 bytes. La función **sizeof()** será interpretada por el compilador, y será reemplazada por el tamaño del tipo de dato especificado.

Supongamos nuevamente que el valor devuelto por *malloc()* y copiado en string es 5200.

La línea *string[1] = 0;* claramente copia el valor 0 en la posición 1 del vector. Pero ¿qué ocurre con la siguiente?

Si string tiene el valor 5200, sería lógico suponer que string+1 tendrá el valor 5201, pero esto no es así. Por el contrario, tomará el valor 5204. ¿qué ha sucedido? **Al sumarle a un puntero un valor, no se suma dicha cantidad a la dirección de memoria, sinó que se aumenta el puntero dicha cantidad de elementos.**

De idéntica forma, puede efectuarse diferencia (resta) de punteros, y el valor devuelto no será la diferencia aritmética de las direcciones sino la cantidad de elementos comprendidos entre los dos punteros.

Volvamos ahora al ejemplo anterior, en donde el string se definía como un puntero a caracter, y se reservaba memoria para el mismo. ¿qué pasaría si quisiésemos que la misma función para cargar el dato fuese la que reservase memoria para el mismo?

Ejemplo 9.5: Punteros

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>

#define MAX_STRING 1000
```

```
char * DevolverString (void)
{
    char * s;

    s = (char *)malloc (MAX_STRING);

    if (!s)
    {
        printf ("Memoria insuficiente\n");
        exit(1);
    }

    printf ("Ingrese el texto:\n");
    scanf ("%s", s);

    return s;
}

void CargarString (char **s)
{
    *s = (char *)malloc (MAX_STRING);

    if (!*s)
    {
        printf ("Memoria insuficiente\n");
        exit(1);
    }

    printf ("Ingrese el texto:\n");
    scanf ("%s", *s);
}

void MostrarString (char * s)
{
    printf ("%s\n", s);
}

void main (void)
{
    char *s1, *s2;

    s1 = DevolverString ();
    CargarString (&s2);

    MostrarString (s1);
    MostrarString (s2);

    free (s1);
    free (s2);
}
```

Veamos que, al igual que en el ejemplo de los enteros, hay dos funciones para cargar el string. La primera de ellas, *DevolverString()* define un puntero a la cadena de caracteres ***char *s***;

Luego reserva memoria y carga dicho puntero la dirección donde se reservó la memoria, y verifica que el resultado de esta operación haya sido exitoso. Si no hubiese memoria disponible, la función *malloc()* retornará NULL (que es un #define con el valor 0).

Finalmente se carga el dato y se devuelve una copia de dicho puntero.

La segunda función *CargarString()* es un tanto más complicada. Ya que debe devolverse a la función principal un *puntero*, esta función deberá saber la posición de memoria donde debe almacenar dicho puntero, es decir, deberá conocer la posición de memoria de un puntero, o lo que es lo mismo, un puntero a un puntero a una cadena de caracteres. ¿suena complicado? Es necesario practicar bastante con esto para comprenderlo correctamente.



Notar que **s* es el dato que la función debe devolver, y *s* es la posición de memoria donde está almacenada esta variable.

Finalmente reservo memoria para el dato, y almaceno la dirección en **s*, es decir, en el puntero a la cadena de caracteres.

El modificador const y los punteros

a hemos visto la palabra reservada *const*, que permite garantizar que el valor de una variable se mantendrá constante durante toda la ejecución de la función. Por ejemplo:

```
void Funcion (const unsigned Valor)
{
    unsigned a;
    a = Valor;
    a++;
    Valor++; /* Incorrecto, la variable Valor es constante y no puede ser
modificada */
}
```

Los punteros, son variables y por lo tanto también pueden ser especificados como constantes. Pero en este caso existen dos variables, que permiten especificar si el puntero o el valor especificado es constante. Veamos un ejemplo. En este caso el valor del puntero será constante, pero no los datos apuntados.

```
void Funcion (char * const p)
{
    char *s;

    *p = 'A'; /* Correcto */
    p[2] = 'A'; /* Correcto */
    *(p+5) = 'A'; /* Correcto */
}
```

```
++p;                /* Incorrecto, p es constante */
((char *)p) = NULL; /* Incorrecto, p es constante */
s = p;              /* Correcto */
++s;                /* Correcto */
}
```

Por otro lado, también es posible especificar que el valor apuntado por *p* es constante, que es lo que generalmente se requiere. En este caso se tendrá:

```
void Funcion (const char *p)
/*
También puede escribirse
void Funcion (char const *p)
*/
{
    *p = 'A';          /* Incorrecto */
    p[5] = 'A';        /* Incorrecto */
    *(p+5) = 'A';      /* Incorrecto */
    *((char *)p+5) = 'A'; /* Correcto, no recomendado */
    ((char *)p)[5] = 'A'; /* Correcto, no recomendado */
    (char *)p++;        /* Correcto, no recomendado */
}
```

Notar que en estos casos el puntero *p* no es modificable directamente pero sí la posición apuntada por *p*. Puede modificarse el valor de *p* mediante un *cast* (conversión de tipo), pero esto no es recomendable.

También es posible especificar que la posición apuntada por *p* no pueda ser modificada. Esto se hace simplemente cambiando la posición de la palabra reservada **const**.

```
void Funcion (char * const p)
{
    *p = 'A';          /* Incorrecto */
    ++p;               /* Correcto */
    p[5] = 'A';        /* Incorrecto */
    *(p+5) = 'A';      /* Incorrecto */
    ((char *)p)[0] = 'A'; /* Correcto, no recomendado */
    *((char *)p+5) = 'A'; /* Correcto, no recomendado */
}
```

Desde ya que es posible especificar las dos opciones, es decir un puntero que no pueda ser modificado, cuyo valor apuntado tampoco pueda serlo. En este caso se tendría:

```
void Función (const char * const p)
```

También es posible que una función devuelva un puntero constante. Veamos un ejemplo:

```
const char * Funcion (const char * p)
{
    return p;
}
```

En este caso, podrán realizarse operaciones tales como:

```
char c;  
c = Funcion("Hola") [0];
```

pero no

```
char c;  
Funcion("Hola") [0] = c;
```

En ningún caso es obligatorio el uso de la palabra **const**, pero sí es buena práctica utilizarla, y puede ayudar a detectar errores, así como a dar claridad al código. Sin embargo su uso es un tanto tedioso, por lo que en la práctica muchas veces se la omite, o se la utiliza sólo en el desarrollo de librerías.

Capítulo X - Punteros y estructuras de datos

Punteros a estructuras

Definamos un registro para almacenar datos de personas:

```
typedef struct
{
    char Nombre[19];
    unsigned Edad;
    char Direccion[40];
    char Telefono[10];
}TDato;
```

Supongamos que queremos almacenar los datos de unas 1000 personas. Para ello deberémos generar un vector con elementos de tipo **TDato**, con 1000 posiciones:

```
void main(void)
{
    TDato Registros[1000];
}
```

Sin embargo, como ya se dijo, los vectores definidos en forma estática se almacenan en la pila del sistema, por lo cual no es bueno el uso indiscriminado de la misma. En un sistema operativo de 16 bits esto simplemente no sería posible, ya que excedería su capacidad. En un sistema de 32 bits, esto todavía es posible, pero de ninguna manera recomendable. Utilicemos entonces memoria dinámica:

```
/* Este programa es incorrecto */
void main (void)
{
    TDato *Registros;

    Registros = (TDato*) malloc (73 * 1000);
    printf ("Tamaño del registro: %d\n", sizeof (TDato));
}
```

Nuevamente este programa es sintácticamente correcto. Más aún, si sumamos el tamaño de los campos de la estructura **TDato** veremos que efectivamente suma 73, con lo cual 73*1000 debería ser la memoria requerida para almacenar el dato.

Sin embargo, si ejecutamos el programa veremos que el valor impreso (en la mayoría de los compiladores) es 74. ¿Qué ha sucedido?

Rta: Por razones internas de aquitectura de las computadoras, estas tienen problemas para acceder a enteros en posiciones de memoria no múltiplos del tamaño del mismo. Esto es, un entero de 32 bits (4 bytes) debe estar en posiciones de memoria múltiplo de 4. Cuando esto no ocurre se dice que el dato está desalineado. En arquitecturas PC, acceder a un dato desalineado requiere el doble de tiempo. En arquitecturas RISC esto directamente no es posible, y si se lo hace, el programa se interrumpirá.

Para solucionar esto, el compilador dejará un espacio desperdiciado, a continuación del campo Nombre, y almacenará el campo Edad en la vigésima posición del registro, con lo cual el tamaño del registro será 74.

Por esta razón, nunca debe suponerse el tamaño de una estructura, sino que debe utilizarse **sizeof()**. Corrijamos ahora nuestro programa:

```
void main (void)
{
    Tdato *Registros;

    Registros = (Tdato*) malloc (sizeof (Tdato) * 1000);
}
```

Escribamos ahora una función que cargue datos en un registro del vector.

```
void CargarRegistro (Tdato * Reg)
{
    strcpy ( (*Reg).Nombre , "Juan");
    (*Reg).Edad = 70;
    strcpy ( (*Reg).Direccion , "Santa Fe 2102");
    strcpy ( (*Reg).Telefono , "344-1111");
}
```

Notar la forma en que debieron escribirse cada uno de los campos del registro:

`(*Reg).Campo`

ya que el uso de paréntesis suele resultar molesto, C permite escribir esto mismo como sigue:

`Reg->Campo`

con lo cual, esta última función quedaría:

```
void CargarRegistro (Tdato * Reg)
{
    strcpy ( Reg->Nombre , "Juan");
    Reg->Edad = 70;
    strcpy ( Reg->Direccion , "Santa Fe 2102");
    strcpy ( Reg->Telefono , "344-1111");
}
```

Agreguemos a nuestro ejemplo una función para imprimir un registro.

```
void MostrarRegistro (Tdato Reg)
{
    ... código necesario para imprimir el registro...
}
```

Si bien este código es correcto, constituye un ejemplo de mala programación. En este caso, el registro no deberá ser modificado, razón por la cual no hace falta pasar un puntero al registro, y puede pasarse el registro completo. Sin embargo, hacer esto requiere crear una copia del registro innecesariamente, desperdiciando así tiempo que podría ser aprovechado.

Este tiempo desperdiciado es pequeño, pero se torna crítico cuando la operación se realiza en forma repetida. Para tener una idea de cómo esto puede llegar a afectar los resultados finales, un programa de encriptado de datos que implementa el tipo de dato entero grande (512 bits), llega a correr unas 10 veces más rápido cuando los estos datos se pasan como punteros, en vez de por valor.



Preferentemente no pasar una estructura como parámetro a una función sino un puntero a la misma.

Capítulo XI - Archivos

En **stdio.h** se define un conjunto standard muy extenso de funciones con las cuales acceder a archivos. Veremos aquí las funciones fundamentales.

```
FILE *fopen (char *archivo, char *modo);
```

Abre un stream a un archivo. Por ahora diremos que esto es equivalente a abrir un archivo, después se verá la diferencia.

archivo es el nombre del archivo a abrir, con el path completo. Si no se indica el path, se usará el directorio actual.

El modo puede ser cualquiera de los siguientes:

```
"r"      Abre el archivo para lectura. El archivo debe existir.
"w"      Abre el archivo nuevo para escritura. Si el archivo ya existe será
sobreescrito.
"a"      Abre el archivo en modo append, es decir agregado. Se empezará a
escribir al final.
"r+"     Abre el archivo para lectura y escritura. El archivo debe existir.
"w+"     Abre el archivo nuevo para lectura y escritura. Si el archivo ya existe
será sobreescrito.
"a+"     Abre un archivo para lectura y agregado.
```

Estos modos deben estar seguidos de la letra 't' (modo texto) o 'b' (modo binario). El segundo de estos dos modos es más eficiente, ya que se lee y escribe sin realizar ninguna conversión. En el primero, cada vez que se escriba un salto de línea, será grabado en el archivo en la forma que utilice el sistema operativo. En UNIX el salto de línea se almacena simplemente como un salto de línea. En DOS/Windows el mismo se almacena como dos bytes, un salto de línea y un retorno de carro. El proceso inverso tiene lugar al leer el archivo.

El modo de apertura del archivo puede resumirse en la siguiente tabla, lo cual da un total de 12 modos:

r		b
	+	t
a		

En caso de poder abrir el archivo, se devuelve un puntero a la información requerida para acceder al mismo. En caso de error se devuelve NULL.

```
int putc (int c, FILE *stream);
```

Escribe un byte en el stream especificado. En caso de error devuelve EOF (255). En caso contrario el byte escrito.

```
int getc (FILE *stream);
```

Lee un byte del archivo. En caso de error devuelve EOF (255).

```
int fclose (FILE *stream);
```

Cierra el archivo abierto.

```
size_t fwrite (const void *buffer, size_t size, size_t count, FILE *stream);
```

Escribe un dato cualquiera al stream. Recibe un puntero al dato a grabar, el tamaño del mismo, la cantidad de elementos (en caso de tratarse de un vector) y el stream donde grabar los datos.

```
size_t fread (void *buffer, size_t size, size_t count, FILE *stream);
```

Lee un dato cualquiera al stream. Recibe un puntero al dato a leer, el tamaño del mismo, la cantidad de elementos (en caso de tratarse de un vector) y el stream de donde leer los datos.

```
int fprintf (FILE *stream, char *format,...);
```

Similar a printf, pero en lugar de imprimirse en pantalla se imprime a un stream.

```
int fseek (FILE *stream, long offset, int origin);
```

Cambia la posición dentro del stream. Recibe el stream a cambiar, la nueva posición, y un parámetro (origin) que indica desde donde se la debe medir. Este parámetro puede ser:

SEEK_SET:	Desde el comienzo del archivo
SEEK_END:	Desde el final
SEEK_CUR:	Desde la posición actual

Escribamos por ejemplo un programa que cree, escriba y lea un archivo cualquiera.

Ejemplo 11.1: Acceso a archivos

```
#include <stdio.h>

void main (void)
{
    FILE *fp;
    char c;

    /* Abro un archivo para escritura en modo texto */
    fp = fopen ("prueba.txt", "wt");

    /* Verifico que no se hayan producido errores */
    if (fp == NULL)
```

```
{
    printf ("Error al crear el archivo\n");
    exit (1);
}
fprintf (fp, "Hola mundo\n"); /* Escribo un texto en el archivo */
fclose (fp);                  /* Cierro el archivo */

/* Abro nuevamente el archivo, pero ahora para lectura en modo texto,
   y verifico que no se hayan producido errores */

if ((fp = fopen ("prueba.txt", "wt")) == NULL)
{
    printf ("Error al abrir nuevamente el archivo\n");
    exit (1);
}
/* Leo un caracter. Si no se trata del fin de archivo lo imprimo */
while ((c = getc (fp)) != EOF)
    printf ("%c", getc(fp));

/* Cierro el archivo */
fclose (fp);
}
```

Streams

Hasta ahora hemos procurado evitar la palabra *archivo*, y usado en lugar de ella la palabra *stream*. Pero ¿qué es un *stream* y qué diferencia tiene con el acceso directo a un *archivo*? Para responder esta pregunta veamos el siguiente ejemplo:

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

void main(void)
{
    FILE *fp;

    /* Abro un archivo para escritura en modo binario */
    if ((fp = fopen ("prueba.txt", "wb")) == NULL)
    {
        printf ("Error al crear el archivo\n");
        exit (1);
    }
    putc (10, fp);      /* Escribo el byte 10 en el archivo */
    fclose (fp);        /* Cierro el archivo */
}
```

Si se verifica el tiempo requerido para ejecutar la llamada a *putc()* mediante programas destinados a tal fin, se verá que esta requiere algunos microsegundos. Sin embargo, el tiempo requerido por el disco rígido para acceder al archivo y grabar el byte es de varios milisegundos. ¿qué ha sucedido? La respuesta es que el byte no ha sido copiado al archivo, sino a un buffer temporal y propio del sistema operativo. Esto es un *stream*.

El valor 10 no ha sido aún escrito en el disco. El valor OK que devuelve el sistema operativo no significa que el mismo haya grabado el valor en el archivo sino tan sólo:

"he tomado conocimiento de que el programa requiere que se almacene este byte en una determinada posición del archivo y he copiado este valor a un buffer propio sin que esto produzca ningún error. Posteriormente, cuando yo lo desee y si puedo hacerlo, escribiré este valor en el archivo y no notificaré a nadie el resultado de la operación".

Esta forma de trabajar del sistema operativo hace que los programas funcionen en una forma muy rápida y eficiente, pero tiene sus riesgos, ya que si el funcionamiento de la computadora se interrumpe súbitamente los datos se perderán indefectiblemente.

Para evitar esto, existe la función `fflush()`.

```
int fflush(FILE *fp);
```

Esta función ordena al sistema operativo que los datos almacenados en el buffer del stream sean transferidos inmediatamente al archivo correspondiente.

Entrada/salida standard

Todo programa al comenzar recibe tres streams ya abiertos:

stdin: Entrada standard, generalmente el teclado.
stdout: Salida standard, generalmente pantalla.
stderr: Salida standard de errores, generalmente pantalla.

Por ejemplo, podría escribirse:

```
fprintf (stdout, "Hola mundo\n");
```

y este texto sería impreso en pantalla.

Únicamente bajo D.O.S. (no así bajo Windows ni UNIX ni Windows) se definen también:

stdaux: Salida auxiliar. Rara vez se la utiliza.
stdprn: Impresora.

Entrada/salida sin streams

Existen también funciones para acceder directamente a archivos sin trabajar con streams. Estas son (entre otras):

```
int open (const char *filename, int oflag [, int pmode]);  
int read (int handle, void *buffer, unsigned int count);  
int write (int handle, void *buffer, unsigned int count);  
int close (int handle);
```

Su uso es poco frecuente y muy similar a las anteriormente vistas para streams, razón por las cuales no se las analizará aquí.

Capítulo XII - Temas varios de C

Variables static

En algunos casos puede ser útil tener variables locales a una función, que conserven su valor aún al salir de la misma. Un ejemplo de esto sería una variable contadora, que fuese local a una función `Timer()`, y que algún dispositivo externo incremente su valor. Otra razón puede ser una función que deba devolver un vector de datos, pero ya que esto no se puede hacer, se puede devolver un puntero a datos estáticos de la función. Pero para ello es necesario que el vector no se destruya al salir de la misma.

En todos estos casos, puede utilizarse la palabra reservada ***static***. Una variable de tipo ***static*** local a una función conservará su valor y no se destruirá al salir de la función. La misma no será duplicada si se implementa recursividad.



Una variable local *static* se comporta en forma idéntica a una variable global, con la única diferencia que sólo será conocida por la función donde se la definió.



Ejemplo 12.1: Variables static

```
int Timer(void)
{
    /* El valor 0 se cargará en la función al comenzar el programa */
    static int Contador = 0;
    Contador++;

    return Contador;
}
```

¿Qué ocurre con las variables globales ***static***?



Una variable global *static* no podrá ser leída ni modificada por otros archivos del proyecto, es decir, pueden haber varias variables globales static con el mismo nombre en diferentes archivos del proyecto, sin que estas interfieran unas con otras.

Variables volatile

Otro modificador aplicable a las variables es la palabra reservada ***volatile***. Una variable definida de esta forma no será optimizada por el compilador, y su valor será leído de memoria cada vez que se lo requiera. Este modificador se utiliza cuando una variable puede ser modificada en forma externa. Este tipo de variables son frecuentes en sistemas que comparten el bus de datos con dispositivos externos. Imaginemos un programa que atienda comunicaciones por modem u otro dispositivo de entrada/salida:

Ejemplo 12.2: Variables volatile

```
void main(void)
{
    volatile char *pLlamadaEntrante;

    /* Esta dirección de memoria 0x21000001 contiene el area de datos del modem */
    pLlamadaEntrante = 0x21000001;

    while (!*pLlamadaEntrante);

    AtenderLLamada();
}
```

En este caso, la variable **pLlamadaEntrante** no apuntará a un área de memoria física, sino a un puerto de entrada/salida o un área de registros de un dispositivo externo. El compilador necesita saber que dicha variable deberá ser efectivamente leída y verificada cada vez que se la requiera, para evitar utilizar optimizaciones tales como cargar la variable en un registro y leer su valor del mismo. Las variables **volatile** son rara vez utilizadas en sistemas modernos ya que el sistema operativo se encarga de todas las operaciones de entrada/salida, y sólo se las utiliza en el desarrollo de drivers.

¿Cómo recibir parámetros de la línea de comandos?

La función `main()` ha sido hasta ahora declarada como:

```
void main(void)
```

Sin embargo, la misma puede recibir y devolver parámetros. En primer lugar la función **main()** puede devolver un entero al sistema operativo, que generalmente se conoce como **ErrorLevel**. Este valor puede ser utilizado por otros programas para saber si el mismo terminó correctamente o no.

Pero la función **main** también puede recibir parámetros. Estos son conocidos como **argc** y **argv**, si bien podría elegirse otro nombre, y se definen como sigue:

```
int main (int argc, char *argv[])
```

El primero contiene la cantidad de parámetros de la línea de comandos, con que se llamó al programa, valor que será por lo menos 1.

El segundo es un puntero a una vector de punteros a caracter, que contienen los parámetros pasados en la línea de comandos. En el primer parámetro de esta estructura es el nombre del programa con el path completo. Veamos un ejemplo:

Ejemplo 12.3: Línea de comandos

```
#include <stdio.h>

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

```
if (argc < 2)
{
    printf ("No se especificaron parámetros.\n");
    printf ("El programa debe llamarse como %s nombre_del_usuario\n",
        argv[0]);
    return 1;
}
printf ("Hola %s\n", argv[1]);
return 0;
}
```

Si ejecutásemos este programa sin parámetros, la salida sería la siguiente:

```
No se especificaron parámetros.
El programa debe llamarse como C:\devel\ejemplos\params.exe nombre_del_usuario
```

y devolvería el valor 1 al sistema operativo.

Si el programa se ejecutase con el parámetro Juan, el programa imprimiría en pantalla

```
Hola Juan
```

y devolvería el valor 0 al sistema operativo.

Existe un tercer parámetro, pocas veces utilizado, llamado **envp**. Este parámetro contiene todas y cada una de las líneas de las variables de entorno del sistema.

```
int main (int argc, char *argv[], char *envp[]);
```

Microsoft ha desarrollado una nueva variante de la función **main()**, que es rara vez utilizada, pero permite compatibilidad con varios lenguajes.

```
int wmain (int argc, wchar_t *argv[], wchar_t *envp[])
```

wchar_t es un nuevo tipo de dato (wide character) que consiste en entero de 16 bits, utilizado para representar caracteres del código extendido conocido como Unicode.

Punteros a funciones

Suele ser muy útil y práctico utilizar punteros a funciones, para modificar o personalizar el funcionamiento de una función o una librería. Un puntero a una función es idéntico a cualquier otro puntero, y se lo declara anteponiendo un asterisco al nombre de la función, encerrados ambos entre paréntesis. Por ejemplo:

```
int (*comparar) (char * a, char *b);
```

Será un puntero llamado **comparar**, cuyo contenido será la dirección de memoria de una función que recibirá dos punteros a carácter y devolverá un entero.

Esta función puede ser llamada como cualquier otra función, pero desde ya que deberá tener un valor válido.

Veamos un ejemplo: Supongamos que se tiene un vector de string, que se quiere ordenar.

Ejemplo 12.4: Punteros a funciones

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TOTAL      12
#define MAX_NAME   20

void Ordenar (char *Nombres[MAX_NAME], int Total, int (*CmpFunc) (char *, char
*))
{
}

void ListarNombres (char *Nombres[MAX_NAME], int Total)
{
    int i;
    for (i=0;i<Total;++i)
        fprintf ("%s\n", Nombres[i]);
}

void main(void)
{
    char Nombres[TOTAL][MAX_NAME] = {
        "Juan", "pedro", "María", "Gabriel",
        "MARTA", "Pablo", "Carolina", "Esteban",
        "Laura", "Patricia", "josefina", "EZEQUIEL"
    };

    printf ("Listado ordenado alfabéticamente\n");
    Ordenar (Nombres, TOTAL, strcmp);
    ListarNombres (Nombres, TOTAL);

    printf ("Listado ordenado alfabéticamente, ignorando mayúsculas y
    minúsculas\n");
    Ordenar (Nombres, TOTAL, StrCaseCmp);
    ListarNombres (Nombres, TOTAL);

    printf ("Listado ordenado alfabéticamente, mujeres primero, hombres
    después\n");
    Ordenar (Nombres, TOTAL, LadiesFirstCmp);
    ListarNombres (Nombres, TOTAL);

    printf ("Listado ordenado alfabético en orden inverso\n");
    Ordenar (Nombres, TOTAL, );
    ListarNombres (Nombres, TOTAL);
}
```

El preprocesador

Una de las características interesantes del lenguaje C consiste en la potencia del preprocesador. Ya se mencionó este tema en el capítulo 2, pero ahora lo veremos con más detalle. Todas las directivas o directrices del preprocesador comienzan con el prefijo #, y **no llevan punto y coma final**.

La directiva **#include** permite incluir un archivo dentro de otro. Si bien puede incluirse cualquier archivo dentro de otro, sólo deben incluirse archivos header (.h, .hpp, .rh).

La directiva **#define** permite definir constantes y macros. Ya se vió la macro **#define** para definir constantes, pero también puede ser utilizada para definir macros, esto es, una constante que puede tomar parámetros. El formato es el siguiente:

```
#define NOMBRE_MACRO (parámetros)
```

Puede especificarse cualquier número de parámetros, pero siempre debe tratarse de un número fijo de parámetros. No se puede especificar una cantidad de parámetros variable, como en la función *printf()*. El nombre de la macro y el primer paréntesis **deben ser escritos sin ningún espacio en el medio**, para ser interpretada como una constante.

Las directivas **#ifdef**, **#ifndef** permiten verificar la definición de una constante.

La directiva **#if (condición)** permite verificar condiciones, en forma similar a if.

Tanto la directiva **#ifdef** como **#ifndef** e **#if (condición)** deben terminar en **#else** o **#endif**.

Existen también una serie de **constantes predefinidas** que resultan sumamente útiles:

<u>FILE</u>	Nombre del archivo actual (código fuente)
<u>LINE</u>	Línea del programa
<u>TIME</u>	Un string con la fecha de compilación del programa

Veamos un ejemplo:

Ejemplo 12.5: El preprocesador - definición de macros

```
#define max(a,b) ((a)>(b) ? (a) : (b))
```

```
void main (void)
{
    int a, b=50;
    a = max (a,10);
}
```

Esta macro permite obtener fácilmente el máximo entre dos números. Si se utilizase una función en lugar de una macro, habría que escribir una función para todos los tipos de datos existentes (enteros, números de punto flotante, etc). Debe tenerse en cuenta que las macros pueden llevar a confusiones si no se las utiliza correctamente. Por ejemplo:

```
int a, b = 50;
a = max (b++, 10);
```

No incrementará la variable **b** una vez sino dos veces, ya que en realidad el código es:

```
a = ((b++) > (10) ? (b++) : (10));
```

Notar también la importancia de utilizar todos los paréntesis. Si no se lo hiciese se podrían producir errores. Por ejemplo, si se definiese:

```
#define max(a,b) a > b ? a : b
```

funcionará bien para:

```
a = max (b, 10);
```

pero no para:

```
a = max (b, 10) / 2;
```

Otro ejemplo de macros muy utilizadas son macros como la siguiente:

Ejemplo 12.6: Macro TRACE

```
#include <stdio.h>

#define TRACE(Txt) {\
    FILE *fp=fopen("salida.txt", "at");\
    if (fp) { fprintf ("Archivo:%s Línea:%5u Texto:%s\n", __FILE__, __LINE__, Txt);\
    fclose(fp); }}
```

```
void main (void)
{
    TRACE ("Comenzando el programa");
    ...
    TRACE ("Fin del programa");
}
```

Esta macro grabará en un archivo (salida.txt) información especificada en el programa, junto con el nombre del archivo y la línea del mismo. Estos códigos son muy útiles en el momento de verificar el funcionamiento de un programa (debug). El carácter '\n' sirve para continuar una línea en el siguiente renglón.

El archivo salida.txt contendrá, luego de la ejecución del programa:

```
Archivo:c:\devel\test1.c Línea:   9 Texto:Comenzando el programa
Archivo:c:\devel\test1.c Línea:  11 Texto:Fin el programa
```

Macros de este tipo suelen ser más complejas, guardando también la fecha y hora en que se produjo el evento, junto con otros datos que pueden ser útiles.

Las definiciones también son utilizadas para modificar un programa de tal forma que sea válido para todas las plataformas. Una serie de macros utilizadas para lograr esto son las siguientes:

```
/* Defino el entero de 16 y 32 bits, para todas las plataformas */
#if defined ( _MSDOS ) || defined ( _WIN16 )
    typedef int    int16;
    typedef long   int32;
```

```
#else
    typedef short int16;
    typedef int    int32;
#endif
```

Existen algunas directrices para activar y desactivar las advertencias (warnings) del compilador, o para indicar que se compile el programa junto con una determinada librería, pero estas directrices no forman parte del standard y varían de un compilador a otro.

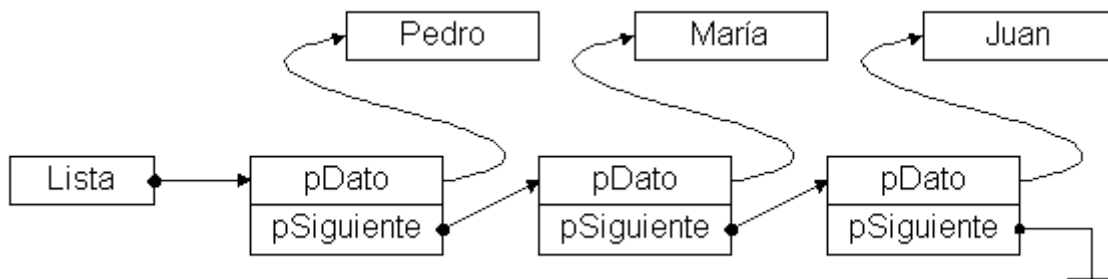
Capítulo XIII - Estructuras de datos

Listas enlazadas

Otro uso muy común de los punteros es la creación de estructuras de datos complejas, tales como listas, árboles, grafos, etc.

```
typedef struct StrLista
{
    char Nombre[100];
    struct StrLista *Siguiete;
}TLista;
```

Veamos un ejemplo; creemos una lista para almacenar cadenas de caracteres. Antes de comenzar, debe recordarse que pueden crearse muchísimos tipos de estructuras 'lista', todas ellas completamente diferentes unas a otras y adaptadas al uso que se le dará. La estructura más genérica es la que se presenta a continuación, donde la lista se arma como nodos, con dos punteros, uno de ellos el puntero al dato, y el segundo el puntero al siguiente elemento de la lista. Este tipo de estructura, si bien resulta ser muy versátil, tiene la desventaja de ser sumamente ineficiente en el uso y administración de la memoria, velocidad de ejecución y, dependiendo del uso que se le de, también en búsqueda/direccionamiento de datos.



Ejemplo 13.1: Listas enlazadas

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
#include <string.h>

#define AL_PRINCIPIO 0
#define AL_FINAL 1

typedef struct str_nodo
{
    char *pDato;
    str_nodo *pSiguiete;
}TNodo, *TLista;

void AgregarDato (TLista *Lista, char *Dato, int PosFlag)
{
    TNodo *pNuevoNodo; /* Puntero al nuevo nodo */

    /* reservo memoria para el nuevo nodo */
```

```
pNuevoNodo = (TNodo *)malloc (sizeof (TNodo));
if (!pNuevoNodo)
{
    printf ("Memoria insuficiente\n");
    exit(1);
}

/* reservo memoria para el dato */
pNuevoNodo->pDato = (char *)malloc (strlen (Dato));
if (!pNuevoNodo->pDato)
{
    printf ("Memoria insuficiente\n");
    exit(1);
}

/* Copiar el dato al nodo */
strcpy (pNuevoNodo->pDato, Dato);

/* Insertar el dato en la lista */
if (PosFlag == AL_PRINCIPIO)
{
    pNuevoNodo->pSiguiente = *Lista;
    *Lista = pNuevoNodo;
}
else
{
    TNodo **ppLast;
    ppLast = Lista;

    /* El nuevo nodo no tiene sucesor */
    pNuevoNodo->pSiguiente = NULL;

    /* Debo recorrer la lista hasta el final */
    while (*ppLast!=NULL)
        ppLast = &((*ppLast)->pSiguiente);

    *ppLast = pNuevoNodo;
}
}

void MostrarLista (TLista pNodo)
{
    int i;

    i = 1;
    while (pNodo != NULL)
    {
        printf ("Dato %d = %s\n", i, pNodo->pDato);
        pNodo = pNodo->pSiguiente;
        ++i;
    }
}

void main (void)
{
    TLista Lista;

    /* La lista esta inicialmente vacía */
    Lista = NULL;
}
```

```
AgregarDato (&Lista, "Maria", AL_PRINCIPIO);
AgregarDato (&Lista, "Pedro", AL_FINAL);
AgregarDato (&Lista, "Raul", AL_FINAL);
AgregarDato (&Lista, "Juana", AL_PRINCIPIO);

MostrarLista (Lista);
}
```

Veamos el programa

El mismo comienza especificando los archivos a incluir; *stdio.h* (que define la entrada/salida standard), *malloc.h* (donde se definen las funciones *malloc()* y *free()*), *stdlib* (donde se definen funciones standard, tales como la funcion *exit()*), y *string.h* (donde se definen las funciones para manejo de strings o cadenas de caracteres).

Se definen luego dos constantes, que se usarán en el momento de insertar los datos, y luego la estructura que contendrá los nodos de la lista. Uno de ellos será el puntero al dato, y el otro el puntero al siguiente nodo de la lista.

```
typedef struct str_nodo
{
    char *pDato;
    str_nodo *pSiguiente;
}TNodo, *TLista;
```

Estas líneas podrían descomponerse para su mejor comprensión como sigue:

```
struct str_nodo
{
    char *pDato;
    str_nodo *pSiguiente;
};

typedef struct str_nodo TNodo;
typedef struct str_nodo *TLista;
```

Notar que se definen tres nombres relacionados con esta estructura:

struct str_nodo, TNodo y TLista.

Los dos primeros son directamente el nombre de esta estructura. El nombre *str_nodo* debe definirse para poder utilizarlo dentro de la estructura. Luego, para no tener que utilizar siempre '**struct str_nodo**', defino el tipo de dato **TNodo**, que será equivalente a este. Finalmente defino un puntero a esta estructura con el nombre **Lista**. Así, **Lista** será equivalente a ***TNodo**, y una **Lista** será un puntero al primer elemento de la lista. En muchos casos suele utilizar una estructura con una cabecera de la lista.

La primer función que se define es **AgregarDato()**. Esta función recibirá un puntero a la lista, es decir, un puntero al puntero al primer nodo. ¿Porqué no recibir simplemente una copia del puntero a la lista? Esto estaría bien, si se quisiese insertar el dato al final, pero si debo insertar un nuevo nodo al comienzo, deberé

modificar el valor del puntero al primer nodo, razón por la cual, deberé conocer el valor de la posición de memoria donde el mismo está almacenado.

En las siguientes líneas reservo memoria para el nuevo nodo, y verifico que la operación haya sido exitosa:

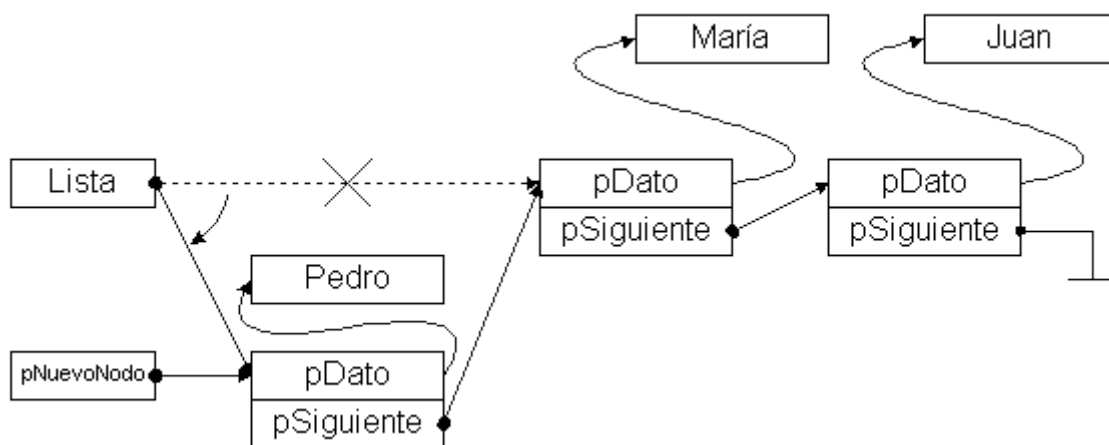
```
pNuevoNodo = (TNodo *)malloc (sizeof (TNodo));  
if (!pNuevoNodo)  
{  
    printf ("Memoria insuficiente\n");  
    exit(1);  
}
```

En forma similar reservo memoria para el dato, y copio luego el dato en la memoria que he reservado a tal fin.

El siguiente paso consiste en insertar el dato en la lista. Para esto, esta función presenta dos posibilidades. La primera, insertar el dato al comienzo. Para esto, ejecutan las siguientes líneas.

```
pNuevoNodo->pSiguiente = *Lista;  
*Lista = pNuevoNodo;
```

La primer línea modifica el puntero del nuevo nodo para que apunte a la lista, y el segundo el valor del puntero al primer nodo para que apunte al nuevo nodo. Debe estar claro que estas dos líneas no pueden ejecutarse en el orden inverso.



La segunda alternativa consiste en insertar el dato al final. Para ello debe recorrerse la lista hasta llegar al final de la misma. Notar que defino la variable **ppNodo** como un puntero al puntero al último nodo de la lista, para poder así modificar su valor. Así voy recorriendo la lista hasta llegar al final de la misma, y finalmente inserto el dato.

La función *MostrarLista()*, debe resultar ahora bastante más fácil de entender, y se deja para el lector. Notar que la misma recibe una copia del puntero al primer elemento de la lista, y no un puntero a este puntero, como en el caso anterior, debido a que este puntero no será modificado en la función.

Sería un muy buen ejercicio para el lector, escribir en este momento una función para borrar datos de la lista.

Otro ejercicio muy útil es la llamada lista doblemente encadenada. Esta es similar a la lista que acabamos de escribir, con excepción de que cada nodo tiene, además de un puntero al siguiente nodo de la lista, un puntero al nodo anterior, con lo cual pueden efectuarse recorridos en ambos sentidos.

Notar también que cada nodo tiene un tamaño fijo, pues almacena simplemente dos punteros, (un puntero al dato y otro al nodo siguiente). Despreciando temas de alineamiento (ver más adelante), cada puntero, suponiendo un ambiente de 32 bits, ocupará 4 bytes, por lo que cada nodo ocupará 8 bytes. Efectuar reservas de memoria (malloc) para tan solo 8 bytes es terriblemente ineficiente, puesto que cada nodo ocupará realmente más de 8 bytes (dependiendo del sistema operativo) y provocará una importante fragmentación de la memoria (quedarán bloques inutilizados). Pueden plantearse otras estructuras bastante más útiles, que contengan al nodo y al dato en un mismo bloque de memoria. Se verá un ejemplo de esto hacia el final del curso C.

Otro punto importante es el siguiente. Supongamos que definimos una estructura como sigue:

```
struct StrDato
{
    char c;
    int n;
}
```

Supongamos que trabajamos en un sistema operativo de 32 bits, donde un entero ocupa 4 bytes. ¿Qué tamaño ocupará esta estructura en memoria? Pues bien, sabemos que un carácter (char) ocupa 1 byte, y un entero 4 bytes, sería de esperar que esta estructura ocupe 5 bytes en memoria. Pero no es así.

El tamaño de esta estructura en memoria dependerá del sistema operativo y del compilador u opciones del compilador que se esté usando. Lo más común son 8 bytes. Lo que ocurre es que para muchos procesadores (casi todos los usados en este momento) resulta muy costoso acceder a un dato que se encuentra desalineado. Simplificando el tema, un entero está alineado cuando su posición en memoria es múltiplo del tamaño del mismo. Por ejemplo un entero de 32 bits (4 bytes) estará alineado si se encuentra en una posición de memoria múltiplo de 4.

En la PC, acceder a un entero desalineado requiere el doble de tiempo. En máquinas RISC esta operación directamente no se puede efectuar, y si se lo intenta el programa se interrumpirá.

Esta función devuelve el string que se copió, razón por la cual podrían efectuarse operaciones tales como:

```
char s[20];
printf ("Copio el string '%s'\n", strcpy (s, "Hola mundo"));
```

lo cual imprimiría:

```
Copio el string 'Hola mundo'
```

PARTE II - El lenguaje de programación C++

Capítulo XIV - Introducción a C++

Introducción

Con el objetivo de ampliar el lenguaje, Bell Laboratories añadió algunas extensiones al lenguaje C, y lo llamó lenguaje "C con clases", nombre que en 1983 se cambió a C++, si bien en los años siguientes y hasta 1989 se efectuaron cambios en el lenguaje, pero manteniendo la compatibilidad con el lenguaje C. Finalmente, durante la década del 90 se realizaron nuevas modificaciones, fundamentalmente se agregaron algunas características más, llegando así al standard ANSI C++ en 1998, que está basado fundamentalmente en el libro *The Annotated C++ Reference Manual*, de Margaret Ellis y Bjarne Stroustrup, y sobre el que se sabían la mayoría de los compiladores anteriores a esta fecha.

La mayoría de las extensiones que C++ agrega al C están relacionadas con la programación orientada a objetos (OOP: Object Oriented Programming). El objetivo de la programación orientada a objetos es permitir al programador trabajar con programas de gran envergadura, encapsulando el código y la información en un diversas estructuras (o clases) organizadas según una estructura jerárquica y abstrayéndose de la implementación concreta de cada una de las partes del programa. Estas son algunas de las ideas básicas que se buscaron al desarrollar el lenguaje C++.

C vs C++

La razón por la cual se desarrolló el lenguaje C++, es la presencia de algunos inconvenientes al desarrollar programas de gran envergadura, en cuanto a lo que se refiere al programador. Ellos son:

☹ Gran dificultad para mantener o modificar un programa.

☹ Gran cantidad de variables.

Si se usan variables globales, estas crecen indefinidamente.

Si se usan variables locales, se complica el llamado a las funciones por la gran cantidad de parámetros que hay que pasar.

☹ Dificultad para crear nuevos tipos.

☹ El programador debe recordar los nombres de demasiadas funciones.

En C++ se busca resolver estos problemas, manteniendo las ventajas y la potencia del lenguaje C, pero permitiendo la creación de estructuras de datos que permitan trabajar como en un lenguaje de más alto nivel. Todos estos puntos están muy relacionados con la aparición de una nueva filosofía de programación, conocida como Programación Orientada a Objetos, que no constituye una ruptura con el modelo de

Programación Estructurada (como lo fue este último con la programación imperativa) sino una extensión del mismo. Entre los puntos más interesantes de este nuevo modelo se encuentran:

- 😊 Las estructuras de datos pueden contener funciones.
- 😊 Se crean niveles de seguridad y accesibilidad a los datos.
- 😊 Se permite la redefinición de operadores (+, -, *, etc)
- 😊 Se crea el concepto de polimorfismo, esto es, que puedan existir funciones con un mismo nombre.
- 😊 Se crea la herencia, para que puedan crearse varios tipos de datos basados en una estructura en común.

Ya que C++ constituye una extensión del lenguaje C, todo programa en C puede compilarse como un programa de C++, si bien hay algunas conversiones entre punteros que en C se reportaban como *warnings*, y ahora se consideran *errores*. Ver: la sección 'Algunas variante de C++' para más detalle.

También es posible mezclar código C con código C++. Sin embargo no sólo se efectuaron variaciones en el lenguaje sino también en que se implementa el llamado a una función o la definición de una variable a nivel compilador. Este hecho obliga a realizar un par de especificaciones adicionales en los códigos C/C++.

Capítulo XV - Comenzando a programar en C++

Algunas variantes de C++

Como regla general todo programa en C se graba en un archivo con extensión **.c**, y todo programa en C++ en uno con extensión **.cpp**. Esta es la forma en que un compilador distingue en código C de uno en C++ en el momento de compilarlo, salvo que se indique lo contrario.

Para empezar, escribamos la clásica función en C para intercambiar el contenido de dos variables de tipo entero, y vayamos refrescando algunos conceptos:

Ejemplo 15.1: Función swap (versión incorrecta)

```
#include <stdio.h>

/* Esta funcion no va a funcionar */
void swapint (int i1, int i2)
{
    int aux;

    aux = i1;
    i1 = i2;
    i2 = aux;
}

void main(void)
{
    int a,b;

    a = 10;
    b = 20;

    printf ("A=%d, B=%d\n", a, b);

    swapint (a, b);

    printf ("A=%d, B=%d\n", a, b);
}
```

El se programa producirá la siguiente salida:

```
A=10, B=20
A=10, B=20
```

Como puede verse, el contenido de las variables **a** y **b** no ha sido intercambiado. ¿Qué ha sucedido? Recordando un poco de C, será claro que, tal como está definida la función *swapint()*, la misma recibirá una **copia** de los parámetros, y no los punteros. Estos serán intercambiados dentro de la función, pero no en la función que la llamó. Para solucionar este inconveniente, la única solución es pasar los punteros a las variables:

Ejemplo 15.2: Función swap versión C

```
#include <stdio.h>

/* Funcion para intercambiar variables */
void swapint (int *i1, int *i2)
{
    int aux;      /* Valor auxiliar */

    aux = *i1;
    *i1 = *i2;
    *i2 = aux;
}

void main(void)
{
    int a,b;

    a = 10;
    b = 20;

    printf ("A=%d, B=%d\n", a, b);

    swapint (&a, &b); /* Intercambiar el contenido de las variables */

    printf ("A=%d, B=%d\n", a, b);
}
```

Notar que para solucionar esto, debió modificarse tanto la llamada a la función como todo el uso de las variables en la función. Cada vez que se quiera trabajar con el contenido de la variable en la función, deberá anteponerse en * (asterisco). Esto resulta bastante molesto, y puede llevar a confusión. Para solucionar esto, en C++ agrega una nueva forma de pasar los parámetros a las funciones (manteniendo las ya existentes, por supuesto, recordemos que todo programa C sigue siendo válido en C++). Es el método conocido como ***pasaje por referencia***.

Escribamos ahora el mismo programa en C++:

Ejemplo 15.3: Función swap y el pasaje de parámetros por referencia

```
#include <stdio.h>

void swapint (int &i1, int &i2)
{
    int aux;

    aux = i1;
    i1 = i2;
    i2 = aux;
}

void main(void)
{
    int a,b;

    a = 10;
    b = 20;
```

```
printf ("A=%d, B=%d\n", a, b);

swapint (a, b); // Pasaje de parámetros por referencia, igual al pasaje por
valor.

printf ("A=%d, B=%d\n", a, b);
}
```

Nótese la forma en que se definió la función *swapint()*, utilizando el símbolo **&** en vez de *****. Esto sería similar a un pasaje por referencia en el lenguaje PASCAL. Tanto en el llamado a la función como dentro de la misma no es necesario hacer ninguna referencia a puntero. Pero C++ va más allá. También es posible **devolver** una variable por referencia (ver próximo ejemplo). Esto hace que esta nueva variante del lenguaje no sólo facilite la programación, sino también permita resolver problemas de formas que antes no era posible.

Notar la presencia de comentarios con *//*. Este es un nuevo tipo de comentarios en C++. El comentario comienza con *//*, y termina con el fin de línea.

Veamos ahora un ejemplo de devolución de variables por referencia:

Ejemplo 15.4: Devolución de parámetros por referencia

```
#include <stdio.h>

int& funcion (int &x)
{
    x++;
    return x;
}

void main(void)
{
    int a,b,c;

    a = 10;
    b = funcion (a);
    funcion (c) = 30;

    printf ("a=%d b=%d c=%d\n", a,b,c);
}
```

Este programa producirá la siguiente salida

```
a=10 b=11 c=30
```

Prestemos atención a las dos llamadas a función. En el primer caso, la llamada a **funcion(a)** no crea una copia de **a**, sino que pasa la variable **a** por referencia. Luego la incrementa, y la devuelve. Pero nuevamente no devuelve una copia de la variable **a**, sino la **variable misma**. Es decir, la llamada a la función es equivalente a escribir:

```
a++;
b = a;
```

Ahora bien, si la función **funcion** no devuelve una copia numérica de la variable sino la variable misma, es posible asignarle un valor. Este es el caso de la siguiente llamada a **funcion(c)**. Esta función incrementa el valor de c, cualquiera que este sea, y luego carga 30 en esta variable. Este código puede traducirse como:

```
c++;  
c = 30;
```

Más adelante se verá que esto es esencial en el desarrollo de funciones **operadores (operator)**.

Otra diferencia entre C y C++ es que asignaciones entre punteros de distinto tipo, tales como:

```
void *pa;  
char *ps;  
  
ps = pa;
```

que en C emitiría tan sólo un mensaje de advertencia (*warning*), en C++ será considerado un *error* y el programa no será compilado.

La forma correcta de hacer esto es:

```
void *pa;  
char *ps;  
  
ps = (char *)pa;
```

Otra variante de C++ es que no es obligatorio escribir la palabra reservada **void** en el encabezado de una función que no recibe ningún parámetro. Por ejemplo, la función

```
int Random (void);
```

podría definirse como:

```
int Random ();
```

Otra variante de C++, es que se permite la definición de variables luego ejecutarse instrucciones del programa. Por ejemplo, el siguiente código no sería válido, ya que **b** y **j** no se declaran al comienzo de la función. Sin embargo, con el objetivo de flexibilizar el código, esto sí es un programa válido en C++.



Ejemplo 15.5: Algunas variantes en C++

```
#include <stdio>  
  
void main (void)  
{  
    int a;  
    scanf ("%d", &a);  
  
    int b = a;  
    for (int j = 0; j < a; ++j)  
        printf ("%d", j * a);  
}
```

Ya que pueden definirse variables en cualquier punto del programa, también podría escribirse una línea como la siguiente:

```
...
unsigned largo = strlen (s);
...
```

Este tipo de definiciones, donde la definición de una variable está junto con la llamada a una función que le da su valor inicial recibe a veces el nombre de inicialización dinámica.

Notar también la sentencia **#include<stdio.h>**. En la misma no se incluye el **.h**. En C++, el **".h"** es opcional, si bien rara vez se lo omite, principalmente porque si bien forma parte del standard, muchos compiladores aún no lo soportan.

Nuevos tipos de datos

Si bien **no forma parte del standard ANSI C++**, la mayoría de los compiladores C++ incluye una nueva forma de declarar los enteros, especificando la cantidad de bits que estos utilizan. Se definen así los tipos de datos:

<code>__int8</code>	Entero de 8 bits
<code>__int16</code>	Entero de 16 bits
<code>__int32</code>	Entero de 32 bits
<code>__int64</code>	Entero de 64 bits

Puede anteponerse la palabra reservada `unsigned` para crear tipos enteros sin signo.

En 1998, se standardizó el lenguaje de programación C++, y se definió el tipo de dato booleano (**bool**), y las definiciones **true** y **false**. El nuevo tipo de dato sólo puede tomar los valores **true** (1) o **false** (0). Códigos como:

```
bool bValor;
bValor = 1;
```

pueden emitir un mensaje de advertencia (warning) al ser compilados. El código correcto es:

```
bValor = true;
```

Algunos compiladores anteriores a esta fecha, definen el tipo de dato `BOOL` utilizando `typedef`, como sigue:

```
typedef unsigned BOOL;
#define TRUE 1
#define FALSE 0
```


Capítulo XVI - Programación orientada a objetos

PROGRAMACIÓN ORIENTADA A OBJETOS

La programación orientada a objetos (conocida como POO u OOP) es una nueva concepción en cuanto a la forma de tratar un problema. Fue desarrollada con el objetivo de adecuarse al creciente tamaño y complejidad de los programas. La misma no se contrapone a la programación estructurada (como lo hacía esta con la programación imperativa), sino que constituye una extensión de la misma. De esta forma, todos los programas y librerías escritas en C continúan siendo válidas en C++. Sin embargo la programación orientada a objetos constituye una forma de pensar y concebir la programación completamente nueva, como un conjunto de entidades abstractas independientes, razón por la cual no se podrá aprovechar toda la potencia de la POO hasta no se haya asimilado esta nueva filosofía de programación.

Las características fundamentales de la programación orientada a objetos, común a todos los lenguajes de programación son:

Objetos

Un objeto es una entidad lógica que contiene datos y código que permite manipularlo. Dicho de otra forma, es una estructura que contiene datos (variables), y funciones para acceder a los mismos, es decir que los datos no son accesibles directamente por ninguna otra parte del programa, sino mediante las funciones escritas a tal fin. Esto se conoce con el nombre de encapsulación. La ventaja de esto es que permite al programador abstraerse de la forma en que los datos son almacenados o procesados, e incluso permite que las características de los mismos, la forma en que se los almacena, etc, puedan sean modificados sin tener que efectuar cambios en el resto del programa.

Como una forma de comenzar a cambiar la forma de pensar un programa, las variables y funciones de los objetos pasarán a llamarse atributos y métodos de los mismos, respectivamente.

Polimorfismo

Básicamente polimorfismo significa que un nombre (nombre de función) puede ser utilizado para diversos fines, dependiendo del contexto donde se lo utilice. Esto permite al programador no tener que memorizar una gran cantidad de nombres funciones. Un ejemplo de esto son las funciones `atoi()`, `atol()` y `atof()`, que convierten un texto ascii a entero corto, entero largo y punto flotante respectivamente.

Herencia

La herencia es un proceso mediante el cual un objeto puede adquirir las características de otro y otros, permitiendo así el almacenamiento de conocimiento o información con una estructura jerárquica. Por ejemplo, un león es un concepto abstracto o clase que forma parte de la clase más grande felino, que a su vez forma parte de la clase mamífero y esta de la clase animal. Si tuviésemos que representar las características o definir el comportamiento de animales, sería muy útil hacerlo con un esquema similar a este. Como veremos a lo largo de este libro, muchos problemas pueden plantearse de esta forma, lo cual facilita enormemente la programación, y permite generar códigos sumamente claros y ordenados.

LAS CLASES

Las clases (**class**) son la estructura fundamental de los objetos. Son similares a una estructura (**struct**) en C, sólo que también pueden contener funciones. De hecho, en C++ las estructuras (**struct**) también pueden contener funciones, por lo que ambas son similares, con la salvedad de que por defecto, en las structs las variables y funciones son públicas, en tanto que en las classes son privadas, y que no soportan algunas características de C++, como por ejemplo la herencia. Ya veremos qué quiere decir esto más adelante.

La clase más simple se define como sigue:

```
class nombre_de_la_clase
{
    funciones y variables de la clase.
};
```

Notar el punto y coma (;) al final de la declaración de la clase. La omisión del mismo suele ser un error muy frecuente.

Las variables y funciones dentro de la clase se crean tal como se lo haría en un programa en C. Pero estas variables y funciones son privadas a la clase, esto es, sólo pueden ser accedidas por funciones de la clase.

Como una forma de cambiar la filosofía de programación, las variables dentro de una clase reciben en C++ el nombre de **atributos de la clase**, en tanto que las funciones pasarán a llamarse ahora **métodos de la clase**.

Para que un atributo o método de la clase pueda ser accedida desde afuera de la misma, se crearon tres niveles de seguridad o de protección de la información: **private** (privadas), **protected** (protegidas) y **public** (públicas).

Los atributos y métodos de tipo **private** sólo pueden ser accedidas por funciones de la clase.

Los atributos y métodos de tipo **protected** sólo pueden ser accedidas por la clase y por las clases que la hereden. Ya se verá el concepto de herencia más adelante.

Los atributos y métodos de tipo **public** pueden ser accedidas libremente. Las variables se definen en una clase.

Los atributos se crean como **private**, **public** o **protected**, definiendo poniendo este título seguido de dos puntos (:) antes de la definición de la variable.

La idea subyacente de la programación orientada a objetos es que en ningún momento se acceda a los atributos de una clase, sino mediante métodos (funciones) de la misma. Esto hace posible que si en un futuro se quisiese cambiar el tipo de variable o la forma en que estas se manipulan, sólo haya que modificar el objeto, y que el resto del programa siga funcionando con el nuevo objeto, sin haber tenido que realizar

ninguna modificación. Incluso, debería ser posible recompilar el programa en un nuevo sistema operativo, totalmente diferente, y que el mismo funcione correctamente, simplemente cambiando las funciones de los objetos correspondientes. A continuación veremos un primer ejemplo de programación en C++.

Creación de la estructura Pila.

Notar el nombre que recibe la clase, y que es el que habitualmente utiliza **Borland®**: el nombre de toda clase comienza con la letra 'T' de *Type* (tipo), seguida por el nombre del tipo, con la primer letra en mayúscula y el resto en minúsculas. **Microsoft®** utiliza un esquema similar, pero en vez de utilizar la letra 'T' utiliza la letra 'C' de *Class*.

Ejemplo 16.1: Objeto Pila (primer versión)

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

#define PILA_MAX 100

class TPila {
private:
    unsigned * Pila;
    unsigned Total;
public:
    int Inicializar (void)
    {
        Total = 0;
        Pila = (unsigned *)malloc (PILA_MAX * sizeof (unsigned));
        if (!Pila)
        {
            printf ("Memoria insuficiente");
            return -1;
        }
        printf ("La pila fue inicializada.\n");
        return 0;
    }
    void Destruir (void)
    {
        if (Pila)
            free (Pila);
        printf ("La pila fue destruida.\n");
    }
    void Encolar (unsigned Word)
    {
        if (Total<PILA_MAX)
        {
            printf ("Encolando %u\n", Word);
            Pila[Total++]=Word;
        }
        else
            printf("Pila esta llena.\n");
    }
    unsigned Desencolar (void)
    {
        if (Total>0)
        {
            printf ("Desencolando %u\n", Pila[--Total]);
            return Pila[Total];
        }
    }
};
```

```
    }
    else
    {
        printf("Pila esta llena.\n");
        return 0;
    }
}

};

void main (void)
{
    /* Crear la pila */
    TPila Pila;
    /* Inicializar la pila */
    Pila.Inicializar ();

    /* Encolar un par de datos */
    Pila.Encolar (10);
    Pila.Encolar (5);

    /* Desencolar los datos */
    Pila.Desencolar ();
    Pila.Desencolar ();

    /* Destruir la pila */
    Pila.Destruir();
}
```

El programa mostrará la siguiente salida:

```
La pila fue inicializada.
Encolando 10
Encolando 5
Desencolando 5
Desencolando 10
La pila fue destruida.
```

Este ejemplo crea una estructura pila, que permite almacenar datos de tipo entero. Notar como se definen las variables de la clase como privadas (**private:**), y las funciones que ván a ser utilizadas por el programa como públicas (**public:**). En realidad, salvo que se indique lo contrario, las variables y funciones en una clase son privadas, por lo que el **private:** está de más, pero de todas formas suele ponérselo por claridad.

En ejemplo puede verse como se crean los atributos (variables) como privadas, y se definen métodos (funciones) para acceder a los datos.

En cuanto al programa, hay dos preguntas iniciales para hacer.

1 - ¿No se podrían definir los métodos de la clase fuera de la definición de la misma?

2 - Siempre que se cree la pila será necesario inicializarla. ¿No se puede hacer esto automáticamente?

Respuestas

1 - Sí. Declarar una función dentro de una clase puede ser muy incómodo y molesto, sobre todo cuando las mismas son demasiado largas. Los métodos pueden definirse fuera de la clase, pero anteponiendo al nombre de la función el nombre de la clase, seguido de :: (doble dos puntos).

Hay una diferencia más entre estas dos variantes y es que, por defecto, las funciones definidas dentro de la clase se expanden dentro del programa al ser compiladas, no así las funciones definidas fuera de la clase.

2 - Sí. Para ello debe definirse un método que tenga el mismo nombre que la clase. Esta función se llama **constructor**, y no puede devolver datos.

Veamos nuestra versión mejorada de la pila:

Ejemplo 16.2: Objeto Pila - Constructores y Destructores

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

#define PILA_MAX 100

class TPila {
private:
    unsigned * Pila;
    unsigned Total;
public:
    TPila ();
    ~TPila ();
    void Encolar (unsigned Word);
    unsigned Desencolar (void);
};

TPila::TPila()
{
    Total = 0;
    Pila = (unsigned *)malloc (PILA_MAX * sizeof (unsigned));
    if (!Pila)
        printf ("Memoria insuficiente.\n");
    else
        printf ("Pila incializada.\n");
}

TPila::~~TPila ()
{
    if (Pila)
        free (Pila);
    printf ("La pila ha sido destruida.\n");
}

void TPila::Encolar (unsigned Word)
{
    if (!Pila)
    {
```

```
        printf("No se pudo inicializar la pila.\n");
        return;
    }
    if (Total<PILA_MAX)
    {
        printf ("Encolando %u\n", Word);
        Pila[Total++]=Word;
    }
    else
        printf("Pila esta llena.\n");
}

unsigned TPila::Desencolar (void)
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return 0;
    }

    if (Total>0)
    {
        printf ("Desencolando %u\n", Pila[--Total]);
        return Pila[Total];
    }
    else
    {
        printf("Pila esta llena.\n");
        return 0;
    }
}

void main (void)
{
    /* Crear la pila */
    TPila Pila;

    /* Encolar un par de datos */
    Pila.Encolar (10);
    Pila.Encolar (5);

    /* Desencolar los datos */
    Pila.Desencolar ();
    Pila.Desencolar ();
}
```

La salida de este programa es la siguiente:

```
La pila fue inicializada.
Encolando 10
Encolando 5
Desencolando 5
Desencolando 10
La pila fue destruida.
```

Notar la definición de la función `~TPila()`. Esta función se llama **destructor** de la clase, y consiste en una función con el mismo nombre que la clase precedida por el carácter `~`, y no puede tomar ni devolver parámetros.

Nunca debe llamarse a un *constructor/destructor* de una clase desde el programa. Esto puede hacerse, pero no operará como el programador esperaría. Los constructores/destructores están pensados para operar automáticamente, no para ser llamados como un método más de la clase.

¿Cuándo se llama a la función destructora de la clase?

Rta: cuando se la deja de utilizar, es decir, en el momento en que se sale de la función donde se la definió, siempre y cuando no se la esté devolviendo en el return.

¿Pero porqué utilizar siempre el mismo tamaño de pila? ¿No se podría de alguna manera indicar la cantidad de objetos máximo que tendrá la pila al crearla?

Rta: Sí. El constructor puede tomar parámetros, al igual que cualquier función.



Los constructores son métodos especiales, pensados para ser llamados automáticamente por el compilador. Puede llamarse a los mismos desde el programa, o desde métodos de la clase, pero estos no operarán en la forma esperada. Por esta razón, nunca deben ser llamados desde el programa.

Veamos un ejemplo:



Ejemplo 16.3: Objeto Pila - Constructor parametrizado

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

#define PILA_MAX 100

class TPila {
private:
    unsigned * Pila;
    unsigned Total;
    unsigned Max;
public:
    TPila (unsigned Maximo);
    ~TPila ();
    void Encolar (unsigned Word);
    unsigned Desencolar (void);
};

TPila::TPila(unsigned Maximo)
{
    Total = 0;
    Max = Maximo;
    Pila = (unsigned *)malloc (PILA_MAX * sizeof (unsigned));
    if (!Pila)
        printf ("Memoria insuficiente.\n");
    else
        printf ("Pila incializada.\n");
}
```

```
TPila::~~TPila ()
{
    if (Pila)
        free (Pila);
    printf ("La pila ha sido destruida.\n");
}

inline void TPila::Encolar (unsigned Word)
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return;
    }
    if (Total<Max)
    {
        printf ("Encolando %u\n", Word);
        Pila[Total++]=Word;
    }
    else
        printf("Pila esta llena.\n");
}

unsigned TPila::Desencolar (void)
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return 0;
    }

    if (Total>0)
    {
        printf ("Desencolando %u\n", Pila[--Total]);
        return Pila[Total];
    }
    else
    {
        printf("Pila esta llena.\n");
        return 0;
    }
}

void main (void)
{
    /* Crear la pila */
    TPila Pila(100);

    /* Encolar un par de datos */
    Pila.Encolar (10);
    Pila.Encolar (5);

    /* Desencolar los datos */
    Pila.Desencolar ();
    Pila.Desencolar ();
}
```


Veamos ahora otro ejemplo de esta misma pila:

Ejemplo 16.4: Objeto Pila - Parámetros por defecto - new y delete

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

#define PILA_MAX 100

/* IMPORTANTE: No pasar una variable de tipo TPila como parámetro a una función
*/
class TPila {
private:
    unsigned * Pila;
    unsigned Total;
    unsigned Max;
public:
    TPila (unsigned MaxPila = PILA_MAX);
    ~TPila ();
    void Encolar (unsigned Word);
    unsigned Desencolar (void);
};

TPila::TPila(unsigned MaxPila)
{
    Total = 0;
    Max = MaxPila;
    Pila = new unsigned[Max];

    if (!Pila)
        printf ("Memoria insuficiente.\n");
    else
        printf ("Pila incializada.\n");
}

TPila::~~TPila ()
{
    if (Pila)
        delete []Pila;
    printf ("La pila ha sido destruida.");
}

void TPila::Encolar (unsigned Word)
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return;
    }
    if (Total<Max)
    {
        printf ("Encolando %u\n", Word);
        Pila[Total++]=Word;
    }
    else
        printf("Pila esta llena.\n");
}

unsigned TPila::Desencolar (void)
```

```
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return 0;
    }

    if (Total>0)
    {
        printf ("Desencolando %u\n", Pila[--Total]);
        return Pila[Total];
    }
    else
    {
        printf("Pila esta llena.\n");
        return 0;
    }
}

void main (void)
{
    /* Crear la pila */
    TPila Pila;

    /* Encolar un par de datos */
    Pila.Encolar (10);
    Pila.Encolar (5);

    /* Desencolar los datos */
    Pila.Desencolar ();
    Pila.Desencolar ();
}
```

Notar como se reemplazo en el constructor, la línea

```
Pila = (unsigned *)malloc (sizeof (unsigned) * max );
```

por

```
Pila = new unsigned[max];
```

y en el destructor, la línea

```
free (Pila);
```

por

```
delete []Pila;
```

new y **delete** son palabras reservadas de C++, y cumple la misma función que **malloc** y **free**. En ambos casos, las dos líneas son similares, salvo por el hecho de que new y delete llaman automáticamente al constructor/destructor de la clase, además de poder sobrecargarse como cualquier operador. Puede seguir utilizándose malloc y free, sin embargo en C++ la convención es utilizar únicamente **new** y **delete**, por ser más simples y adaptarse más a la filosofía de objetos. Nunca debe liberarse con **delete** un puntero alocado con **malloc** o liberar con **free** un puntero reservado con **new**.

Notar que en la llamada a **delete** se escribieron dos corchetes []. Esto se debe a que en la memoria que se va a liberar, no se encuentra almacenado un único dato, sino una determinada cantidad de los mismos, es decir, se trata de un vector. El [] indica que debe ejecutarse más de un destructor (uno para cada uno de los elementos de vector).

```
Dato = new TDato;
```

es similar a:

```
Dato = malloc (sizeof (TDato));  
Dato.TDato();
```

y

```
Dato = new TDato[cantidad];
```

es similar a:

```
Dato = malloc (sizeof (TDato)*cantidad);  
for (i = 0; i < cantidad; ++i)  
    Dato[i].TDato();
```

y

```
delete Dato;
```

es similar a

```
Dato.~TDato();  
free (Dato);
```

y

```
delete []Dato;
```

es similar a

```
for (i = 0; i < cantidad; ++i)  
    Dato[i].~TDato();  
free (Dato);
```

(Recordar que si se trata de un puntero a un vector de objetos, debe anteponerse [] al nombre del puntero, en la llamada a delete. (delete []Dato).

En la definición de la función Encolar() se utilizó la palabra reservada **inline**:

```
inline void TPila::Encolar (unsigned Word)
```

Esta obliga al compilador a expandir el código de la función dentro del código, en vez de realizar la llamada a la función. Esto aumentará el tamaño del código, por lo cual sólo debe ser utilizado para funciones pequeñas o cuya eficiencia sea muy importante.

Notar la advertencia al comienzo de la función de que no se puede pasar una variable de tipo TPila como parámetro a una función (ni tampoco puede ser devuelto por una función). Es decir, cosas como:

```
/* Este programa está mal */
void Func (TPila PilaCopia)
{
    ...
}

void main (void)
{
    TPila Pila;
    Func (Pila)
}
```

constituyen un error de programación. Lo que ocurre es que al llamar a la función Func() se creará una copia de la variable Pila (PilaCopia), y al salirse de la función Func() la misma será destruida. Pero el destructor de PilaCopia liberará la memoria de la variable PilaCopia, entre ellos el puntero Pila de esta variable, **¡¡¡que es el mismo que el de la variable Pila!!!**, con lo cual al regresar a la función main, el puntero Pila.Pila ya habría sido liberado. Finalmente cuando al salir de la función main() se libere nuevamente la memoria se producirá un error fatal.

Importante

Se verá como resolver este problema en la sección UTILIZACIÓN DE MEMORIA DINÁMICA EN LAS CLASES. Como regla general, **NUNCA UTILIZAR ALOCACIÓN/LIBERACIÓN DE MEMORIA** (malloc-free / new-delete) **DENTRO DE CONSTRUCTORES/DESTRUCTORES** de objetos, hasta no haber leído la sección.

Capítulo XVII - Parámetros por defecto y sobrecarga de funciones

PARÁMETROS POR DEFECTO

Otro punto a notar es la definición del constructor de la clave:

```
TPila (unsigned MaxPila = PILA_MAX);
```

¿Qué significa que la variable esté igualada a un valor?

Pues simple, que una función puede tomar valores por defecto. Esto hace posible no especificar valores al llamar a la función, en cuyo caso se utilizarán los valores por defecto, definidos en el encabezado de la función. A esto se llama parámetros por defecto y puede utilizarse en cualquier función.

La única condición que impone C++ es que las variables que tienen valores por defecto deben estar del lado derecho de la función, y que al llamar a la función, no puede especificarse ningún otro parámetro a partir del primero que no se especifique. Por ejemplo, puede tenerse algo como:

```
funcion (int a, int b, int c=1, int d=2, int e=3);
```

pero no algo como

```
funcion (int a, int b, int c=1, int d=2, int e);
```


y no hay forma de especificar el parámetro 'e' si no se especifican previamente 'c' y 'd'.

SOBRECARGA DE FUNCIONES

Una de las características muy útiles de C++ es la sobrecarga de funciones, esto es, definir varias funciones con el mismo nombre. Por ejemplo, en C, se tienen las funciones `atoi()`, `atol()`, y `atof()`, para convertir un texto en formato numérico. Pero ya que existen 3 tipos de datos numéricos, se deben definir tres funciones, según el tipo de dato al que deba ser traducido el texto. Pero esto obliga al programador a memorizar tres funciones.

Para evitar este tipo de problemas, en C++ se pueden definir varias funciones con el mismo nombre, y el compilador decidirá en el momento de compilar a qué función debe llamarse, según la forma en que se la llame. A esto se conoce también con el nombre de polimorfismo.

Veamos el siguiente ejemplo de nuestra clase `TPila`:

 **Ejemplo 17.1:** Objeto pila - sobrecarga de funciones y polimorfismo

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

```
#include <iostream.h>

#define PILA_MAX 100

class TPila {
private:
    unsigned * Pila;
    unsigned Total;
    unsigned Max;
public:
    TPila (unsigned MaxPila=PILA_MAX);
    ~TPila ();
    void Encolar (unsigned Dato);
    void Encolar (unsigned Dato1, unsigned Dato2);
    unsigned Desencolar (unsigned Cantidad=1);
};

TPila::TPila(unsigned MaxPila)
{
    Total = 0;
    Max = MaxPila;
    Pila = new unsigned[Max];
    if (!Pila)
        printf ("Memoria insuficiente.\n");
    else
        printf ("Pila incializada.\n");
}

TPila::~~TPila ()
{
    if (Pila)
        delete []Pila;
    printf ("La pila ha sido destruida.");
}

void TPila::Encolar (unsigned Dato)
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return;
    }

    cout <<"Encolando " <<Dato<<"\n";

    if (Total<PILA_MAX)
        Pila[Total++]=Dato;
    else
        printf("Pila esta llena.\n");
}

void TPila::Encolar (unsigned Dato1, unsigned Dato2)
{
    Encolar (Dato1);
    Encolar (Dato2);
}

unsigned TPila::Desencolar (unsigned Cantidad)
{

```

```
if (!Pila)
{
    printf("No se pudo inicializar la pila.\n");
    return 0;
}
if (Cantidad==0)
    return 0;

if (Total>0)
{
    if (Cantidad>1)
        Desencolar (Cantidad-1);

    cout <<"Desencolando "<<Pila[--Total]<<"\n";
    return Pila[Total];
}
else
{
    printf("Pila esta vacia.\n");
    return 0;
}
}

void main (void)
{
    TPila Pila(10);
    Pila.Encolar (10);
    Pila.Encolar (20,30);

    Pila.Desencolar ();
    Pila.Desencolar (2);
}
```

Para empezar, en la tercer línea del programa se incluye el archivo `iostream.h`. Este archivo define la entrada/salida standard en C++. Aquí se define, para empezar, la función `cout`. La forma de utilizar esta función es simple: **`cout <<`** parametros separados por '**`<<`**'. Puede seguir utilizándose **`printf`**, pero **`cout`** está más cercana a la filosofía de C++. Lo primero que se nota es que no es necesario especificar el tipo de dato que se va a imprimir. Más adelante se verá como definir funciones de este tipo.

por ejemplo, para imprimir un número de punto flotante, que con `printf` se haría como:

```
printf ("Valor = %f\n", valor);
```

se escribiría como:

```
cout << "Valor = " << valor << "\n";
```

Notar como no es necesario especificar el tipo de dato, pero tampoco es posible especificar la forma en que este dato se debe imprimir, como se podría hacer con `printf`

Puede verse además que existen dos funciones `Encolar()`, ambas con el mismo nombre, y que incluso, una llama a la otra. Esto permite que el programador no tenga que memorizar dos nombres, sino uno solo. A qué función se llama en cada caso? Rta: muy simple: los nombres de las funciones son los mismos, pero los parámetros que reciben no, por lo tanto, el tipo y la cantidad de parámetros que recibe la función especifica a qué función se llama. Dicho de otra forma, una función no se identifica por su nombre, como en C, sino también por los parámetros que recibe.

Notese también que la función `desencolar` está definida como:

```
unsigned Desencolar (unsigned Cantidad=1);
```

En este caso, el parámetro `Cantidad` especifica la cantidad de datos a desencolar. Ya que generalmente se desea desencolar un único dato, puede no especificárselo, en cuyo caso, la variable tomará el valor por defecto, que se especificó en la función. Recordar que pueden existir varios parámetros con valores por defecto, pero es condición obligatoria que todos ellos se encuentren del lado derecho de la definición de la función.

La sobrecarga de funciones puede conducir a algunos casos de ambigüedad, donde el compilador no sepa a qué función debe llamarse, los cuales serán reportados indefectiblemente como errores, aún cuando para el programador pueda ser indistinto. Por ejemplo, imaginemos un caso como el siguiente:

```
class TEjemplo
{
    public:
        Funcion (int a);
        Funcion (int a, int b=1);
}

{
    ...
    class TEjemplo Ej;
    Ej.Funcion (1);
}
```

En este caso, no hay forma de saber a cuál de las dos funciones ***Funcion*** se desea hacer referencia, y por lo tanto la línea en negrita no puede ser compilada.

Existe también una función llamada `cin`, similar a `scanf()`. Por ejemplo, para leer el valor de un entero, lo que normalmente se haría como:

```
int a;
scanf ("%d", &a);
```

ahora puede escribirse como

```
cin >>a;
```

Capítulo XVIII - Herencia

HERENCIA

Algo común en programación, es que se trabaje con tipos o estructuras de datos parecidas, o basadas unas en otras. Ver la sección *Programación Orientada a Objetos* para más información. Por ejemplo, una estructura de datos muy simple es la ***lista***. Las estructuras de datos conocidas como ***pila*** y ***cola*** no son más que listas en donde se insertan elementos al comienzo y se los saca del comienzo/final o viceversa. Si se quisiese implementar estos tipos de datos, no tiene sentido escribir tres tipos de datos, que serán evidentemente muy similares. Para evitar esto, C++ permite definir clases que hereden a su vez otros tipos

de datos. La clase más simple recibe el nombre de **clase base**, y la clase heredera se denomina **clase derivada**.

Veamos un ejemplo: contruyamos el objeto TLista. Por simplicidad, la lista será usada únicamente para almacenar enteros, con un par de funciones para insertar y quitar datos en cualquier posición de la lista y una función que devuelva el total de datos en el la misma.

Ejemplo 18.1: Objeto Lista

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

#define MAX 100 // Maxima cantidad de elementos que se almacenaran
typedef unsigned TData; // Tipo de dato que se almacenara en la lista/pila/cola

/***** Definicion del objeto TLista *****/
class TLista
{
    private:
        TData Datos[MAX];
        TData Aux;
        unsigned Total;
    public:
        TLista (void)
        { Total = 0; }
        int Insertar (unsigned Pos, TData Dato);
        TData Sacar (int Pos);
        unsigned LeerTotal (void)
        {
            return Total;
        }
};

int TLista::Insertar (unsigned Pos, TData Dato)
{
    // Verificar si queda espacio en la lista
    if (Total>=MAX)
    {
        cout<<"No se pueden almacenar mas datos.\n";
        return -1;
    }
    // Verificar la posicion donde se quiere insertar el dato
    if (Pos>Total)
    {
        cout<<"Se quieren insertar datos en una posicion muy elevada.\n";
        return -1;
    }
    // Mover los datos hacia posiciones mas altas en el array
    memmove (Datos+Pos+1, Datos+Pos, (Total-Pos)*sizeof (TData));

    // Almacenar el dato
    Datos[Pos] = Dato;

    // Incrementar la cantidad de datos en el array
    ++Total;
}
```

```
// Retornar OK;
return 0;
}

TDato TLista::Sacar (int Pos)
{
    TDato Ret;

    // Verificar que el dato a retornar exista
    if (Pos < 0 || Pos>=Total)
    {
        cout<<"No existe el dato en la lista.\n";
        return Aux;
    }

    // Copiar el dato a retornar
    Ret = Datos[Pos];

    // Mover los datos de posiciones mas altas a posiciones mas bajas dentro
    // del array.
    memmove (Datos+Pos, Datos+Pos+1, (Total-Pos-1)*sizeof (TDato));

    // Decrementar la cantidad de datos en el array
    --Total;

    return Ret;
}
```

Definamos ahora los objetos pila y cola:

Ejemplo 18.2: Objetos Pila y Cola - Herencia

```
/****** Definicion del objeto TPila *****/
class TPila:public TLista
{
public:
    int Push (TDato Dato);
    TDato Pop (void);
};

int TPila::Push (TDato Dato)
{
    return Insertar (LeerTotal(), Dato);
}

TDato TPila::Pop (void)
{
    return Sacar (LeerTotal()-1);
}

/****** Definicion del objeto TCola *****/
class TCola:public TLista
{
public:
    int Encolar (TDato Dato);
}
```

```

        TDato LeerCola (void);
};

int TCola::Encolar (TDato Dato)
{
    return Insertar (LeerTotal(), Dato);
}

TDato TCola::LeerCola (void)
{
    return Sacar (0);
}

/***** Funcion main() *****/
void main (void)
{
    TCola Cola;
    TPila Pila;

    cout << "Operaciones con la pila\n";
    Pila.Push(10);
    Pila.Push(20);
    cout << Pila.Pop() << "\n";
    cout << Pila.Pop() << "\n";

    cout << "Operaciones con la cola\n";
    Cola.Encolar(30);
    Cola.Encolar(40);
    cout << Cola.LeerCola() << "\n";
    cout << Cola.LeerCola() << "\n";
}

```

Veamos ahora la definición de los objetos Pila y Cola.

```

class TPila:public TLista
{
    ...
}

```

La definición de la clase es similar a las ya vistas, con la salvedad de **public TLista**. Esto lo que hace es indicar que la clase **TPila** debe heredar en forma pública todos los atributos y métodos de la **clase TLista**.

Al igual que los atributos y los métodos, la forma en que los mismos son heredados por las clases herederas puede ser **public**, **protected** y **private**. El siguiente recuadro indica el tipo de atributo/método que se tendrá en una clase heredera.

Tipos de herencia

Tipo de dato	Herencia public	Herencia protected	Herencia private
Private	no son accesibles	no son accesibles	no son accesibles
Protected	protected	protected	private
Public	public	protected	private

En caso se no especificarse el tipo de herencia, se supone por defecto herencia de tipo **private**.

Por ejemplo, si una clase hereda los datos de su padre en forma **private**, los atributos y métodos de tipo **protected** del padre pasarán a ser un atributos y métodos de tipo **private** en la clase heredera.

La forma en que se heredan los atributos y los métodos puede parecer un tanto confuso, pero con el uso se va aclarando.

Las variables que no se heredan siguen existiendo, pero no son accesibles desde la clase heredera. Veamos un ejemplo:

Ejemplo 18.3: Objeto Figura - Herencia

```
class TFigura
{
    private:
        unsigned Color;
    public:
        SetColor (unsigned Col)
            { Color = Col; }
        GetColor (unsigned Col)
            { return Color; }
};

class TCuadrado:public TFigura
{
    private:
        unsigned Lado;
    public:
        SetLado (unsigned L)
            { Lado = L; }
};
```

Si ahora creásemos una variable de tipo **TCuadrado**, esta tendría dos atributos: **Color** y **Lado**, pero sólo **Lado** será visible desde la misma y sólo podrá accederse al atributo Color mediante los métodos **SetColor** y **GetColor**, que serán privados en la clase **TCuadrado**. Se verá más a cerca de herencia en la sección III - Programación avanzada en C++.

Introducción a la herencia múltiple

Una misma clase puede construirse como derivada de más de una clase base. Las clases base de una clase derivada se especifican separadas por comas. Por ejemplo, para definir una clase D como derivada de tres clases base A, B y C debe escribirse:

```
class D : public A, private B, private C
{ ... };
```

En este caso la clase D heredará en forma pública la clase A y en forma privada las clases B y C. No hay límite en el número de clases que pueden heredarse. La herencia múltiple puede llevar a casos más complejos cuando una misma clase es heredada más de una vez. Para más información ver la sección UN POCO MÁS SOBRE HERENCIA - HERENCIA MÚLTIPLE

Distribución en memoria de atributos, en clases herederas

Supongamos que se tenga un caso como el siguiente:

Ejemplo 18.4: Herencia múltiple

```
class A {  
    int a;  
};  
  
class B {  
    int b;  
};  
  
class C: public A, public B {  
    int c;  
};
```

Si ahora se crease un objeto de tipo **C**, en la mayoría de los casos, los compiladores almacenan la variable **a**, luego la **b** y finalmente la **c**, pero esto no está especificado en ningún lugar ni forma parte de la definición del lenguaje ANSI C++, por lo que no tiene por qué ser así. Es decir que el orden de las variables en memoria en un objeto de tipo C podría ser:

```
a - b - c  
b - c - a
```

o cualquier otro.

Capítulo XIX - Casting de objetos

Casting de objetos

Volvamos al caso anterior en que se tienen tres tipos de objetos, uno de los cuales se contruye como derivado de los dos anteriores. Supongamos que ahora tengamos tres punteros que apunten a variables de estos tres tipos:

Ejemplo 19.1: Casting de objetos

```
A *pa;  
B *pb;  
C *pc;
```

Es posible en C++ obtener tan sólo la parte B de un objeto C, simplemente asignando un puntero de tipo C a un puntero de tipo B, es decir, escribiendo:

```
pb = pc;
```

Desde ya que tiene sentido escribir ***pb = pc***, pero no ***pc = pb***;

Volviendo a la línea ***pb=pc***, ***pb*** apuntará ahora a la parte ***B*** del objeto apuntado por ***pc***, y por lo tanto el valor de ***pb*** no tiene por qué ser el mismo que el de ***pc*** (y normalmente no lo será), luego de efectuar la operación. Por ejemplo:

```
/* Este código es incorrecto */  
pb = pc;  
  
if (pb != NULL)  
    Funcion (pb);
```

Así como está, la línea en negrilla es incorrecta o al menos incompleta (si bien completamente compilable). ***pb*** tomará en valor de ***pc***, más el desplazamiento de la parte ***B*** dentro de ***C***. Si ***pc*** fuese ***NULL***, ***pb*** tomará el valor del offset o posición de la parte ***B*** dentro de ***C***, el cual no tiene por qué ser cero. Es común utilizar la notación ***pb = pc + delta (b)***, para describir esta situación. La forma correcta de escribir la línea (y la que generalmente se utiliza) es:

```
pb = (pc == NULL) ? NULL : pc;
```

lo cual es equivalente a:

```
if (pc == NULL)  
    pb = NULL;  
else  
    pb = pc;
```

Nota

Esta situación aquí descrita es un tanto confusa. Ciertos compiladores asignan a la variable de la clase base el valor **NULL** en vez de su offset dentro del objeto, cuando el puntero del lado derecho de la igualdad tiene el valor NULL, es decir, automáticamente se ejecuta

```
pb = (pc==NULL) ? NULL : pc;
```

Tipos de casting entre objetos

Es posible para el programador efectuar distintos tipos de casting entre clases base y derivadas de un mismo objeto. Existen para esto las palabras reservadas: **dynamic_cast**, **static_cast**, que permiten especificar la operación a realizar.

dynamic_cast realizará una verificación en tiempo de ejecución de que la operación realizada es válida, provocando una excepción de no serlo, en tanto que **static_cast** realizará la conversión sin verificar si la misma es o no segura. Desde ya que debe tratarse de clases base-derivada o viceversa, o se producirá un error de compilación. Veamos un ejemplo:

Ejemplo 19.2: dynamic_cast/static_cast

```
class B {...};
class D: public B {...};
{
    B *b = new B;
    D *d = new D;

    B *b2 = dynamic_cast <B*> d; // Conversión correcta.
                                // Se verificará la validez de la operación.
    B *b3 = static_cast <B*> d; // Conversión correcta. No se verificará la
validez                                // de la operación.

    D *b2 = dynamic_cast <B*> b2; // Conversión correcta.
                                // Se verificará la validez de la operación.
    D *b3 = static_cast <B*> b2; // Conversión correcta. No se verificará la
validez                                // de la operación.

    D *b4 = dynamic_cast <B*> b; // Conversión incorrecta.
                                // Se verificará la validez de la operación
                                // produciendo una excepción (error en tiempo
de ejecución).
    D *b5 = static_cast <B*> b; // Conversión correcta. No se verificará la
validez de                                // la operación.
}
```

Es posible realizar conversiones entre partes de objetos no directamente relacionadas, pero que formen parte de un objeto mayor. Esto se conoce con el nombre de **cross-cast**. Veamos un ejemplo:

Ejemplo 19.3: Cross-cast

```
class B {...};
```

```
class C {...};
class D: public B,C {...};
...
{
    D *d = new D;

    B *b = dynamic_cast <B*> d;    // b apuntará a la parte B de D
    C *c = dynamic_cast <C*> b;    // Conversión correcta. c apuntará a la parte C
de D (cross cast)
}
```


Los constructores y la herencia

Cabe preguntar qué ocurre con los constructores de una clase base, al crear una clase derivada. La respuesta es que siguen existiendo, y se llamarán automáticamente al crear la clase derivada. Ya que al especificar un constructor de la clase derivada no se especifican parámetros para el constructor de la clase base, únicamente se llamará por defecto (de existir) el constructor que no reciba parámetros.

Como ya dijimos, nunca debe llamarse un constructor de una clase desde un método de la misma, lo cual es válido también para un constructor de la clase derivada. Veamos un ejemplo:

Ejemplo 19.4: Los constructores y la herencia

```
class A {
    public:
        A (void) {...}
        A (int x)    {...}
};

class B {
    public:
        B (void) {...}
        B (int x)    {...}
};

class C : private A, public B {
    public:
        C (int x)
        {...}
};
```

Ahora bien, si creásemos un objeto de tipo C, se llamará automáticamente los constructores A::A(void) y B::B(void), pero nunca a A::A(int x) y B::B(int x). ¿Hay forma de hacer que se llame a estos dos últimos constructores al crear una clase de tipo C? Rta: Sí, especificándolo junto con el constructor de la clase derivada, a continuación del nombre del constructor de la misma, y antes de la declaración. Por ejemplo:

```
class C : private A, public B {
    public:
        C (int x) : A (x) , B (x/2)
        {...}
};
```

En este caso por ejemplo, al crearse el objeto **B** en la forma **B b(10)**, se llamará al constructor de **A** en la forma **A(10)** y al constructor de la clase **B** como **B(5)**.

Capítulo XX - Sobrecarga de operadores y funciones friend

OPERADORES

Ya que el ejemplo de la Pila creció demasiado, empecemos un nuevo ejemplo desde cero, y vayamos refrescando algunos conceptos. Creemos la clase complejos.

Ejemplo 20.1: Objeto Complejo

```
#include <iostream.h>

class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
        TComplejo (int Re=0, int Im=0);
};

TComplejo::TComplejo (int Re, int Im)
{
    Real = Re;
    Imag = Im;
}

void main(void)
{
    TComplejo a, b(4,3);
    TComplejo c;
}
```

Notar que en el constructor de la clase se utilizan nombres distintos para los parámetros de la función, de los nombres de las variables a cargar, es decir, se define el constructor como:

```
TComplejo::TComplejo (int Re, int Im);
```

En vez de:

```
TComplejo::TComplejo (int Real, int Imag);
```

Esto es así para poder diferenciar las variables de la clase de los parámetros de la función. Existe una forma de hacer esto utilizando la palabra reservada **this**, pero esto se verá más adelante.

Agreguemos ahora una función para sumar números complejos.

Esta función deberá tomar dos números complejos, y devolver su suma. En principio podría pensarse que esto puede hacerse como sigue:

Ejemplo 20.2: Objeto Complejo - Función Sumar

```
#include <iostream.h>

class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
        TComplejo (int Re=0, int Im=0);
};

TComplejo Sumar(TComplejo C1, TComplejo C2)
{
    TComplejo C3;

    C3.Real = C1.Real + C2.Real;
    C3.Imag = C1.Imag + C2.Imag;

    return C3;
}

TComplejo::TComplejo (int Re, int Im)
{
    Real = Re;
    Imag = Im;
}

void main(void)
{
    TComplejo a, b(4,3);
    TComplejo c;

    c = Sumar (a, b);
}
```

Pero hay un problema; la función:

```
TComplejo Sumar (TComplejo C1, TComplejo C2);
```

no pertenece a la clase TComplejo, y por lo tanto no puede acceder a los datos privados de la misma. Es decir que no podrán efectuarse las operaciones:

```
C3.Real = C1.Real + C2.Real;
C3.Imag = C1.Imag + C2.Imag;
```

Para solucionar esto, debe declararse a la función Sumar como 'amiga' (friend) de la clase:

Ejemplo 20.3: Objeto Complejo - funciones friend

```
class TComplejo
{
    private:
        int Real;
        int Imag;
```

```
public:
    TComplejo (int Re=0, int Im=0);
    friend TComplejo Sumar(TComplejo C1, TComplejo C2);
};
```

Con lo cual ahora sí se tiene un código correcto.

Agreguemos ahora otra nueva función Sumar(), que sea llamada por un número complejo, tome como parámetro otro número complejo, y devuelva el resultado de la suma, es decir, sea llamada como:

```
TComplejo a, b, c;
c = a.Sumar (b); // Quiero efectuar la operación c = a + b
```

Ejemplo 20.4: Objeto Complejo - Función Sumar (segunda variante)

```
#include <iostream.h>

class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
        TComplejo (int Re=0, int Im=0);
        TComplejo Sumar (TComplejo C);
        friend TComplejo Sumar (TComplejo C1, TComplejo C2);
};

TComplejo TComplejo::Sumar (TComplejo C)
{
    TComplejo R;

    R.Real = Real + C.Real;
    R.Imag = Imag + C.Imag;

    return R;
}

TComplejo Sumar (TComplejo C1, TComplejo C2)
{
    TComplejo C3;

    C3.Real = C1.Real + C2.Real;
    C3.Imag = C1.Imag + C2.Imag;

    return C3;
}

TComplejo::TComplejo (int Re, int Im)
{
    Real = Re;
    Imag = Im;
}

void main (void)
```

```
{
    TComplejo a (1,3), b (4,3);
    TComplejo c,d;

    c = Sumar (a,b);
    d = a.Sumar (b);
}
```

Notar como ahora se tienen dos métodos Sumar(), y el compilador sabe a cual llamar, según la cantidad de parámetros que reciben. Estos dos métodos hacen exactamente lo mismo, aunque son llamados en una forma diferente. De hecho, la segunda versión de Sumar(), aunque un tanto chocante en cuanto a la forma en que se la utiliza, tiene claras ventajas con respecto a la anterior, y por eso es fundamental prestar atención y comprender el código correctamente. La primera es que forma parte de la clase, y por lo tanto no requiere ser declarada como *friend* lo cual conviene evitar por ir en contra del encapsulamiento. La segunda, y la más importante, es que puede ser declarado como un operador. Para esto se requiere usar la palabra reservada **operator**. Veamos nuevamente el ejemplo:

Ejemplo 20.5: Objeto Complejo - operador+

```
#include <iostream.h>

class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
        TComplejo (int Re=0, int Im=0);
        friend TComplejo Sumar (TComplejo C1, TComplejo C2);
        TComplejo Sumar (TComplejo C);
        TComplejo operator+ (TComplejo C);
};

TComplejo TComplejo::Sumar (TComplejo C)
{
    TComplejo R;

    R.Real = Real + C.Real;
    R.Imag = Imag + C.Imag;

    return R;
}

TComplejo TComplejo::operator+ (TComplejo C)
{
    TComplejo R;

    R.Real = Real + C.Real;
    R.Imag = Imag + C.Imag;

    return R;
}

TComplejo Sumar (TComplejo C1, TComplejo C2)
{
    TComplejo C3;

    C3.Real = C1.Real + C2.Real;
```

```

    C3.Imag = C1.Imag + C2.Imag;

    return C3;
}

TComplejo::TComplejo (int Re, int Im)
{
    Real = Re;
    Imag = Im;
}

void main (void)
{
    TComplejo a (1, 3), b (4, 3);
    TComplejo c, d, e, f;

    c = Sumar (a, b);
    d = a.Sumar (b);
    e = a.operator+ (b);
    f = a + b;
}

```

Notar que la forma en que se definió el método **operator+** es idéntica a como se definió la segunda versión de Sumar(), y por lo tanto debe ser llamada en la misma forma. Pero por ser la función un operador, es decir por estar definida con la palabra reservada **operator**, C++ permite omitir el texto **.operator** y el paréntesis, con lo cual, solo queda el operador. C++ permite sobrecargar los siguientes operadores:

Tabla 2: Operadores sobrecargables

+	-	*	/	%	^
!	=	<	>	+=	--
^=	&=	=	<<	>>	<<=
<=	>=	&&		++	--
()	[]	new	delete	&	
~	*=	/=	%=	>>=	==
!=	,	->	->*		

Notar que **new** y **delete** son considerados operadores, y por lo tanto pueden ser sobrecargados.

Esto permite definir tipos de datos propios y operadores sobre ellos, y crear así un código que se parecerá mucho a un lenguaje de alto nivel. Por ejemplo, algo muy cómodo y común es definir un tipo de dato TString, y poder ejecutar operaciones tales como:

```

TString S1, S2, S3;

S1 = "Hola";
S2 = " mundo"
S3 = S1 + S2;

cout << S3;

```

lo cual producirá la salida:

```
Hola mundo
```

Volvamos a nuestro ejemplo de números complejos. Agreguemos una función para imprimir números complejos.

Ejemplo 20.6: Objeto complejo - sobrecarga de iostream

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

class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
        TComplejo (int Re=0, int Im=0);
        friend ostream& operator<< (ostream &stream, TComplejo C);
};

TComplejo::TComplejo (int Re, int Im)
{
    Real = Re;
    Imag = Im;
}

ostream& operator<<(ostream &stream, TComplejo C)
{
    stream << " (" << C.Real << " + i " << C.Imag << ") ";

    return stream;
}

void main (void)
{
    TComplejo a (1, 3);

    cout <<"a =" << a << '\n';
}
```

Notar la forma en que se sobrecargó el operador ***operator<<***, para poder definir la salida standard de la clase, no como miembro de la clase, sino como un operador global, "**amigo**" de la clase. Esto debe hacerse así porque el operador << deberá anteponerse a la **izquierda** de la variable de tipo complejo a mostrar, y no a la **derecha** como el resto de los operadores aritméticos. Es decir, para mostrar la variable **a** no se escribe **a<<** sino **<<a**. El manejo de streams se verá con más detalle en la sección III - Programación Avanzada en C++.

De la misma forma puede sobrecargarse el operador ++ como miembro de la clase para implementar operaciones como a++, pero para implementar el operador ++a deberá sobrecargarse el operador ++ en forma global, como amigo a la clase.

Escribamos ahora el operador +=. Este operador debe sumar un complejo a otro, dejando el resultado en el que se encuentra a la izquierda. En principio parecería que el problema se resuelve escribiendo una función como sigue:

```
void TComplejo::operator+= (TComplejo C)
{
    Real += C.Real;
```

```
    Imag += C.Imag;  
}
```

Lo cual permitiría efectuar operaciones tales como:

```
TComplejo C1, C2;  
  
C1 += C2;
```

Esto no es incorrecto, pero sí incompleto. Ya que en C/C++, al trabajar con tipos de datos convencionales, es posible escribir cosas como:

```
int a, b, c;  
  
c = a += b;
```

Esto no es posible hacerlo con nuestra función

```
void TComplejo::operator+= (TComplejo C)
```

por la simple razón de que esta función no devuelve ningún parámetro, y por lo tanto, el siguiente operador (=), que esta definido como una copia de dato, no puede acceder al dato requerido. Pero esto puede solucionarse, de dos formas:

```
TComplejo TComplejo::operator+= (TComplejo C)  
{  
    Real += C.Real;  
    Imag += C.Imag;  
  
    return *this;  
}  
  
TComplejo& TComplejo::operator+= (TComplejo C)  
{  
    Real += C.Real;  
    Imag += C.Imag;  
  
    return *this;  
}
```

Ambas funciones trabajan correctamente, y permiten efectuar operaciones tales como:

```
TComplejo C1, C2, C3;  
  
C3 = C1 += C2;
```

Lo primero que se nota es la presencia de un puntero, **this**. (en castellano: **este**) ¿Qué es este puntero? ¿Dónde se lo definió? Este puntero se define automáticamente en todas las clases de C++, y apunta a los datos de la clase. Este puntero se pasa automáticamente a toda función de la clase, para que esta pueda acceder a los datos de la misma. Así, toda clase tiene un puntero con el nombre **this**, que apunta a la estructura, es decir que, en nuestra función **operator+=** podríamos haber escrito:


```
Real += C.Real;
Imag += C.Imag;
```

ó

```
this->Real += C.Real;
this->Imag += C.Imag;
```

ó lo que es lo mismo:

```
(*this).Real += C.Real;
(*this).Imag += C.Imag;
```

Ahora bien, vayamos al return. En ambos casos se devuelve:

```
return *this;
```

Pero en el primer caso, la función está definida como:

```
TComplejo TComplejo::operator+= (TComplejo C)
```

es decir que se devolverá un dato de tipo TComplejo, que será copiado en la pila del sistema, cuando la función retorne, en tanto que en el segundo caso, la función está definida como:

```
TComplejo& TComplejo::operator+= (TComplejo C)
```

por lo que solo se copiará a la pila un puntero al dato TComplejo actual, y por lo tanto resultará más eficiente.

Un operador particular es el operador ++. Este (al igual que el operador --) puede ser prefijo tanto como postfijo. Pero el significado de estas dos variantes es diferente.

El operador ++ prefijo puede implementarse como sigue:

```
TComplejo& TComplejo::operator++(void)
{
    printf ("Operador ++ (prefijo)\n");
    Real++;
    return *this;
}
```

Pero el operador ++ postfijo, debe incrementar la variable, pero devolver el valor original de la variable. Por esto, no puede devolverse la variable original, sino que habrá que crear una copia de la misma. Por esta razón, el operador prefijo resulta ser más eficiente que el postfijo.

Para distinguir el operador prefijo del postfijo, este último recibe un parámetro entero, cuyo valor no debe ser tenido en cuenta.

```
TComplejo TComplejo::operator++(int)
```

```
{
    TComplejo c;
    c = *this; // Creo una copia de la variable
    Real++;    // Incremento la variable original
    return c;  // Retorno la copia
}
```

Notar que en este caso se devuelve una variable **TComplejo** por **valor y no por referencia**, como en el caso anterior. Esto se debe a que la variable **c** es local a la función **operator++(int)** razón por la cual no tiene sentido devolverla por referencia, ya que la misma se destruirá al salir de la función.

En caso de no definirse una de estas funciones, la que esté definida será utilizada tanto como prefijo y postfijo. Esto puede conducir a errores, razón por la cual se recomienda nunca escribir uno sólo de estos operadores.

Lo mismo ocurre con el operador --.

PARTE III - Programación avanzada en C++

Capítulo XXI: Entrada/Salida en C++

El lenguaje de programación C tiene una de las interfaces de entrada/salida más potentes y flexibles de todos los lenguajes de programación. Siendo así, cabe la pregunta de porqué C++ define su propio sistema de entrada/salida. En realidad, C++, más que definir un nuevo sistema de entrada/salida duplica la ya existente, adaptándola a los nuevos tipos de datos. Por ejemplo si tuviese algo como

```
struct MiTipoDeDato
{
    char Calle[100];
    int Altura;
    int CP;
    int Piso;
}Dato;
```

no hay forma en C, de adaptar printf() para este tipo de dato, es decir, no puedo hacer:

```
printf ("%MiTipoDeDato", Dato);
```

Sin embargo, sí es posible en C++ sobrecargar el operador << de tal forma que pueda hacerse:

```
cout<<Dato;
```

El sistema de entrada/salida de C++, al igual que el de C, opera con **streams**. Ver streams en C para más información. En C++ tiene cuatros streams predefinidos que se crean automáticamente al ejecutar el programa. Estos son: **cin**, **cout**, **cerr** y **clog**. **cin** está asociado a la entrada standard, **cout** a la salida standard, y **cerr** y **clog** a la salida de errores. La diferencia entre **cerr** y **clog** es que **clog** se envía a la salida cuando el buffer de salida se llena, en tanto que **cerr** los datos envía la salida inmediatamente.

Estos **streams** están definidos en el archivo **iostream.h**, donde se definen las clases **streambuf**, y en un nivel superior, heredando la clase anterior, la clase **ios** y luego las clases **istream**, **ostream** e **iostream**.

Así, para adaptar la entrada/salida C++ para un nuevo tipo de dato, habrá que sobrecargar los operadores << y >> de estos streams al nuevo tipo de dato. A esto se suele llamar insertadores y extractores.

Veamos un ejemplo: Recordemos clase **TComplejo**, y definamos la entrada/salida de C++.

Ejemplo 21.1: Objeto Complejo - entrada salida en C++

```
class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
```

```
TComplejo (int Re=0, int Im=0);
};

// Insertador para la clase TComplejo (imprime el dato en la salida standard)
ostream &operator<<(ostream &stream, TComplejo C)
{
    stream << '(' << C.Real << ' + i * ' << C.Imag << ')';
    return stream;
}

// Extractor para la clase TComplejo (carga el dato desde la entrada standard)
istream &operator>>(istream &stream, TComplejo &C)
{
    cout << "Ingrese en numero complejo Re, Im:"
    stream >>C.Real >> C.Imag;

    return stream;
}
```

Por supuesto, para poder acceder a las variables **Real** y **Imag** de la clase **TComplejo**, será necesario definir estos operadores como *amigos* (**friend**) de la clase **TComplejo**, es decir:

Ejemplo 21.2: Métodos const

```
class TComplejo
{
    private:
        int Real;
        int Imag;
    public:
        TComplejo (int Re=0, int Im=0);
        friend istream &operator>>(istream &stream, TComplejo &C)
        friend ostream &operator<<(ostream &stream, TComplejo C)
};
```

Métodos constantes

Es posible en C++ definir métodos constantes, también llamados métodos de sólo lectura. Estos métodos se caracterizan porque no pueden modificar ningún atributo de la clase, y se los declara utilizando la palabra reservada **const**, al final de la declaración del método. Por ejemplo:

```
class TComplejo
{
    private:
        double Real;
        double Imag;
    public:
        TComplejo (double Re=0, double Im=0);
        double Abs (void) const
        {
            /* Este método puede leer los atributos del
               objeto pero no modificarlos */
            double r;
```

```
    r = Real*Real + Imag*Imag;  
    return sqrt (r);  
}  
};
```

Capítulo XXII - Templates

Volvamos ahora a nuestro ejemplo de la pila, que nos permite almacenar y luego recuperar enteros. ¿Qué pasaría si necesitásemos pilas para varios tipos de datos? Si estuviésemos trabajando en C, no habría otra posibilidad que crear una estructura pila para cada tipo de dato. Y lo que es peor, !!!modificar cada una de ellas, cada vez que se quisiese hacer un cambio en la forma de trabajar de la pila!!!. Pero afortunadamente, C++ define algo llamado **template**. Esta es una palabra reservada de C++.

Veamos un ejemplo.

Ejemplo 22.1: Clase Pila - templates

```
#include <stdio.h>
#include <malloc.h>
#include <iostream.h>
#define PILA_MAX 100

template <class Tipo> class TPila {
private:
    Tipo * Pila;
    unsigned Total;
    unsigned Max;
public:
    TPila (unsigned MaxPila);
    ~TPila ();
    void Encolar (Tipo Dato);
    Tipo Desencolar (void);
};

template <class Tipo> TPila<Tipo>::TPila(unsigned MaxPila)
{
    Total = 0;
    Max = MaxPila;
    Pila = new Tipo[Max];
    if (!Pila)
        printf ("Memoria insuficiente.\n");
    else
        printf ("Pila incializada.\n");
}

template <class Tipo> TPila<Tipo>::~~TPila ()
{
    if (Pila)
        delete []Pila;
    printf ("La pila ha sido destruida.\n");
}

template <class Tipo> void TPila<Tipo>::Encolar (Tipo Dato)
{
    if (!Pila)
    {
        printf ("No se pudo inicializar la pila.\n");
    }
}
```

```
        return;
    }

    if (Total<PILA_MAX)
    {
        cout <<"Encolando:"<<Dato<<'\n';
        Pila[Total++]=Dato;
    }
    else
        printf("Pila esta llena.\n");
}

template <class Tipo> Tipo TPila<Tipo>::Desencolar (void)
{
    if (!Pila)
    {
        printf("No se pudo inicializar la pila.\n");
        return 0;
    }

    if (Total>0)
    {
        cout <<"Encolando:"<<Pila[--Total]<<'\n';
        return Pila[Total];
    }
    else
    {
        printf("Pila esta llena.\n");
        return 0;
    }
}

void main (void)
{
    TPila<char> Pila(10);

    Pila.Encolar ('a');
    Pila.Encolar ('b');

    Pila.Desencolar ();
    Pila.Desencolar ();
}
```

De esta forma estamos creando una clase *TPila*, que puede ser utilizada para encolar/desencolar cualquier tipo de dato, incluso estructuras y clases.

El tema `template` es más extenso, y no es la idea de este curso profundizar más en este tema. Para mayor información consultar bibliografía especializada.

Veamos otro ejemplo. Vamos a crear el tipo de dato *TVector*. Esto es algo muy útil en programación en C/C++, ya que es un error muy común y difícil de encontrar el excederse en el tamaño de un array. Me refiero por ejemplo a cosas del tipo:

```
char Str[100];

Str[100] = 0 ;
```

En este caso, el compilador no reportará ningún error, ni probablemente tampoco lo haga el programa al ejecutarse. Pero esto está sobrescribiendo datos en la memoria, y sin lugar a duda producirá errores que se verán en otras partes del programa. Sería muy útil poder crear un array que verificase el rango del índice.

Veamos un ejemplo.

Ejemplo 22.2: Objeto Vector - templates

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <malloc.h>
#include <string.h>
#include <conio.h>

template <class Tipo, int Size> class TVector
{
    private:
        Tipo Datos[Size];
        Tipo Aux;
    public:
        inline Tipo& operator[] (unsigned n);
};

template <class Tipo, int Size> inline Tipo& TVector<Tipo,Size>::operator[]
(unsigned n)
{
    unsigned Max;
    Max = sizeof (Datos) / sizeof (Tipo);

    if (n >= Max)
    {
        /* Insertar un BreakPoint en esta posicion, para poder analizar
           porque se produjo el error */
        cout << "Error: Se excede la capacidad del buffer.\n";

        return Aux;
    }
    return Datos[n];
}

void main(void)
{
    clrscr();

    TVector <int,4>Vect;

    for (int i=0; i<5;++i)
        Vect[i] = 100+i;

    for (i=0; i<5;++i)
        cout << Vect[i]<<'\n';
}
```

Este programa produce la siguiente salida:

```
Error: Se excede la capacidad del buffer.
100
101
102
103
Error: Se excede la capacidad del buffer.
104
```

Veamos el código fuente en detalle:

```
template <class Tipo, int Size> class TVector
```

El programa comienza definiendo una clase llamada TVector, que será un template, es decir, que recibirá parámetros en el momento de su creación. En este caso se reciben dos datos, el primero, el tipo de dato específico de la clase, el tipo de dato de los elementos del vector. El segundo, la cantidad de elementos máximos del vector.

Con esto, ya puedo crear vectores de cualquier tipo de dato. Por ejemplo, podría debería escribir:

```
char str[100];   como TVector <char,100> str;
int vect[200];  como TVector <int,200> vect;
```

¿Cómo hago ahora para acceder a los datos del vector? Para empezar, podría definir un par de funciones:

```
template <class Tipo, int Size> void TVector<Tipo,Size>::CargarElemento
(unsigned Posicion, Tdato dato);
template <class Tipo, int Size> Tipo TVector<Tipo,Size>::LeerElemento
(unsigned Posicion);
```

Sin embargo, ¿porqué no utilizar operadores para hacer esto? C++ permite sobrecargar el operador [] (corchete), y podríamos hacerlo de las siguientes formas:

```
Tipo TVector::operator[] (unsigned n)
{
    return Datos[n];
}
```

ó

```
Tipo& TVector::operator[] (unsigned n)
{
    return Datos[n];
}
```

Nuevamente, la primer versión devolverá el dato en cuestión, es decir que si definimos:

```
TVector <int, 10>V;
int dato;
```

podremos hacer cosas como:

```
dato = V[1];
```

pero NO:

```
V[1] = dato;
```

Esto se soluciona con la segunda versión de **`operator[]`**, que devuelve un puntero al dato.

Ahora bien, ya que la clase fue creada como **`template<class Tipo, int Size>`**, será necesario agregar esto a la definición de la función, con lo cual:

```
Tipo& TVector<Tipo,Size>::operator[] (unsigned n)
```

pasará a ser ahora:

```
template <class Tipo, int Size> Tipo& TVector<Tipo,Size>::operator[] (unsigned n)
```

Una última corrección que puede hacerse a la función es agregar la palabra reservada **`inline`**. Esta hace que el compilador expanda en línea el código, en vez de llamar a la función, con lo cual el programa se vuelve más rápido (pero también más grande). Esto conviene hacerlo, ya que en caso contrario, cada acceso al vector requerirá una llamada a una función.

```
template <class Tipo, int Size> inline Tipo& TVector<Tipo,Size>::operator[]  
(unsigned n)
```

Finalmente, y el objetivo de nuestro código es verificar que el índice que se pasa al operador esté dentro del rango permitido. Esto lo podemos hacer de la siguiente forma:

```
template <class Tipo, int Size> inline Tipo& TVector<Tipo,Size>::operator[]  
(unsigned n)  
{  
    unsigned Max;  
    Max = sizeof (Datos) / sizeof (class Tipo);  
  
    if (n >= Max)  
    {  
        /* Insertar un BreakPoint en esta posicion, para poder analizar  
           porque se produjo el error */  
        cout << "Error: Se excede la capacidad del buffer.\n";  
  
        return Aux;  
    }  
    return Datos[n];  
}
```

¿Qué es todo esto? Vamos por partes. La línea:

```
Max = sizeof (Datos) / sizeof (class Tipo);
```

calcula la cantidad de elementos que pueden alojarse en el vector, dividiendo el tamaño total del vector de datos sobre el tamaño individual de cada dato. En el paso siguiente, verificamos que el índice, **`n`**, esté dentro del rango permitido. En caso contrario, indicamos el error. Notar la utilidad de poner un breakpoint (punto de ruptura) en la línea que produce el error. Esto hará que al verificar el programa, la ejecución se interrumpa en dicha línea, y se pueda verificar qué ocasionó que se llegue a esta situación.

Finalmente queda una pregunta. ¿Porqué el **`return Aux`**? Lo que ocurre aquí es que no puedo efectuar la operación: **`return Datos[n]`**, ya que **`n`** está fuera de rango. Podría interrumpir la ejecución del programa, **`exit(1)`**; pero en un programa importante, debe reportarse el error de una forma adecuada. Incluso puede ser que el programa tenga otras formas de solucionar el inconveniente, por ejemplo, si se tratase de un sistema de control, puede ser grave que súbitamente el programa se interrumpa. Pero necesariamente tengo que

devolver un dato, ya que la función así lo requiere. Podría devolver una posición cualquiera del array, por ejemplo la cero, pero podrían ocurrir cosas como:

```
TVector <int, 4>Vect;  
Vect[5] = 10;
```

En este caso la posición cero del array sería sobrescrita, lo cual no es correcto. Por ello, la solución más adecuada es, o bien reservar un lugar adicional en el array, o por claridad, definir un dato auxiliar.

Clases template dentro de clases template

¿Pueden usarse las clases template como base para construir nuevas clases, o usar clases template como argumento de clases template? Desde ya que sí, son clases como cualquier otra. Como ejemplo, a continuación se usará la clase *TVector* como base para crear el tipo de dato *TString*. Luego, a continuación se usarán estas dos clases para crear una lista de strings. Desde ya que esta estructura de clases no resultará óptima, pero eso no es importante en este momento.

Ejemplo 22.3: Objeto Vector - templates anidados

```
#include <stdio.h>  
#include <stdlib.h>  
#include <iostream.h>  
#include <malloc.h>  
#include <string.h>  
#include <conio.h>  
  
/*****  
/***          C L A S E   V E C T O R   E S T A T I C O          ***/  
/*****/  
template <class Tipo, int Size> class TVector  
{  
    private:  
        Tipo Datos[Size];  
        Tipo Aux;  
    public:  
        inline Tipo& operator[] (unsigned n);  
};  
  
template <class Tipo, int Size> inline Tipo& TVector<Tipo,Size>::operator[]  
(unsigned n)  
{  
    unsigned Max;  
    Max = sizeof (Datos) / sizeof (Tipo);  
  
    if (n >= Max)  
    {  
        /* Insertar un BreakPoint en esta posicion, para poder analizar  
        porque se produjo el error */  
        cout << "Error: Se excede la capacidad del buffer.\n";  
  
        return Aux;  
    }  
}
```

```
        return Datos[n];
    }

/*****
/****          CLASE    TSTRING, DERIVADA DE TVECTOR          ****/
/****          *****/
class TString: public TVector <char, 100>
{
    private:
    public:
        TString& operator=(char *s)
        {
            for (unsigned i=0;i<strlen(s)+1;++i)
                (*this)[i] = s[i];
            return *this;
        }
        friend ostream& operator<< (ostream &stream, TString &s);
};

ostream& operator<< (ostream &stream, TString &s)
{
    stream << &(s[0]);
    return stream;
}

/*****
/****          F U N C I O N    M A I N          ****/
/****          *****/
void main(void)
{
    clrscr();
    TString s;
    TVector <TString, 10> StrList;

    s = "hola";
    StrList[1] = "HOLA ";
    StrList[2] = "MUNDO";


    cout <<s <<"\n";
    cout <<StrList[1]<<"\n";
    cout <<StrList[2]<<"\n";
}
```

La salida de este programa es la siguiente:

```
hola
HOLA
MUNDO
```

EL PREPROCESADOR

Finalmente un último comentario con respecto al programa anterior. Efectivamente, cada vez que se quiera acceder a una posición del array, se ejecutarán varias instrucciones, lo cual volverá al programa bastante ineficiente. Para evitar esto hay una solución, que es la utilización de definiciones. Por ejemplo podría escribirse:

 **Ejemplo 22.4:** El preprocesador

```
template <class Tipo, int Size> inline Tipo& TVector<Tipo,Size>::operator[]
(unsigned n)
{
    #if defined (DEBUG_MODE)
        unsigned Max;
        Max = sizeof (Datos) / sizeof (class Tipo);

        if (n >= Max)
        {
            /* Insertar un BreakPoint en esta posicion, para poder analizar
               porque se produjo el error */
            cout << "Error: Se excede la capacidad del buffer.\n";

            return Aux;
        }
    #endif
    return Datos[n];
}
```

Si al comienzo del programa se define `DEBUG_MODE` (`#define DEBUG_MODE`), se efectuaran todas las verificaciones, lo cual se dejará de hacer con sólo eliminar esta definición.

Otra macro útil al trabajar con C++ es `__cplusplus`. Esta macro estará definida si se compila el programa utilizando un compilador C++ y es fundamental para compilar código C junto con C++ (Ver apéndice V). Ejemplo:

```
void main (void)
{
    FILE *fp;

    #ifndef __cplusplus
        /* Mostrar el tipo de compilador utilizado */
        printf ("El programa se compiló en C\n");
    #else
        cout <<"El programa fue compilado en C++ en "<< __DATE__ <<"\n";
    #endif
}
```

Capítulo XXIII - Un poco más sobre herencia - Herencia múltiple

En las secciones anteriores vimos cómo podía definirse una clase como derivada de otra, y así sucesivamente. Esto permite crear estructura muy complejas de clases, que se heredan unas a otras. Incluso es posible hacer que una misma clase se construya heredando más de una clase. Por ejemplo:

Ejemplo 23.1: Herencia múltiple

```
class TA
{ ... };

class TB
{ ... };

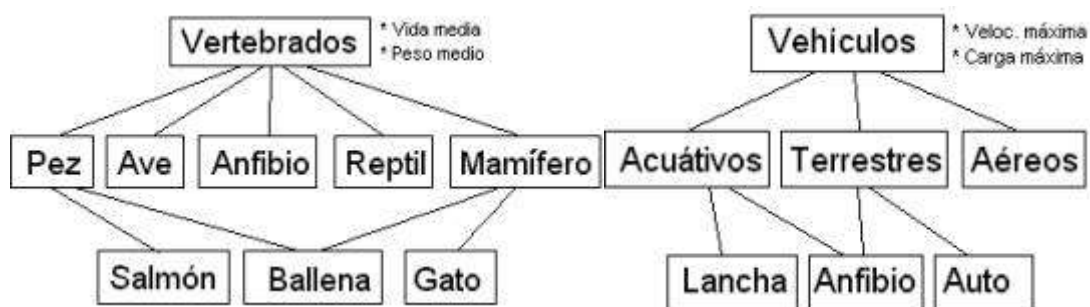
class TC : public TA, private TB
{ ... };
```

Pueden surgir situaciones complejas cuando una misma clase se hereda más de una vez. Por ejemplo, imaginemos ahora dos situaciones como las siguientes:

Situación 1: Quiero crear clases para describir los seres vivos, clasificándolos en aves, peces, anfibios, reptiles, mamíferos. Para describir la Ballena, me puede interesar caracterizarla como un mamífero pero con características de pez.

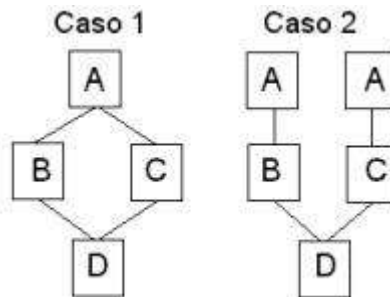
Situación 2: Quiero crear clases para describir características de algunos vehículos. Pero surge un vehículo anfibio capaz de desplazarse por agua y por tierra.

Estos casos se grafican esquemáticamente a continuación:



Los textos marcados con asterisco (*) son los atributos de la clase. Veamos estos dos casos con más detalle. En el primer ejemplo, al definir la clase **Ballena**, la clase **Vertebrados** será incluida dos veces, una por **Pez** y otra por **Mamífero**. Lo mismo sucederá en el segundo ejemplo con la clase **Anfibio**. Pero en el primer caso no quiero que se dupliquen los atributos **VidaMedia** y **PesoMedio** de la clase predecesora, en tanto que en el segundo caso sí quiero que se dupliquen los atributos **VelocMaxima** y **CargaMaxima**, ya los mismos serán diferentes cuando el vehículo se esté comportando como un vehículo acuático o como terrestre.

Es decir, se tienen las siguientes situaciones:



¿Cómo se resuelve esto? En caso de plantearse esta situación C++ se asume por defecto el segundo caso. Si lo que se quiere es el primero, debe utilizarse **herencia virtual**.

Veamos como implementar estos dos casos:

Empecemos por el segundo, que es el más simple:

Ejemplo 23.2: Herencia múltiple - Múltiple heredad de una clase

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

class TA
{
public:
    int a;
};

class TB:public TA
{
};

class TC:public TA
{
};

class TD:public TB,public TC
{
};

void main (void)
{
    TD d1;

    /* d1.a = 1;
    Sería es ambiguo, ya que no habría forma de saber a cual
```



```
    se los dos atributos 'a' se quiere hacer referencia.
*/
dl.TB::a = 1;
dl.TC::a = 2;

printf ("dl.TB::a = %d    dl.TC::a = %d\n", dl.TB::a, dl.TC::a);
}
```

Si quiere que se entienda por defecto una de estas dos opciones, puede usarse la palabra reservada using:

Ejemplo 23.3: Herencia múltiple - operador using

```
void main (void)
{
    TD dl;

    using TC::a;
    /* dl.a = 1;
    Sería es ambiguo, ya que no habría forma de saber a cual
    se los dos atributos 'a' se quiere hacer referencia.
    */
    dl.TB::a = 1;
    dl.a = 2; /* Por defecto se entiende dl.TC::a*/

    printf ("dl.TB::a = %d    dl.TC::a = %d\n", dl.TB::a, dl.TC::a);
}
```

Este programa produce la salida

```
dl.TB::a = 1    dl.TC::a = 2
```

Veamos ahora como implementar el primer caso:

Ejemplo 23.4: Herencia múltiple - clases virtuales

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

class TA
{
    public:
        int a;
};

class TB:public virtual TA
{
};

class TC:public virtual TA
{
};

class TD:public TB,public TC
{
}
```

```
};

void main(void)
{
    TD d1;

    d1.TB::a = 1;
    d1.TC::a = 2;
    d1.a = 3; /* Ahora esta línea no es ambigua */

    printf ("d1.TB::a = %d    d1.TC::a = %d\n", d1.TB::a, d1.TC::a);
}
```

Este programa produce la salida

```
d1.TB::a = 3    d1.TC::a = 3
```

Notar la palabra reservada ***virtual*** en la definición de las clases ***TB*** y ***TC***, que indica que la clase ***TA*** debe ser heredada una única vez. Notar también que no es posible duplicar tan solo parte de los atributos, es decir, no hay forma de declarar un atributo ***virtual***. Sí se puede declarar métodos virtuales pero el significado de los mismos es diferente, ya que no tiene sentido hablar de duplicar un mismo código.

Los métodos virtuales se verán en la siguiente sección.

Capítulo XXIV - Métodos virtuales y funciones puras y abstractas

Supongamos una clase para almacenamiento de datos. Esta clase tendría los métodos tales como insertar, borrar, editar, grabar a disco y cargar de disco. Un programador podría utilizar esta *clase base*, para construir bases de datos más sofisticadas, y podría querer reemplazar funciones de la clase base por otras mejoradas, o al menos agregarle otras funcionalidades. Por ejemplo, supongamos que quisiese reemplazar el método de escritura a disco de la clase base. Desde el punto de vista del programa o de nuevas clases herederas, basta con definir una nueva función con el mismo nombre, por ejemplo:

Ejemplo 24.1: Clase Base

```
class TBase {
    ...
public:
    Insertar (...);
    Editar (...);
    Borrar (...);
    GrabarADisco(...);
    CargarDeDisco(...);
};

class TBaseMejorada:private TBase {
public:
    GrabarADisco(...);
};
```

En este caso, si se llamase al método **GrabarADisco()** desde el programa principal o desde una clase derivada de **TBaseMejorada**, se llamaría a la función definida en esta última. Sin embargo esta nueva función no será llamada desde la clase TBase, la cual seguirá llamando a su propio método. Por ejemplo, si la clase TBase tuviese un método AutoSave, que automáticamente llamase a GrabarADisco(), se llamaría al método original.

¿Cómo puede hacerse para que siempre se llame al método de la clase derivada, aún desde la clase base?
Rta: Definiendo el método como virtual.

Las llamadas a un método definido como virtual serán completamente reemplazadas por llamadas a los métodos similares de clases derivadas, de existir estos. Todavía puede llamarse a las funciones virtuales originales, si se especifica previamente el nombre de la clase seguida de :: (doble dos puntos). Veamos ahora el ejemplo completo.

Ejemplo 24.2: Clase Base - métodos virtuales

```
class TBase {
    ...
public:
    void AutoSave ()
    {
        GrabarADisco();
    }
    void Insertar ();
```

```
void Editar ();
void Borrar ();
virtual void GrabarADisco()
{
    cout <<"Base grabada\n";
}
void CargarDeDisco();
};

class TBaseMejorada:public TBase {
public:
    void GrabarADisco(...)
    {
        cout << "Realmente desea grabar?\n";
        if (;SolicitarConfirmación ())
            return;
        TBase::GrabarADisco();
    };
};

void main (void)
{
    TBaseMejorada Base;

    Base.AutoSave();
}
```

Este programa produciría la siguiente salida:

```
Realmente desea grabar?
Base grabada
```

Ya que un método virtual será posiblemente redefinido en una clase derivada, tiene sentido pensar en que el mismo pueda ser declarado, pero no definido, es decir, que en la clase base no se especifique ningún código para dicho método, tan solo se declare su existencia para que pueda ser llamado por métodos de la clase, dejando su declaración a clases derivadas. Métodos de este tipo se conocen como **métodos o funciones virtuales puras**. Las clases que contengan una o más funciones puras se conocen con el nombre de **clases abstractas**, y no podrán usarse para crear variables de este tipo, sino tan solo clases derivadas en donde se declaren estas funciones virtuales.

Los métodos virtuales puros se declaran en forma idéntica a cualquier método virtual, pero igualados a cero. Por ejemplo:



Ejemplo 24.3: Clase Base: Métodos puros y abstractos

```
class TBase {
...
public:
    void AutoSave ();
    void Insertar ();
    void Editar ();
    void Borrar ();
    virtual void GrabarADisco() = 0;
```

```
};
```

LA HERENCIA VIRTUAL Y EL COMPILADOR

¿Cómo se resuelve el problema de la herencia virtual desde el punto de vista del compilador? Para ello volvamos a analizar el ejemplo del capítulo anterior.

Ejemplo 24.4: Herencia virtual

```
class TA
{ int a; };
class TB: virtual public TA{
{ int b; };
class TC: virtual public TA{
{ int c; };
class TD: public TB, public TC
{ int d; };
```

Ejemplo 24.5: Herencia virtual

```
class TA
{ int a; };
class TB: public TA{
{ int b; };
class TC: public TA{
{ int c; };
class TD: public TB, public TC
{ int d; };
```



En el segundo caso estará claro que una clase tipo TD se compondrá de 5 variables de tipo entero, y por lo tanto será de esperar que el tamaño de la misma sea de $5 \times \text{sizeof}(\text{int})$ bytes.

Si analizásemos el contenido la estructura TD en la memoria de la PC podríamos encontrar algo como lo siguiente:

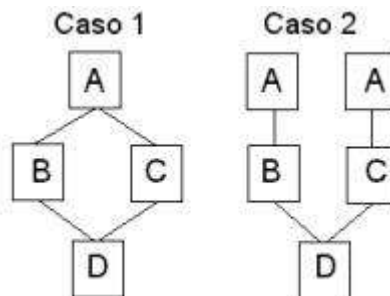


Qué ocurre con el primer caso. Ya que la clase TA se hereda en forma virtual, tanto en TB como en TC, una clase de tipo TD contendrá 4 variables: a, b, c y d. Sería de esperar que el tamaño de la clase sea de $4 * \text{sizeof}(\text{int})$ bytes, pero no es así. Dependiendo del compilador, pueden obtenerse valores tales como $6 * \text{sizeof}(\text{int})$. ¿Qué ha ocurrido? Si el compilador simplemente armase la nueva clase TD con 4 variables sería imposible efectuar casting entre objetos (ver el capítulo casting de objetos). Concretamente, no podrían efectuarse operaciones tales como:

```

TA *A;
TB *B;
TC *C;
TD *D = new TD;
A = D;      // Obtengo la posición de la clase tipo TA dentro de la tipo TD
B = D;      // Obtengo la posición de la clase tipo TB dentro de la tipo TD
C = D;      // Obtengo la posición de la clase tipo TC dentro de la tipo TD
    
```

Una alternativa a esto consiste en duplicar efectivamente las variables de la clase TA, como en el ejemplo anterior, y garantizar que estas dos variables tengan siempre el mismo valor, pero esto es imposible, ya que una de ellas podría ser modificada de una forma no prevista. La alternativa implementada por C++ es utilizar punteros. En todo caso en que se herede una clase en forma virtual, la misma será accedida internamente mediante un puntero. Por ejemplo, podríamos pensar en una situación como la siguiente:



Capítulo XXV - Utilización de memoria dinámica en clases - Constructor de copia

La memoria dinámica y los objetos

Veamos ahora otro ejemplo. Ya que estamos definiendo vectores, porqué no definir un vector que utilice memoria dinámica, es decir que reserve memoria al ser creado y la libere cuando ya no se lo requiera. C++ parece ideal para esto. Permite implementar un constructor, encargado de reservar memoria para él, y un destructor que automáticamente lo destruya cuando se lo deja de utilizar. Pues bien, hagámoslo:

Ejemplo 25.1: Objeto String

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <iostream.h>

class TString
{
    private:
        char *Str;
        unsigned Size;
    public:
        TString (unsigned S);
        void operator=(char * Txt);
        ~TString ();
};

TString::TString (unsigned S)
{
    Size = S;

    Str = new char [Size];

    if (!Str)
    {
        cout << "Memoria insuficiente. No se puede inicializar string.\n";
        /* En un programa importante, no debera llamarse a exit(), sino
           retomar y reportar el error adecuadamente */
        exit (1);
    }

    cout << "String inicializado.\n";
}

TString::~TString ()
{
    if (!Str)
    {
        delete []Str;
        Str = NULL;
    }
    cout << "String destruido.\n";
}
```

```
void TString::operator=(char * Txt)
{
    if (!Str)
    {
        cout << "Se quiere asignar datos a un string sin inicializar.\n";
        return;
    }
    if (strlen (Txt) >= Size)
    {
        cout << "Cadena demasiado larga.\n";
        return;
    }
    strcpy (Str,Txt);
}

void main(void)
{
    clrscr();
    TString s1(100);

    s1="Hola mundo";
}
```

El programa produce la siguiente salida al ser ejecutado:

```
String inicializado.
String destruido.
```

Lo cual es correcto. Agreguemos ahora una función que reciba un TString, es decir, que haga algo como lo siguiente:

```
void MiFuncion (TString S)
{
    ...
}

void main(void)
{
    clrscr();
    TString s1 (100);

    s1 = "Hola mundo";

    MiFuncion (s1);
}
```

Aparentemente no debería haber ningún problema, no ha cambiado mucho. Pero al ejecutar el programa se obtiene la siguiente salida:

```
String inicializado.
String destruido.
String destruido.
```


!!!Se han destruido dos string, pero se ha creado uno sólo!!! Esto ya indica la presencia de un error. ¿Pero qué es lo que ha ocurrido? Para responder esta pregunta es necesario prestar atención al código del programa, y a la forma en que los constructores y destructores funcionan.

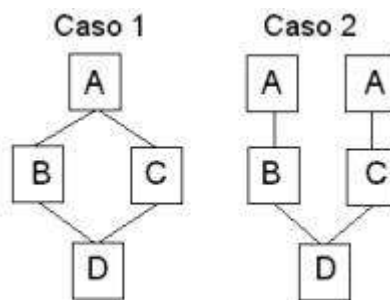
En la segunda línea de la función main() se crea el dato **s1**. Al crear este dato se llama al constructor del mismo, que reserva memoria para el dato.



Un par de líneas más abajo, se llama a la función **MiFuncion()**, a la cual se pasa una copia del dato **s1**. Este dato tendrá el nombre **S**. En este punto se tiene la siguiente situación:



Pero la variable **S** es efectivamente una variable de tipo **TString**, que pertenece a la función **MiFuncion()**. Y por lo tanto, al salir de la misma, se llamará al destructor de la variable. Este destructor destruirá el dato ***Str**, y por lo tanto se tendrá la siguiente situación:



Finalmente, cuando se salga de la función main, se llamará al destructor de la variable **s1**, la cual a su vez **LIBERARÁ NUEVAMENTE LA MEMORIA YA LIBERADA**, lo cual puede provocar entre otras cosas que el sistema se "cuelgue" o se reporte un mensaje de error.

En el ejemplo que acabamos de ver, puede solucionarse este inconveniente haciendo algo como lo siguiente:

```
void MiFuncion (TString &S)
{ ... }
```

o como lo siguiente:

```
void MiFuncion (TString *S)
{ ... }
```

Esto funcionará correctamente. En el primer caso S no es una nueva variable sino la misma, pasada por referencia, y en el segundo S no es una variable de tipo TString sino un puntero. Sin embargo si modificamos la función para que devuelva una copia del dato como sigue:

```
TString MiFuncion (TString &S)
{
    return S;
}
```

Volveríamos a tener el mismo problema.

Constructor de copia

Retomemos el problema planteado en la sección anterior. Estos problemas surgen porque estamos creando copias del objeto original, pero no duplicamos los datos dinámicos, los cuales siguen apuntando a la misma posición. Para solucionar todos estos inconvenientes simplemente se debe crear un **constructor de copia** y un **operador =**.

Un constructor de copia es un constructor de la clase como cualquier otro, con la salvedad de que recibe como único parámetro un objeto del mismo tipo, por referencia.

Por ejemplo, en este caso se tendría:

```
TString::TString (TString &S2)
{ ... }
```

Este constructor será llamado automáticamente cada vez que automáticamente se duplique un objeto de clase.

Está prohibido definir un constructor como sigue:

```
TString::TString (TString S2)
```

ya que al llamarlo, habría que crear una copia del objeto, y para crear esta copia habría que llamar a este constructor, y así indefinidamente.

Pero todavía falta un detalle, el constructor de copia no es suficiente, ya que este será llamado cada vez que el compilador **automáticamente** duplique la variable. Todavía falta solucionar el problema cuando el

programador explícitamente duplique la variable, es decir cuando se escriba `Var1 = Var2`. Si no se lo hiciese, se usaría la asignación por defecto, que es una copia literal del contenido.

Es bueno que todo programador principiante agregue a los constructores, destructores y operadores, código suficiente para poder verificar el correcto funcionamiento de los mismos.

Veamos ahora como quedaría este ejemplo:

Por simplicidad dejaremos el operador `=` para después.

Ejemplo 25.2: Objeto String - Constructor de copia

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <iostream.h>
5  #include <string.h>
6  #include <conio.h>
7
8  class TString{
9      private:
10         char * Str;
11     public:
12         TString ();
13         TString (char *s);
14         ~TString ();
15         TString (TString &S2);
16 };
17
18 TString::TString ()
19 {
20     cout<<"Construyendo un string vacio\n";
21     Str = NULL;
22 }
23
24 TString::TString (char *s)
25 {
26     cout<<"Construyendo un string con el texto: "<<s<<"\n";
27     Str = new char[strlen(s)+1];
28     strcpy (Str,s);
29 }
30
31 TString::~~TString ()
32 {
33     if (Str)
34         cout<<"Destruyendo el string con el texto: "<<Str<<"\n";
35     else
36         cout<<"Destruyendo un string vacio\n";
37     if (Str)
38         delete []Str;
39     Str = NULL;
40 }
41
42 TString::TString (TString &S2)
```

```
43  {
44      if (S2.Str)
45          cout<<"Duplicando el string con el texto: "<<S2.Str<<"\n";
46      else
47          cout<<"Duplicando un string vacio\n";
48
49      if (S2.Str == NULL)
50          Str = NULL;
51      else
52      {
53          Str = new char[strlen(S2.Str)+1];
54          strcpy (Str, S2.Str);
55      }
56  }
57
58  TString MiFunc (TString S2)
59  {
60      return S2;
61  }
62
63  void main(void)
64  {
65      clrscr();
66      TString S("Hola");
67
68      MiFunc (S);
69  }
```

Este programa produce la siguiente salida:

```
Construyendo un string con el texto: Hola
Duplicando el string con el texto: Hola
Duplicando el string con el texto: Hola
Destruyendo el string con el texto: Hola
Destruyendo el string con el texto: Hola
Destruyendo el string con el texto: Hola
```

Veamos qué ocurre en detalle. En la línea 66 se construye el primer string con el texto Hola. Luego en la línea 68 se lo duplica, para llamar a la función *MiFunc()*. Esta variable será la variable **S2**, que se define en la línea 58. Un par de líneas más abajo, en la línea 60 se duplica nuevamente este string, el cual se almacena en una variable temporal, que es devuelta por la función. En la línea 61, al salir de la función, se destruye la variable **S2**, que deja de tener significado, y se retorna de la función. La variable temporal devuelta se pierde por no ser copiada a ningún lugar, y en la línea 66, inmediatamente después del retorno de la función, se destruye. Finalmente, al salir de la función *main()* se destruye la última variable de tipo **TString (S)** que quedaba.

Otro ejemplo

Alentados por estos buenos resultados, agreguemos ahora los operadores += y +, para concatenar dos strings en el primero y en uno nuevo respectivamente.

Antes de hacerlo, agreguemos un par de funciones simples, una para vaciar el string: void **TString::Clear()** y otra para mostrar su contenido, sobrecargando el operador << como ya se ha visto. No nos olvidemos del operador = que había quedado pendiente.

Estas funciones serán como sigue:

Ejemplo 25.3: Objeto String - operadores varios

```
void TString::Clear ()
{
    if (Str)
        delete []Str;
    Str = NULL;
}

ostream& operator<<(ostream &stream, TString &S)
{
    stream<<S.Str;
    return stream;
}

/* Código incorrecto */
TString& TString::operator=(TString &S2)
{
    if (S2.Str)
        cout<<"Duplicando (=) el string con el texto: "<<S2.Str<<"\n";
    else
        cout<<"Duplicando (=) un string vacio\n";

    if (S2.Str == NULL)
        Str = NULL;
    else
    {
        Clear();
        Str = new char[strlen(S2.Str)+1];
        strcpy (Str, S2.Str);
    }
    return *this;
}
```

El operador = parecería ser correcto. Incluso con un análisis exhaustivo se verá que lo es, salvo para el caso en que se hiciese algo como: **Var1 = Var1**; ya que **strcpy()** no puede copiar un string sobre si mismo. La asignación **a = a** parecería no tener sentido. Sin embargo es válida y es muy común que se produzca cuando se utilizan operadores en cascada. Se verá esto más adelante.

El código correcto es el siguiente:

```
TString& TString::operator= (TString &S2)
{
    if (this == &S2)
        return *this;

    if (S2.Str)
        cout<<"Duplicando (=) el string con el texto: "<<S2.Str<<"\n";
    else
        cout<<"Duplicando (=) un string vacio\n";

    if (S2.Str == NULL)
        Str = NULL;
    else
```

```
{
    Clear();
    Str = new char[strlen(S2.Str)+1];
    strcpy (Str, S2.Str);
}
return *this;
}
```

Muy bien, ahora creemos el operador **+=**. Este operador deberá concatenar el variables de tipo **TString**. Es decir, deberá ser similar a **strcpy()**. Este operador permitirá efectuar operaciones como:

```
S1 += S2;
```

o como

```
S1 += TString ("hola");
```

Podría crearse otro operador **+=** que permita concatenar cadenas de caracteres (pero esto lo dejaremos de lado ya es trivial), tal como:

```
S1 += "hola";
```

```
/* Codigo incorrecto */
TString& TString::operator+=(TString &S2)
{
    /* Si el string a concatenar no esta vacio */
    if (S2.Str != NULL)
    {
        if (Str == NULL)
            Str = strdup (S2.Str);
        else
        {
            Str = (char *)realloc (Str, strlen (Str) + strlen (S2.Str) + 1);
            strcat (Str, S2.Str);
        }
    }

    return *this;
}
```

Notar el símbolo **&** en la indicación del tipo a devolver (**TString&**) en la definición de los operadores **=** y **+=**, que indica que debe devolverse la variable por referencia y no una copia de la misma. Si no lo hubiesemos puesto, el return hubiese creado una copia temporal de la variable ***this**, a la cual hubiese luego destruido, lo cual hubiese significado una pérdida de tiempo.

Al igual que antes, el operador **+=** parecería ser correcto, salvo por el hecho de que no soporta operaciones tales como: **Var1 += Var1**; ya que **strcat()** no puede concatenar un string con si mismo. El código correcto es el siguiente:

```
TString& TString::operator+=(TString &S2)
{
    /* Si el string a concatenar no esta vacio */
    if (S2.Str != NULL)
    {
        if (Str == NULL)
```

```
        Str = strdup (S2.Str);
    else
    {
        Str = (char *)realloc (Str, strlen (Str) + strlen (S2.Str) + 1);
        memmove (Str+strlen(Str), S2.Str, strlen (S2.Str)+1);
    }

    return *this;
}
```

Finalmente agreguemos el operador **+**. Este operador deberá crear un nuevo string que contenga los dos anteriores concatenados, y devolverlo en una variable de tipo **TString**. Qué más simple y sencillo que escribir un operador como sigue:

```
/*Codigo incorrecto */
TString& TString::operator+(TString &S2)
{
    TString S3(*this);
    S3 += S2;

    return S3;
}
```

Sin embargo este código no compilará, ya que la variable S3 dejará de existir inmediatamente al terminar la función, y por lo tanto no puede devolversela por referencia. Una alternativa es devolver una copia de la variable:

```
TString TString::operator+(TString &S2)
```

es decir, sin el **&**. Pero esto hará que se cree una copia de la variable al salir de la función, la cual deberá ser luego destruida, y lo cual será una pérdida de tiempo innecesaria.

La solución es utilizar una variable estática, es decir, que no se destruya al salir de la función:

```
TString& TString::operator+(TString &S2)
{
    static TString S3;

    cout<<"Sumando '"<<*this<<"' con '"<<S2<<"'\n";

    S3 = *this;
    S3 += S2;

    return S3;
}
```

Supongamos que ahora se tiene luego un código como el siguiente:

```
void main (void)
{
    clrscr ();
    TString S1 ("Hola");
    TString S2 (" mundo");
```

```
TString S3;

S3 = S1 + S2 + TString (" del C++");

cout <<"El string final es: "<<S3<<"\n";
}
```

El listado completo del programa se muestra a continuación:

Ejemplo 25.4: Objeto String - Versión final

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <malloc.h>
4  #include <iostream.h>
5  #include <string.h>
6  #include <conio.h>
7
8  class TString{
9      private:
10         char * Str;
11     public:
12         void Clear ();
13         TString ();
14         TString (char *s);
15         ~TString ();
16         TString (TString &S2);
17         TString& operator=(TString &S2);
18         TString& operator+=(TString &S2);
19         TString& operator+(TString &S2);
20         friend ostream& operator<<(ostream &stream, TString &S);
21     };
22
23     ostream& operator<<(ostream &stream, TString &S)
24     {
25         stream<<S.Str;
26         return stream;
27     }
28
29     void TString::Clear ()
30     {
31         if (Str)
32             delete []Str;
33         Str = NULL;
34     }
35
36
37     TString& TString::operator+=(TString &S2)
38     {
39         /* Si el string a concatenar no esta vacio */
40         if (S2.Str != NULL)
41         {
42             if (Str == NULL)
43                 Str = strdup (S2.Str);
44             else
45                 {
```



```
46         Str = (char *)realloc (Str, strlen (Str) + strlen (S2.Str) + 1);
47         memmove (Str+strlen (Str), S2.Str, strlen (S2.Str)+1);
48     }
49 }
50
51     return *this;
52 }
53
54 TString::TString ()
55 {
56     cout<<"Construyendo un string vacio\n";
57     Str = NULL;
58 }
59
60 TString::TString (char *s)
61 {
62     cout<<"Construyendo un string con el texto: "<<s<<"\n";
63     Str = new char[strlen(s)+1];
64     strcpy (Str,s);
65 }
66
67 TString::~~TString ()
68 {
69     if (Str)
70         cout<<"Destruyendo el string con el texto: "<<Str<<"\n";
71     else
72         cout<<"Destruyendo un string vacio\n";
73     if (Str)
74         delete []Str;
75     Str = NULL;
76 }
77
78 TString::TString (TString &S2)
79 {
80     if (S2.Str)
81         cout<<"Duplicando el string con el texto: "<<S2.Str<<"\n";
82     else
83         cout<<"Duplicando un string vacio\n";
84
85     if (S2.Str == NULL)
86         Str = NULL;
87     else
88     {
89         Str = new char[strlen(S2.Str)+1];
90         strcpy (Str, S2.Str);
91     }
92 }
93
94 TString& TString::operator=(TString &S2)
95 {
96     if (this == &S2)
97         return *this;
98
99     if (S2.Str)
100         cout<<"Duplicando (=) el string con el texto: "<<S2.Str<<"\n";
101     else
102         cout<<"Duplicando (=) un string vacio\n";
103
104     if (S2.Str == NULL)
105         Str = NULL;
106     else
```

```
107     {
108         Clear();
109         Str = new char[strlen(S2.Str)+1];
110         strcpy (Str, S2.Str);
111     }
112     return *this;
113 }
114
115
116 TString MiFunc (TString S)
117 {
118     return S;
119 }
120
121 /* Codigo incorrecto */
122 TString& TString::operator+(TString &S2)
123 {
124     static TString Stmp;
125
126     cout<<"Sumando '"<<*this<<"' con '"<<S2<<"'\n";
127
128     Stmp = (*this);
129     Stmp += S2;
130
131     return Stmp;
132 }
133
134 void main (void)
135 {
136     clrscr();
137     TString S1("Hola");
138     TString S2(" mundo");
139     TString S3;
140
141     S3 = S1 + S2 + TString (" del C++");
142
143     cout <<"El string final es: "<<S3<<"\n";
144 }
```

La salida completa de este programa es la siguiente:

```
Construyendo un string con el texto: Hola
Construyendo un string con el texto: mundo
Construyendo un string vacio
Construyendo un string con el texto: del C++
Construyendo un string vacio
Sumando 'Hola' con ' mundo'
Duplicando (=) el string con el texto: Hola
Sumando 'Hola mundo' con ' del C++'
Duplicando (=) el string con el texto: Hola mundo del C++
Destruyendo el string con el texto: del C++
El string final es: Hola mundo del C++
Destruyendo el string con el texto: Hola mundo del C++
Destruyendo el string con el texto: mundo
Destruyendo el string con el texto: Hola
Destruyendo el string con el texto: Hola mundo del C++
```

Analicemos el programa en detalle:

En las líneas 137, 138 y 139 se crean los tres string iniciales, los dos primeros con los textos "Hola", " mundo", y el tercero inicialmente vacío.

La línea 141 **se evalúa de izquierda a derecha**, pero lo primero que se hace es armarla en forma completa. Para ello, antes de comenzar a evaluarla se crea un string temporario con el texto " del C++".

Esta línea 141 debe interpretarse como sigue:

```
S3.operator= ( (S1.operator+ (S2.operator)).operator+ (TString (" del C++")) );
```

Lo cual puede resultar un tanto confuso inicialmente.

Se efectúa la suma **S1 + S2**. Para hacerlo se llama al operador **+**, que comienza creando la variable **Stmp**, inicialmente vacío (línea 124), lo cual también se reporta a la salida (línea 126).

En la línea 128 se copia en **Stmp** el texto "Hola", llamando al operador **=**, lo cual se reporta en la línea 100.

En la línea 129 se concatena a **Stmp** el texto " mundo", llamando al operador **+=** definido en la línea 37. Para simplificar el código, no se puso ningún mensaje en esta función.

inalmente en la línea 131 se retorna por referencia la variable estática **Stmp** de tipo **TString**, y recién aquí ha terminado la primer suma.

Pero **Stmp** es una variable de tipo **TString**, por lo que ahora se tiene:

```
S3 = Stmp + TString (" del C++");
```

Lo cual es equivalente a escribir:

```
S3 = Stmp.operator+ (TString (" del C++"));
```

Se ejecuta ahora el operador **+**, llamado por la variable **Stmp**. Es decir que ahora, dentro del operador **+**, el puntero **this** apuntará a la variable **Stmp**. Notar que en la línea 128 se tiene: **Stmp = (*this)**; y es por ello que es necesario que el operador **=** soporte cosas como **a = a**, si bien es redundante. En la línea 129 se concatena ahora el texto " del C++", llamando al operador **+=**.

Finalmente se retorna del operador **+**, devolviendo **Stmp** por referencia, la cual se pasa como parámetro al operador **=**, que copia su contenido en la variable **S3**, lo cual se informa en la línea 100. Lo único que falta para terminar con la ejecución de la línea es destruir la variable **TString** temporaria, que contiene " del C++".

En la línea 143 se muestra el string **S3** resultante.

Finalmente, antes de finalizar la ejecución del programa se destruyen las variables **S1**, **S2**, **S3**, y **Stmp**.

El único inconveniente de este programa es la palabra `static`, que podría traer problemas en un sistema multitarea en el cual pudiesen ejecutarse dos operadores `+` simultáneamente, como podría ser UNIX (no es el caso de DOS ni de WINDOWS (en principio)). Para solucionar esto, simplemente basta que el operador `+` devuelva una copia del objeto, en vez del puntero al mismo, es decir:

```
122     TString& TString::operator+(TString &S2)
123     {
124         static TString Stmp;
```

desde ya que esto obligará a crear una copia del objeto, que luego habrá que destruir, con lo cual el programa será más lento.

Objetos temporales

Finalmente queda un último punto a considerar. Muchas funciones reciben parámetros de tipo `char *`, y sería bueno que nuestro tipo de dato (`TString`) también las soporte, es decir, que pueda llamar a estas funciones sin tener que crear una copia del dato. La solución obvia para esto es definir una función que devuelva el puntero a la cadena de caracteres:

Una primer alternativa para hacer esto es la siguiente:

Ejemplo 25.5: Objetos temporales

```
class TString {
private:
    char * Str;
public:
    ...
    char * GetString (void)
    {
        return Str;
    }
    ...
};
```

De esta forma podríamos efectuar operaciones tales como:

```
char *Str;
TString s;
...
strcpy (Str, s.GetString());
```

Esta función ***GetString()*** puede definirse como un operador, para facilitar su uso:

Ejemplo 25.6: Sobrecarga de Casting

```
class TString{
private:
    char * Str;
public:
    ...
    operator char*() (void)
    {
        return Str;
    }
    ...
};
```

De esta forma podríamos efectuar operaciones tales como:

```
char *Str;
TString s;
...
strcpy (Str, s);
```

Este código es correcto. Sin embargo qué pasaría si ejecutásemos un código como el siguiente:

Ejemplo 25.7: Problemas con sobrecarga

```
void MiFuncion (char * Str)
{
    ...
}

{
    /* Este código es incorrecto */
    TString s1,s2;
    s1 = "hola ";
    s2 = "mundo";
    MiFuncion (s1 + s2);
}
```

Esto produciría un error, por la siguiente razón: la llamada a

```
MiFuncion (s1 + s2);
```

es equivalente a llamar a:

```
MiFuncion ( (s1 + s2).GetString() );
```

Analicemos en detalle este código. La llamada a `s1+s2` crea un objeto temporario de tipo `TString`:

```
TString StringTemporario; StringTemporario = s1 + s2;
```

A continuación se llama a la función de casteo, `GetString()` de este objeto temporario:

```
char * PunteroTemporario = StringTemporario;
```

o lo que es lo mismo:

```
char * PunteroTemporario = StringTemporario.operator char* ();
```

o lo que es lo mismo:

```
char * PunteroTemporario = StringTemporario.GetString ();
```

A continuación SE DESTRUYE el StringTemporario, que ya "no se usa".

Finalmente se llama a MiFuncion(), con un puntero que quedó apuntando a basura. Este es un error que puede producir la interrupción del programa, pero si MiFuncion() es relativamente simple, en la práctica esto ocurre muy esporádicamente, con lo cual este error puede ser muy difícil de encontrar, siendo un deber del programador tener mucho cuidado de no cometer este error.

La forma correcta de programar este código es la siguiente:

```
{
    TString s1, s2, s3;
    s1 = "hola ";
    s2 = "mundo";
    s3 = s1 + s2
    MiFuncion (s3);
}
```

Con el objetivo de solucionar este inconveniente, algunos compiladores (NO TODOS) mantienen las variables temporales hasta el final de la instrucción en la cual se los creo, es decir **StringTemporario** será destruida inmediatamente después de llamar a MiFuncion(), pero esto no es standard, y de ninguna manera solucionaría errores como:

```
{
    /* Este codigo es incorrecto */
    TString s1, s2;
    char *Str;
    s1 = "hola ";
    s2 = "mundo";
    Str = s1 + s2;
    MiFuncion (Str);
}
```

Como conclusión a esto simplemente debe tenerse en cuenta que no debe trabajarse con datos internos de variables temporales, ya que su destrucción está regida por el compilador.

Resumen de puntos a tener en cuenta

La creación de objetos con memoria dinámica puede llevar a gran cantidad de confusiones y errores. Sin embargo el tema puede verse muy simplificado si se siguen los siguientes preceptos básicos:

- Crear un constructor de copia.
- Sobrecargar el operador = (asignación)

- Debe haber siempre un constructor, que inicialice todas las variables dinámicas.
- Debe haber un destructor, que libere la memoria utilizada.
- Nunca debe llamarse explícitamente un constructor de la clase. Estos deben llamarse automáticamente por el compilador.
- Todos los operadores que se definan que puedan recibir objetos del mismo tipo que la clase, deben poder soportar ser llamados por sí mismo. Ej: $a = a$; $a += a$; $a = a + a$, etc.
- Todas las funciones-operadores que retornen variables del tipo de la clase deben retornar las mismas por referencia, para evitar crear copias innecesarias del objeto.
- Nunca obtener datos internos de variables temporales. Si no se tiene experiencia, limitar el uso de variables temporales a lo estrictamente necesario.

Capítulo XXVI - Memoria compartida entre objetos

En la sección anterior se vió todas las precauciones que debían tenerse al trabajar con constructores/destructores y memoria dinámica. Sin embargo, a veces se requiere ir más allá, compartiendo información entre variables del mismo tipo. Un ejemplo de esto es una base de datos, que debe ser accedida desde varios módulos. Si la base está en disco, puede accederse a la misma haciendo algunos malabares y usando lockeo de registros, pero si la base está en memoria, no queda otra solución más que compartir un puntero a los datos. En ambos casos será muy útil poder compartir algunos datos más, como por ejemplo la cantidad de registros.

Al momento de abrir una base por primera vez, deberá crearse un conjunto de datos con los cuales manipular la información, los cuales deberán ser compartidos por cada una de las objetos que manipulan la base. Ya que todos deberán compartir estos datos, estos no podrán ser estáticos a cada objeto, es decir, se tendrá algo como:

```
class TBase{
    TBaseData * Data;
};
```

Estos datos deberán ser inicializados en cada objeto, en el momento de llamar al constructor, y destruidos al llamar al destructor.

```
class TBase{
    TBaseData * Data;
    TBase ()
    { Data = ... }
    ~TBase ()
    { delete (Data); }
};
```

Sin embargo, si los datos son compartidos por varias estructuras, no deberán liberarse los datos, mientras alguna estructura los esté utilizando. Concretamente lo que se hace, es definir un flag **Referencias**, que indica la cantidad de veces que se encuentra abierta una misma base, y sobrecargar los operadores de copia y de asignación, para que corrijan este flag:

Ejemplo 26.1: Clase Base - Memoria compartida entre instancias de un objeto

```
class TBase{
    TBaseData * Data;
public:
    TBase ()
    {
        /* Inicializo una nueva base */
        Data = new ...
        Data->Referencias = 1;
    }
    ~TBase ()
    {
        /* Abandono la base actual */
    }
};
```



```
    Data->Referencias--;
    if (Data->Referencias == 0)
        delete (Data);
}
TBase (TBase &NuevaBase)
{
    /* Abandono la base actual */
    Data->Referencias--;
    if (Data->Referencias == 0)
        delete (Data);
    /* Comienzo la nueva base */
    Data = NuevaBase.Data;
    Data->Referencias++;
}
TBase& operator= (TBase &NuevaBase)
{
    ...
}
};
```

En este caso, la función ***operator=()*** será similar al constructor ***TBase(TBase &)***, con la única diferencia de que debe tenerse cuidado si se efectúan llamadas como ***Base = Base;***

Esta forma de compartir información entre varias variables es muy poderosa y útil, pero también peligrosa. Si cualquiera de estas funciones llega a fallar, el error puede ser muy difícil de localizar.

Si se trabaja con un programa ***multithread***, debe realizarse lockeos antes de leer o modificar la variable ***Referencias***, entre otras precauciones. Pero este tema queda fuera del alcance de este libro.

Capítulo XXVII - Sobrecarga de operadores new y delete

Una lectura atenta de la sección de operadores muestra que las palabras reservadas **new** y **delete** son consideradas operadores, y por lo tanto pueden ser sobrecargados. Sin embargo hay algunas restricciones:

- Ambos operadores son estáticos (aún cuando no se lo especifique).
- Puede existir un único operador delete, pero puede existir más de un operador new.
- new debe recibir como primer parámetro un dato de tipo size_t y debe devolver un puntero a void.
- delete debe recibir como primer parámetro un puntero de tipo void, y no puede devolver nada.
- new se llama ANTES de reservar memoria para el dato, y por lo tanto no puede tener acceso a las variables de la clase (es por esto que es de tipo static).

Algunos ejemplos de esto son los siguientes:

Ejemplo 27.1: Sobrecarga de new y delete

```
class A {
public:
    void * operator new (size_t size);
    void operator delete (void *dato);
};

class B {
public:
    void * operator new (size_t size);
    void * operator new (size_t size, int flags);
    void operator delete (void * dato, int flags);
};
```

Puede sobrecargarse new y delete con propósitos tales como por ejemplo, en una lista dinámica, reservar con un único malloc() memoria para el encabezado del elemento de la lista como para el dato. Veamos un ejemplo de esto:

Ejemplo 27.2: Sobrecarga de new y delete (2)

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

```
#include <malloc.h>
#include <iostream.h>
#include <string.h>
#include <conio.h>

class TNode{
private:
    void *Dato;
    TNode * Siguiente;
public:
    void *operator new(size_t SizeNode);
    void *operator new(size_t SizeNode, size_t DatoSize);
    void operator delete (void *Dato)
    { free (Dato); }
    TNode (void)
    {
        Dato = this+sizeof (TNode);
        Siguiente = NULL;
    }
};

// Este es el constructor que se llamará para reservar memoria para el nodo y el
// dato.
void* TNode::operator new(size_t SizeNode, size_t DatoSize)
{
    cout << "Reservando " << SizeNode << "+" << DatoSize << " bytes\n";
    return malloc (SizeNode+DatoSize);
}

// Este es el constructor que se llamaría normalmente al llamar a 'new TNode;'
void* TNode::operator new(size_t SizeNode)
{
    cout << "Reservando " << SizeNode << " bytes\n";
    return malloc (SizeNode);
}

void main (void)
{
    TNode *Nodo;

    Nodo = new (50) TNode;
}
```



No suele ser una buena idea sobrecargar el operador new en objetos que sean exportados o estén disponible para el uso del programador, y si se lo hace, no debe modificarse la cantidad de memoria que reservan. El problema radica principalmente en su uso en listas y vectores. Están disponibles varias implementaciones de listas y vectores que permiten almacenar cualquier tipo de objeto (*templates*), muchas de las cuales reservan memoria para el objeto sobrecargando el operador new o realizando operaciones tales como: *malloc (sizeof (Objeto))*.

Veamos un ejemplo que puede ayudar a refrescar algunos conceptos. Construyamos un objeto capaz de almacenar una lista de objetos de cualquier tipo. Desde ya, existen muchas tipos de implementaciones posibles, cada una de las cuales tendrá sus ventajas y desventajas. Elegiremos como base para este ejemplo un vector:

Ejemplo 27.3: Template de vectores

```
template <class Type> class TVector
{
    private:
        Type *Data;           // Vector con los datos
        unsigned Total;       // Número total de elementos almacenados
    public:
        /* Constructor de la clase */
        TVector (void)
        { Data = NULL; Total = 0; }
        /* Destructor de la clase */
        ~TVector ();
        /* Agregar un elemento a la clase */
        void Add(Type &t);     // Agregar un nuevo elemento al final del vector
};
```

Este objeto permitirá almacenar un vector de objetos del tipo *Type*.

Surge aquí un problema: ¿Cómo hacemos para insertar un dato en esta lista?

Hay dos soluciones cada una de las cuales tiene sus ventajas y desventajas:

Alternativa 1:

```
/* Agregar un objeto a la lista (alternativa 1) */
template <class Type> void TVector<Type>::Add(Type &t)
{
    Type *NewData;

    /* Reservo memoria para los nuevos datos */
    NewData = new Type[Total+1];

    /* Copio los datos existentes al nuevo vector */
    if (Total)
        for (unsigned i = 0; i < Total; ++i)
            NewData[i] = Data[i];
    /* Inserto el nuevo dato */
    NewData[Total++] = t;

    /* Destruyo los datos originales y establezco el nuevo vector de datos */
    delete []Data;
    Data = NewData;
}
```

El destructor correspondiente de la clase será:

```
template <class Type> TVect<Type>::~TVect ()
{
    /* Destruir los objetos */
    if (Data)
        delete []Data;
}
```

```
}
```

Esta alternativa es transparente y segura, independientemente del tipo de objeto que se esté almacenando. Incluso no habría inconveniente en que el objeto original almacenado en la lista sobrecargase los operadores **new** y/o **delete**. La contrapartida es que cada vez que se inserte un elemento, se construirá una copia de todo el vector y se llamarán a tantos constructores y destructores como elementos haya en la lista, con lo cual se vuelve bastante ineficiente.

Alternativa 2:

Si la memoria se reservase utilizando **malloc()**, podría modificarse el tamaño del vector utilizando la función **realloc()**, sin que se deban construir copias y destruir los objetos ya almacenados.

```
/* Agregar un objeto a la lista (alternativa 2) Versión incorrecta */
template <class Type> void TVector<Type>::Add(Type &t)
{
    Type *NewData;      /* Reservo memoria para el nuevo dato */
    NewData = (Type *)realloc (Data, sizeof (Type)*(Total+1));

    /* Aumento el total */
    ++Total;
}
```

Pero en este ejemplo no se llamará al constructor del nuevo elemento, y para que todo funcione bien, debo garantizar que esto suceda. Podría pensarse que simplemente podría llamarse al constructor del objeto en la forma:

```
Data[Total].Type();
```

Pero esto no está permitido. La solución consiste en sobrecargar el operador **new**, de forma tal que se llame al constructor, pero que este no reserve memoria. Ya que no hay forma de sobrecargar el constructor **new** de la clase original, deberá sobrecargarse este operador en forma global.

```
static inline void* operator new (size_t, void *ptr)
{
    return ptr;
}
```

La palabra reservada **static** se utilizó para que este operador sólo esté disponible en los archivos donde se utilice esta clase. La palabra reservada **inline** se utiliza para que el operador se expanda en línea, en lugar de ser llamado como una función. Notar que este operador no reserva memoria, como sería de esperar.

Ahora bien escribamos la versión final del método Add.

```
static inline void* operator new (size_t, void *ptr)
{
    return ptr;
}

/* Agregar un objeto a la lista (alternativa 2) Versión incorrecta */
template <class Type> void TVector<Type>::Add(Type &t)
{
```

```
Type *NewData;      /* Reservo memoria para el nuevo dato */
NewData = (Type *)realloc (Data, sizeof (Type)*(Total+1));

/* Contruyo el objeto */
::new ((void*) (Data+Total)) Type;

/* Aumento el total */
++Total;
}
```

El destructor correspondiente de la clase será:

```
template <class Type> TVect<Type>::~TVect ()
{
    /* Destruir los objetos */
    if (!Data)
        return;
    /* Llamar a los destructores para cada uno de los objetos insertados */
    Type *p;
    for (p=Data;Total;--Total,++p)
        Data->~Type();
    /* Liberar la memoria */
    free (Data);
}
```

Esta alternativa es más eficiente que la anterior, ya que no deben construirse y destruirse objetos innecesariamente. La contrapartida es que el objeto original no puede sobrecargar los operadores new y delete.

En las clases de vectores provistas por los compiladores de **Borland C++**(Compilador Borland C++ 5.00) se utiliza la primer alternativa. En las clases provistas por **Microsoft Visual C++** (Compilador Microsoft Developer Studio - Visual C++ 4.0) se utiliza la segunda alternativa. Ambas librerías reservar memoria para varios elementos cada vez que esto es necesario, de tal forma que no se requiere efectuar estas operaciones cada vez que se inserta un nuevo elemento en el vector. Al trabajar se esta forma, en la alternativa 1 los objetos serán construidos por adelantado, al reservarse memoria para los mismos, en tanto que en la segunda alternativa serán construidos al ser insertados efectivamente en la lista.

Capítulo XXVIII - Manejo de excepciones

El lenguaje C++ define una nueva y muy poderosa forma de manejar las excepciones, entendiendo por excepciones toda instrucción que no puede ser ejecutada por el procesador. Ejemplos de excepciones son divisiones por cero, overflow, underflow, accesos a memoria en zonas prohibidas, en algunos sistemas operativos acceso a datos desalineados en memoria, etc. Las excepciones son manejadas normalmente por el sistema operativo, quien en la mayoría de los casos implementa la interrupción inmediata del programa.

En C++ existen varias palabras reservadas utilizadas en el manejo de excepciones, entre las cuales se encuentran **try** y **catch**, y que permiten muy fácilmente implementar el manejo de excepciones.

Ya que la forma en que se producen y manejan las excepciones está muy relacionado con el sistema operativo, si bien ANSI C++ define la forma en que deben procesarse estas, pueden haber variantes. Por ejemplo, **Microsoft** define además las palabras reservadas **__try** y **__catch**, con nuevas funcionalidades.

Nota: muchas de las palabras reservadas utilizadas en esta sección están redefinidas por **Microsoft** con un doble underscore; Ej: **try**, **catch**, **except** y **leave** está definidas como **__try**, **__catch**, **__except** y **__leave**.

La forma en que se implementa el manejo de excepciones es como sigue:

```
try{
    código
}
catch (argumento)
{
    ...
}
```

El código encerrado en **try** es cualquier código en C++ cuyas excepciones se desean controlar y que puede incluir otros manejadores de excepciones.

La sentencia **catch** puede recibir un único argumento, y define la forma en que la excepción debe ser controlada. Pueden definirse más de una sentencia catch. Por ejemplo:

Ejemplo 28.1: Manejo de excepciones

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

int Division (int a, int b)
{
    return a / b;
}

int main(void)
{
```

```
int r;

cout <<"Comenzando\n";

try{
    r = Division (10, 0);
    cout << "r=" << r << "\n";
}
catch (char * str)
{
    cout <<"Error:" << str << "\n";
}
catch (int x)
{
    cout <<"Error:" << x << "\n";
}
catch (...)
{
    cout << "Error desconocido\n";
}
cout << "Fin del programa";
return 0;
}
```

La salida de este programa es:

```
Comenzando
Error desconocido
Fin del programa
```

El programa comienza creando una variable entera llamada *r*. A continuación se llama a la función **División**, pero la división por cero generará automáticamente una excepción. Si no se hubiese definido el controlador de excepciones, esta sería controlada por el sistema operativo, y el programa sería interrumpido. Pero en este caso se verificará primero si alguno de los manejadores de excepciones definidos a continuación del bloque **try**, puede manejar esta excepción. La última de ellas **catch(...)** es un controlador genérico, por lo que se imprimirá "Error desconocido" en pantalla. El programa continuará a continuación desde el final del bloque **try**, con lo cual la línea `cout << "r=" << r << "\n";` no será ejecutada.

Notar que en el programa se definen 3 sentencias **catch**, o sea 3 controladores de excepciones. ¿Cuál de ellos será llamado? **Rta:** El primero que se pueda. Es decir que si se hubiese definido **catch(...)** antes que los otros dos, estos últimos jamás serían llamados.

Puede utilizarse la palabra reservada **throw** para forzar al generación de una excepción. Por ejemplo podríamos haber escrito nuestra función *Division* como sigue:

Ejemplo 28.2: Generación de excepciones

```
int Division (int a, int b)
{
    if (b == 0)
        throw "División por cero";
    return a / b;
}
```



```
}
```

En cuyo caso la salida del programa hubiese sido:

```
Comenzando
Error: División por cero
Fin del programa
```

Es posible atrapar también otras excepciones, como por ejemplo accesos a zonas prohibidas. Por ejemplo, el siguiente programa no sería interrumpido por el sistema operativo, ni sería incorrecto:

Ejemplo 28.3: Manejo de excepciones (2)

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

void Funcion (int *p)
{
    *p = 2;
}

int main(void)
{
    int r;

    cout << "Comenzando\n";

    try{
        Funcion(NULL) ;
    }
    catch (...)
    {
        cout << "Error desconocido\n";
    }
    cout << "Fin del programa";
    return 0;
}
```

La salida de este programa es:

```
Comenzando
Error desconocido
Fin del programa
```

¿Qué sucedería si hubiesemos definido la función Función como sigue?

Ejemplo 28.4: Excepciones y variables locales

```
class T
{
    public:
```

```
T() { cout <<"Construyendo T\n"; }
~T() { cout <<"Destruyendo T\n"; }
};

void Funcion (int *p)
{
    T t;

    *p = 2;
    cout << "*p=" << *p;
}
```

Rta: La salida del programa hubiese sido:

```
Comenzando
Construyendo T
Destruyendo T
Error desconocido
Fin del programa
```

Esto ilustra otra de las características que hacen la programación en C++ muy poderosa: **antes de llamarse a un controlador de excepciones se llama primero a los destructores de todas las variables locales**. Esto permitiría a un programa, por ejemplo, cerrar correctamente todos los archivos abiertos, y finalizar todas las tareas pendientes antes de reportar un error fatal o interrumpir el proceso.

Es posible también definir controladores para excepciones particulares. Para ello se utiliza la palabra reservada **except**. Veamos un ejemplo:

Ejemplo 28.5: Manejo de excepciones (except)

```
void main (void)
{
    // En Microsoft debe usarse __try en vez de try
    try{
        int a = 2;
        int b = 0;
        a = a / b;
    }
    // En Microsoft debe usarse __except en vez de except
    __except (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)
    {
        /* GetExceptionCode() no forma parte del standard ANSI
        Las definiciones de excepciones, tales como
        EXCEPTION_INT_DIVIDE_BY_ZERO
        pueden variar de un sistema a otro. */

        cout << "Entero dividido cero\n";
    }
}
```



Está prohibido utilizar una instrucción *goto* de afuera hacia adentro de un bloque *try*, y viceversa. Para salir inmediatamente de un bloque *try* puede utilizarse la palabra reservada *leave*.

Ejemplo 28.6: Manejo de excepciones (leave)

```
try{
    int a, b;
    a = 2;
    b = 0;
    if (b == 0)
        leave;
    a = a / b;    // Si b es igual a 0 esta instrucción jamás será ejecutada.
}
catch (...){
    cout <<"Error desconocido";    // Este controlador tampoco será llamado.
}
```

Existen varias funciones relacionadas con excepciones, tales como ***RaiseException()***, pero no forman parte del standard y sus definiciones pueden variar mucho de un compilador/sistema operativo a otro. Para más información consultar el manual del compilador.

APÉNDICE I - Palabras reservadas de C

Palabras reservadas de C

auto	do	if	struct
asm	double	int	switch
break	else	long	typedef
case	enum	register	union
char	extern	return	unsigned
const	float	short	void
continue	for	signed	volatile
default	goto	sizeof	while
		static	

APÉNDICE II - Palabras reservadas de C++

Palabras reservadas de C++

asm	double	namespace	switch
auto	dynamic_cast	new	template
bad_cast	else	operator	this
bad_typeid	enum	private	throw
break	except	protected	try
case	extern	public	type_info
catch	finally	register	typedef
char	float	reinterpret_cast	typeid
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	while
			xalloc

Apéndice III - Caracteres reservados

Carácter	Representación	Valor dec.	Valor hexa	Secuencia escape
Carácter nulo	NUL	0	0	\0
Alert	BEL	7	7	\a
Backspace	BS	8	8	\b
Tabulación horizontal	HT	9	9	\t
Nueva línea	NL	10	A	\n
Tabulación vertical	VT	11	B	\v
Formfeed	FF	12	C	\f
Retorno de carro	CR	13	D	\r
Comilla doble	"	34	22	\"
Comilla simple	'	39	27	\'
Signo de pregunta	?	63	3F	\?
Backslash	\	92	5C	\\
Nro octal	ooo	--	--	\ooo
Nro hexa	hhh	--	--	\xhhh

Apéndice IV - EL PREPROCESADOR

Macros predefinidas

<code>__DATE__</code>	Fecha de compilación (Formato: Mmm dd yyyy)
<code>__FILE__</code>	Archivo actual, incluyendo path
<code>__LINE__</code>	Línea actual. Este valor puede ser alterado con la directiva <code>#line</code>
<code>__TIME__</code>	String con la hora de compilación (Formato: hh:mm:ss)
<code>__TIMESTAMP__</code>	Fecha y hora de la última modificación de código fuente.

Si bien no forma parte del standard, la mayoría de los compiladores C++ definen también la macro `__cplusplus`.

Directivas del preprocesador

La directiva `#if`, junto con `#else`, `#elif` y `#endif` controlan la compilación de una parte del programa. Si la expresión que sigue a `#if` resulta verdadera, la porción de código siguiente será compilada. En caso contrario, será ignorada. Existen otras directivas tales como:

<code>#elif</code>	permite evaluar una nueva expresión.
<code>#ifdef</code>	CTE es similar a <code>#if defined(CTE)</code>
<code>#ifndef</code>	CTE es similar a <code>#if !defined(CTE)</code>
<code>#undef</code>	permite anular una definición de una constante.
<code>#define</code>	permite definir macros y constantes
<code>#import</code>	es utilizada para incluir información de una librería de tipos. Es utilizada al trabajar con componentes.
<code>#line</code>	permite modificar el número de línea actual, para la macro <code>__LINE__</code>
<code>#include</code>	es utilizada para incluir un archivo dentro de otro. Si se utilizan los símbolos <code><></code> se buscará el archivo en el o los directorios <code>include</code> . Si se utiliza <code>""</code> se buscará el archivo en primer lugar en el directorio actual, y luego en los directorios <code>include</code> .
<code>#pragma</code>	permite modificar opciones de compilación, tal como activar y desactivar warnings, e indicar al linker que utilice determinadas librerías.

Apéndice V - Mezclando código C y C++

Existen algunos inconvenientes al mezclar librerías escritas en C con otras escritas en C++. Si bien el C++ constituye una extensión del lenguaje C, existen algunas diferencias que deben tenerse en cuenta. Desde ya que no es posible utilizar clases u objetos dentro de código en C, pero tampoco puede llamarse directamente a funciones en C desde código en C++, ni viceversa. El problema radica en que la forma en que se implementa la llamada a una función en C++ es diferente a la que se utiliza en C.

Si se dispone el código fuente de las librerías, pueden transformarse los archivos en .c en archivos .cpp, obligando así al compilador a utilizar el standard de C++. Esto puede resultar un tanto tedioso, y sólo es posible si se dispone del código fuente de las librerías, lo cual no siempre es posible.

Existe una única forma para solucionar este inconveniente, y especificar a en todas las funciones que vayan a compartirse (tanto en librerías C como C++) que debe utilizarse el standard C para llamada a funciones. Esto se hace anteponiendo extern "C" a la declaración de toda función, en los archivos .H, al compilar el código en C++.

El preprocesador define una constante llamada __cplusplus que permite saber si se está compilando en lenguaje C o en C++. Para lograr fácilmente una compatibilidad total, basta agregar a todos los archivos .H a compartir entre código C y C++ las siguientes líneas:

Ejemplo A.5.1: Mezclando código C y C++

```
/* Comienzo del archivo .h */
#ifdef __cplusplus
    extern "C" {
#endif
```

... Toda las declaraciones de funciones del archivo.h ...

... No hay ningún inconveniente en que estén también aquí las declaraciones de tipos de datos, variables, constantes, macros, etc ...

```
#ifdef __cplusplus
}
#endif
/* Fin del archivo .h */
```

De esta forma, ya sea que el archivo .h sea incluido en un archivo .c o .cpp, se definirá la que las funciones deberán llamarse utilizando el standard C.

Es decir que para mezclar librerías y código en C con C++ basta con agregar estas líneas al comienzo y al final de todo archivo .h.

Ya que existe mucho código escrito en C, es muy probable que en programas de gran envergadura se deban mezclar librerías en ambos lenguajes. Incluir código C en programas en C++ es muy simple, como ya se vió. Pero utilizar librerías en C++ dentro de programas en C no lo es, y generalmente resulta muy tedioso y conduce a programas poco claros. La mejor forma de hacer esto es programar una capa de conversión de la librería C++ a una librería en C. Por ejemplo, si tuviésemos que adaptar nuestra clase cola para trabajar en un programa C, podríamos hacer lo siguiente:

Ejemplo A.5.2: Mezclando código C y C++ (2)

```
/* Código en C */
{
    void *pCola;
    InicializarCola (&pCola); /* Inicializo la cola */
    ColaPush (pCola, 20); /* Insertar el número 20 en la cola */
}

/* Código en C++: Adaptación de la clase cola a un programa C */
void InitCola (void **pCola)
{
    TCola *Cola;
    Cola = new TCola; // Creo la cola
    *pCola = (void *)Cola; // Devuelvo la cola
}

int ColaPush (void *pCola, int n)
{
    TCola *Cola;
    Cola = (TCola *)pCola; // Convierto el puntero a void en un puntero a la
clase
    return Cola.Push (n); // Inserto un elemento en la cola
}
```

Apéndice VI - Resumen de funciones especiales de C++

A continuación se muestra una tabla que resume todas las funciones especiales que pueden existir en una clase, y su comportamiento.

Función	pueden heredarse	pueden ser virtuales	pueden retornar un valor	funciones miembro o amigas (friend)	generadas por defecto
constructor de copia	no	no	no	miembro	si
constructor por defecto	no	no	no	miembro	si
otros constructores	no	no	no	miembro	no
destructor	no	si	no	miembro	si
conversión	si	si	no	miembro	no
= (asignación)	no	si	si	miembro	si
operador=	si	si	si	cualquiera	no
()	si	si	si	miembro	no
[]	si	si	si	miembro	no
->	si	si	si	miembro	no
new	si	no	void*	miembro (static)	no
delete	si	no	void	miembro (static)	no
otros operadores	si	si	si	cualquiera	no
otras funciones miembro	si	si	si	miembro	no
otras funciones friend	no	no	si	friend	no

PARTE IV – WINDOWS API

Capítulo I - Introducción a la programación bajo Windows

Introducción

El sistema operativo Windows plantea una nueva forma de interacción de los programas con el usuario, donde el modelo de programación utilizado en DOS resulta poco adecuado. En primer lugar, no habrá un único programa en ejecución, como lo había bajo DOS, sino un conjunto de programas, mejor llamados aplicaciones (desde el punto de vista del sistema operativo, procesos) los cuales deberán interactuar entre sí. Se suma a esto que toda aplicación deberá procesar (en tiempo real) una serie muy grande de eventos, tales como teclado, mouse, timers, etc. Esta situación lleva al desarrollo de programas extensos, y plantea la necesidad de una programación clara y ordenada. Es por ello que la Programación Orientada a Objetos resulta ideal para desarrollar programas bajo Windows, si bien no es indispensable.

Esta es la razón por la cual varias empresas como Borland y Microsoft han desarrollado bibliotecas muy extensas de objetos, que crean niveles de abstracción superiores, de tal forma que pocas veces se requiere descender hasta el nivel más bajo e interactuar directamente con el API (Application Program Interface) de Windows, tema que desarrollaremos brevemente a continuación.

Con el objetivo de facilitar la familiarización con el ambiente Windows, se darán en este capítulo definiciones poco rigurosas, que serán redefinidas en capítulos posteriores.

La filosofía de programación Windows

Nos centraremos en la programación 32 bits (Windows 95/98/2000/NT). Los sistemas operativos Windows de 16 bits (Windows 3.1 y anteriores), además de ser muy poco utilizados en la actualidad, presentaban una serie de inconvenientes para la programación, como el reducido tamaño de la cola (stack), de tan sólo algunos KBytes.

La programación bajo el sistema operativo Windows, plantea una nueva forma de diseñar y concebir los programas. Este nuevo estilo de programación, lejos de ser complejo como lo puede parecer en un comienzo, resulta muy práctico al momento de diseñar programas de gran envergadura. Pero un programa para Windows debe ser pensado en una forma completamente distinta a un programa bajo DOS, fundamentalmente porque el mismo no estará corriendo en forma independiente, sino en un sistema multitarea nativo, y deberá interactuar tanto con otros programas como con el sistema operativo mismo. Este hecho da origen a un nuevo paradigma de programación, conocido como **Programación Orientada a Eventos**. Un evento podría ser, por ejemplo, la presión de una tecla, la llegada de datos a un dispositivo o un comando para que una ventana se redibuje. Este nuevo modelo de programación puede ser aplicado también en otros ambientes, y no constituye una ruptura con los modelos de programación estructurada ni orientada a objetos (como ocurría entre la primera y la programación imperativa) sino una extensión de los mismos.

Un programa para Windows puede ser escrito tanto en C como en C++. Sin embargo, por las características del sistema, el lenguaje C++ resulta especialmente adecuado por su capacidad de trabajar con objetos y encapsulamiento del código, que permiten manejar niveles superiores de abstracción e independizarse de la implementación concreta de los algoritmos.

De hecho, como ya se verá, Windows mismo ya provee un primer nivel de abstracción, y varios "objetos" básicos, con los que se puede trabajar a través de *handles* (identificadores : enteros de 32 bits), recordemos que en C no existen objetos propiamente dichos. Pero para comprender la necesidad de este nuevo paradigma de programación, veamos en qué difiere la programación bajo Windows de la programación habitual bajo DOS. Y para ello, antes es necesario saber qué es "Windows".

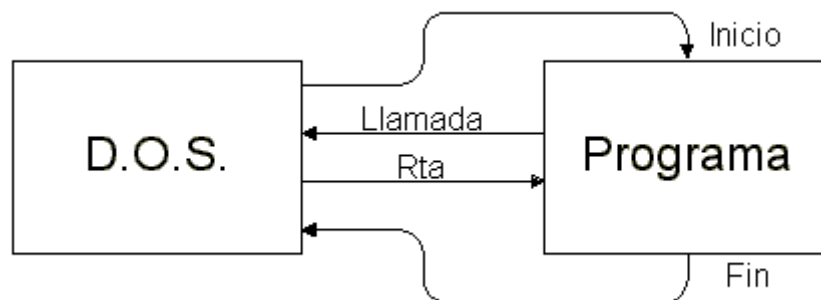
¿Qué es Windows?

Desde el punto de vista del programador, Windows (al igual que DOS) no es más que un gran conjunto de funciones, que puede utilizar un programa. Es a este conjunto de estas funciones al que se conoce como API de Windows (como mencionamos antes, son las siglas de Application Program Interface y están programadas en C). Esta concepción todavía sigue siendo muy limitada a lo que era el DOS. En el ambiente Windows es necesario ampliarla diciendo que es también un proceso o conjunto de procesos que pueden comunicarse con nuestra aplicación o proceso.

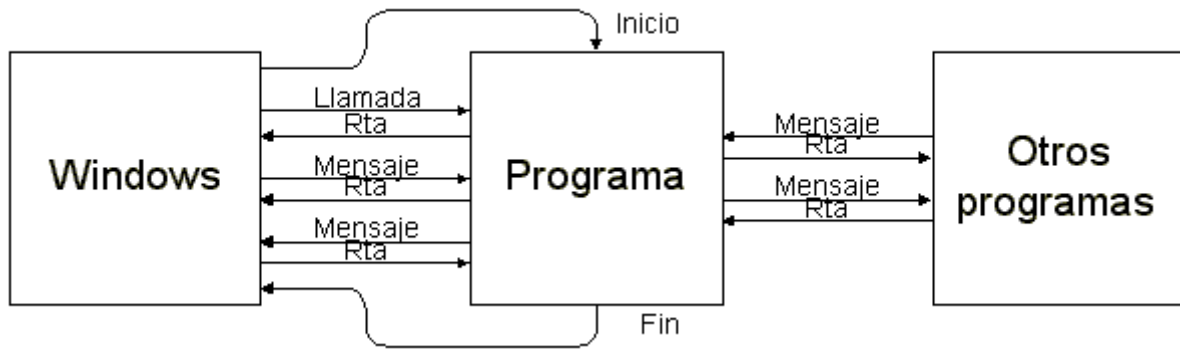
Muy bien, pero ¿qué hay de nuevo?

Para responder esta pregunta, veamos un ejemplo, que puede ayudar a entender la idea. Si alguien dijese que tiene un código en C/C++, la primer pregunta que un programador, habituado a trabajar en DOS, probablemente haría es ¿QUÉ HACE?. En lugar de ello, un programador para WINDOWS podría preguntar ¿A QUÉ RESPONDE? ó ¿QUÉ IMPLEMENTA? Esto marca la principal diferencia de la programación entre ambos ambientes. Sí bien un código en particular puede estar pensado para "hacer algo", un programa (o aplicación) bajo Windows no se concibe como un programa que "haga", sino como un conjunto de funciones, muchas veces agrupadas en objetos, que responden a eventos externos, generalmente mensajes, pudiendo estos generar nuevos eventos.

Un programa bajo DOS se compone de una serie de funciones, que incluyen llamadas al sistema operativo, del cual recibe respuestas. Un ejemplo de esto es: la llamada: "Abrir un archivo de disco", de la cual se recibe como respuesta un identificador con el cual acceder al mismo, o un código de error.



Una aplicación bajo Windows debe concebirse como un programa que puede enviar *mensajes* a Windows o a otras aplicaciones, de los cuales recibirá *respuestas* (ya veremos más adelante qué es esto), pero también puede recibir mensajes de Windows u otras aplicaciones, a los cuales debe responder. Siguen existiendo las llamadas convencionales al sistema operativo, en la forma que existían en DOS.

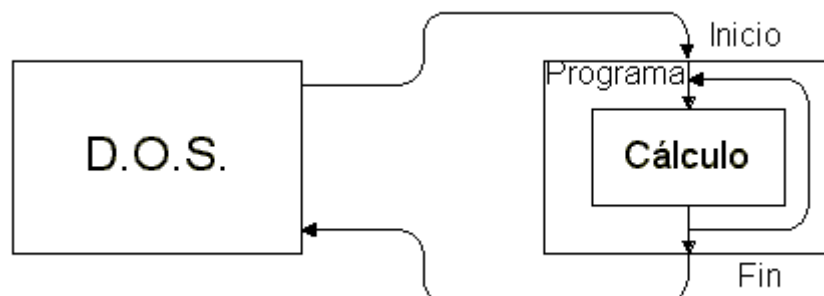


Es precisamente la *interfaz* (API) quien define a los miembros de la familia de sistemas operativos basados en Windows. Desde el punto de vista del programador, se trata de un conjunto de rutinas (funciones, estructuras, mensajes e interfaces consistentes y uniformes para todas las plataformas de 32 bits basadas en Windows) que llevan a cabo servicios de bajo nivel ejecutados por el sistema operativo y las aplicaciones usan para interpretar, enviar y responder mensajes. Mediante su uso, se pueden desarrollar aplicaciones que corran con éxito en todas esas plataformas siendo, además, capaces de sacar ventajas de las características y capacidades únicas de cada una de ellas. Las distintas implementaciones de los programas dependen de las capacidades de las características subyacentes en cada una de las plataformas; siendo la diferencia más notable que algunas funciones llevan a cabo sus tareas sólo en las plataformas más poderosas (por ejemplo, las funciones de seguridad sólo responden completamente bajo los sistemas operativos Windows NT). Muchas de las diferencias son, en realidad, limitaciones del sistema, como las restricciones sobre la cantidad de ítems que una dada función puede manejar.

Queda más claro ahora qué se debe entenderse cuando se dice que los *sistemas operativos basados en Windows* están manejado por eventos. Cada vez que un evento ocurre, el sistema operativo envía mensajes a los procesos relevantes. Por lo tanto, un *programa basado en Windows* recibe mensajes, los interpreta y toma la acción apropiada.

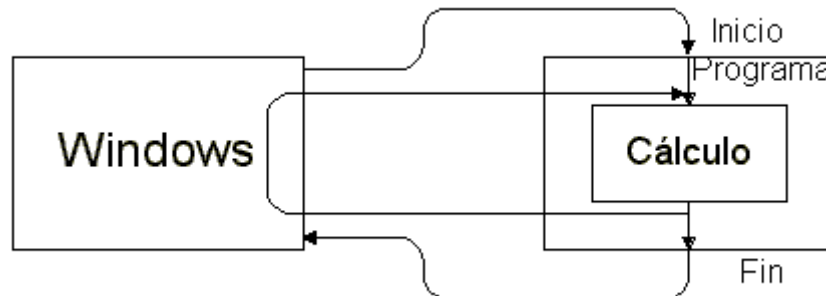
En este contexto, un mensaje es entonces un paquete de datos usado para comunicar una información o proveer una respuesta. Los mensajes pueden pasarse entre el sistema operativo y una aplicación, entre diferentes aplicaciones, entre hebras dentro de una aplicación y entre ventanas dentro de una aplicación.

Veamos otro ejemplo. Supongamos un programa que requiera realizar un lazo en forma repetida para realizar un cálculo complejo, el cual requiere mucho tiempo para completarse. En un programa DOS, la ejecución del programa seguiría un esquema como el siguiente:



Bajo Windows, este esquema, (si bien puede utilizarse), haría que el programa deje de responder a mensajes externos mientras está realizando el cálculo (hay formas de solucionar esto, volveremos sobre esto mas adelante). Un programa

bajo Windows debe en todo momento responder a eventos externos, y si el lazo requiere mucho tiempo para su ejecución, deben tomarse medidas para la aplicación continúe respondiendo a mensajes externos. Hay básicamente 3 formas de hacer esto, las cuales se verá más adelante. La más simple consiste en implementar el ciclo de tal forma que se cierre dentro del sistema operativo, y fuera de la función que lo está realizando.



Para ver la necesidad de cambiar la forma de programación convencional a este nuevo esquema intentemos resolver el siguiente problema:

Se requiere diseñar un programa (para DOS) que imprima en pantalla los caracteres presionados. Al mismo tiempo, el programa debe emitir un 'beep' cada un segundo y debe controlarse constantemente el estado de una máquina. Si esta llega a fallar debe notificarse al operario inmediatamente. Finalmente si se presiona ESC el programa debe terminar.

El programa se podría concebir de muchas formas, pero una bastante clara consiste en separar la verificación de los eventos (timer, teclado, verificación de alarmas) con el procesamiento de los mismos. Podría escribirse una función diferente para procesar cada evento; pero si estas dos tareas (detección de eventos y respuesta a los mismos) fuesen desarrollados por dos grupos independientes, y ambas partes previeran un crecimiento importante, una forma clara de solucionar esto es el desarrollo de una única función genérica de procesamiento de eventos. Así, una forma de escribir el programa es la siguiente:

```
/* Archivos a incluir */
#include
#include
#include
#include
#include
#include

/* Mensajes procesables */
#define COMENZANDO 1
#define FINALIZANDO 2
#define TECLA 3
#define TIMER 4
#define ALARMA 5
int ProcesarMensaje (int mensaje, int param)
{
    switch (mensaje)
    {
        case COMENZANDO:
            clrscr();
            printf ("Comenzando\n");
            break;
```

```
    case FINALIZANDO:
        printf ("\nPrograma finalizado\n");
        break;
    case TECLA:
        if (param == 27)
            return 1;
        putch (param);
        break;
    case TIMER:
        sound (440);
        delay(100);
        nosound ();
        break;
    case ALARMA:
        printf ("ALARMA %u\n", param);
        break;
}
return 0;
}
int VerificarMaquina(void)
{
    /* Verificar el estado de la máquina y devolver cualquier
       error que se detecte */
    return 0; /* Devuelvo OK */
}
void main(void)
{
    int salir = 0;
    time_t t1;
    int Alarma = 0;

    ProcesarMensaje (COMENZANDO, 0);
    do{
        /* Verifico el estado del teclado */
        if (kbhit())
            salir = ProcesarMensaje (TECLA, getch());
        /* Verifico si se debe emitir un timer */
        if (time(NULL) != t1)
        {
            t1 = time (NULL);
            salir = ProcesarMensaje (TIMER, 0);
        }
        /* Verifico el estado de la maquina */
        Alarma = VerificarMaquina();
        if (Alarma)
            salir = ProcesarMensaje (ALARMA, Alarma);
    }while (!salir);
    ProcesarMensaje (FINALIZANDO, 0);
}
```

Esta nueva forma de programación se la conoce con el nombre de **Programación Orientada a Eventos**, y las funciones que procesan los mensajes (como *ProcesarMensaje()* en nuestro ejemplo) se denominan funciones **callback**. Queda claro ahora porqué la función *ProcesarMensaje* debe requerir un tiempo de procesamiento despreciable, o impedirá procesar otros eventos en forma "simultánea".

En Windows, el ciclo de la función **main** encargado de verificar y generar los eventos está fundamentalmente a cargo del sistema operativo, quien notifica a nuestra aplicación los eventos ocurridos mediante una cola de mensajes. La función *ProcesarMensaje()* debe procesar este mensaje y retornar el control a Windows inmediatamente.



En resumen, todo programa bajo Windows debe concebirse como una serie de componentes (funciones en C) que interactúan entre sí a través de mensajes y respuestas.

Capítulo II - Comenzando

Comenzando

Un programa básico pensado para correr en sistemas operativos basados en Windows tiene generalmente tres elementos primarios:

- una ventana
- un "bombeador" de mensajes (en realidad, un lazo de mensajes)
- un procesador de mensajes.

Pese a que una ventana es pensada comúnmente en términos de un despliegue visual, también puede estar definida como no visible; por ejemplo, si se está programando una aplicación sin interfaz de usuario, generalmente se utiliza una ventana no visible para procesar mensajes; cada ventana tiene un identificador de ventana (*hwnd*) asociado con un procesador de mensajes que identifica los mensajes para esa ventana.

El "bombeador" de mensajes es un lazo simple que corre continuamente mientras corre la aplicación recibiendo mensajes y despachándolos a un procesador de mensajes apropiado; cuando ocurren eventos que generan mensajes, el sistema operativo coloca los mensajes en una cola de mensajes.

En Windows es posible crear en un mismo programa varios puntos o hebras de ejecución (threads) en un mismo proceso o aplicación. Esta constituye una característica muy poderosa que pocos sistemas operativos proveen. Al crear un programa con varias threads (multithread) deben realizarse algunas correcciones o aclaraciones a las definiciones de este capítulo, las cuales se verán más adelante.

Windows define una serie de elementos o pseudo-objetos (no confundir con objetos de C++, en C no existen objetos, y por lo tanto no hay forma de que Windows los provea), que pueden manipularse a través de handles (identificadores), que en la práctica son, como mencionamos, enteros de 32 bits. Entre los más comunes se encuentran:

HINSTANCE	Instancia de un programa. A partir de ahora hablaremos de un proceso. Windows identifica cada proceso que está corriendo con un número entero.
HWND	Ventana. (ya lo mencionamos mas arriba)
HICON	Icono
HDC	Contexto para dibujar en pantalla
HMENU	Menú de una aplicación
HBITMAP	Bitmap
HPEN	Lápiz
HBRUSH	Trama para pintar

De todos estos *handles*, el fundamental es *HWND*, utilizado para identificar una ventana. Una ventana puede recibir y transmitir mensajes de/hacia a otras ventanas o desde Windows, y este es el sistema de comunicación principal y fundamental de Windows.

Todo mensaje se compone de 4 parámetros (actualmente todos ellos son enteros de 32 bits):

hWnd	Ventana destino
uMsg	Mensaje
wParam	Word Parameter (primer parámetro auxiliar)
lParam	Long Parameter (segundo parámetro auxiliar)

Por esta razón, todo programa capaz de recibir notificaciones de Windows generalmente crea al menos una ventana (pudiendo esta, como decíamos, ser invisible).

Toda ventana se compone de una gran cantidad de parámetros que la definen, entre los cuales se encuentran la identificación de la ventana padre (volveremos sobre esto mas adelante) y la función que controlará los mensajes de las ventanas. Debe hacerse una aclaración en este punto. En Windows, una ventana es una entidad abstracta definida como tal, representable en pantalla mediante un rectángulo, capaz de procesar mensajes. En la práctica, casi todos los elementos constitutivos de una ventana (botones, casilleros de edición, listados, iconos, bitmaps, animaciones, etc) son a su vez ventanas, e incluso hay ventanas que no se dibujan en pantalla. Por esta razón podría ampliarse la definición de ventana diciendo que es, también, una entidad abstracta que implementa un "objeto" y todo el código relacionado con él. Notar que la definición de ventana difiere de la que habitualmente utiliza un usuario.

La razón de esto es muy simple. Sería una tarea sumamente compleja definir en cada ventana el código necesario para procesar los mensajes provenientes de cada uno de los elementos y botones que esta tenga. Además, todos los elementos del mismo tipo, (botones, listados, etc), se comportarán de la misma forma. Por esta razón es muy simple trabajar estos elementos como nuevas ventanas, que tendrán el código necesario para dibujarse e interactuar con el ambiente Windows a través de mensajes. La tarea del programador se limita en tal caso a agregar el código necesario para integrar este elemento con el resto de la aplicación, y de querer hacerlo, modificar el funcionamiento por defecto de este objeto o elemento.

Llega entonces el momento de definir correctamente una ventana:



Una ventana es una entidad abstracta, compuesta por varios parámetros, cuyas características fundamentales son capacidad para recibir y procesar mensajes y desplegarse en pantalla.

Finalmente, debe agregarse que, a diferencia de un programa en DOS, un programa para Windows se compone, además del código ejecutable, de recursos (*resources*), que pueden ser cosas muy diversas (ventanas de dialogo, bitmaps, iconos, tablas de textos, cursores, animaciones, tabla de aceleradores, versión del programa y cualquier tipo de dato que al programador le pueda resultar útil incluir) pero los fundamentales son las ventanas de dialogo. Estos recursos deben incluirse con el código fuente en el momento de compilar el programa, en un archivo de texto, llamado *resource script*, cuya extensión es .rc.



Todo programa Windows se compone, además del código fuente, de recursos (información de cualquier tipo), definida en un archivo denominado 'resource script', que debe ser compilado junto con el programa.

Sólo puede existir un único archivo de recursos en un programa, pero este puede incluir a su vez referencias de otro. Ver documentación del entorno para más información.



Antes de continuar con definiciones, veamos un ejemplo de un programa Windows, para facilitar la familiarización con el entorno.

Primer Ejemplo de programa Windows

Este programa se compone de tres archivos:

- **ejemplo1.c:** Código fuente principal del programa
- **ejemplo1.rc:** Recursos de la aplicación
- **resource.h:** Declaraciones de recursos

Los archivos ejemplo1.rc y ejemplo1.h se generan en forma visual, utilizando un editor de recursos, provisto junto con el compilador.

Al igual que al programar bajo DOS, al desarrollar un programa compuesto de varios archivos es necesario crear un proyecto. Tanto Borland C++ como Visual C++, permiten crear varios tipos de proyectos Windows. En este caso deberemos crear un proyecto vacío. Esta operación depende del compilador utilizado, y se hace como sigue:

Borland C++ 5.0	Debe seleccionarse la opción "Nuevo proyecto", y agregar los archivos ejemplo1.c y ejemplo1.rc al proyecto.
Visual C++ 4.0	Debe seleccionarse la opción "File" - "New" - "Project Workspace" - "Application". Utilizando el menú "Insert" - "Insert Files Into Project" deben insertarse los archivos ejemplo1.c y ejemplo1.rc.
Visual C++ 6.0	Debe seleccionarse la opción "File" - "New" - "Project" - "Win32 Application". Una vez creado el proyecto, presionando el botón derecho del mouse sobre carpeta "FileView-Source Files" y seleccionando la opción "Add files to project" deben insertarse los archivos ejemplo1.c y ejemplo1.rc.

Los archivos **.c**, **.cpp** y **.rc** deben agregarse al proyecto, los **.h** no. Ciertos entornos de programación, como Visual C++ 6.0 permiten incluir los **.h** en una sección especial del proyecto, *Header files*, si bien esto sólo se por claridad, pero no se tiene en cuenta al compilar.

Archivo EJEMPLO1.C

```
01  #include <windows.h>
02  #include "resource.h"
03
04  /* Instancia del programa */
05  HINSTANCE inst;
```

```
06
07     LONG CALLBACK FuncionPrincipal (HWND hWnd, UINT wParam, WPARAM wParam,
LPARAM lParam)
08     {
09         char Texto [1000];
10         int Ret;
11
12         switch (wParam)
13         {
14             case WM_INITDIALOG:
15                 /* Este mensaje se envía a la ventana en el momento de
crearla,
16                 justo antes de mostrarla en pantalla */
17                 break;
18             case WM_COMMAND:
19                 /* Este mensaje se envía a la ventana cada vez que se presiona
un
20                 botón (entre otras cosas) */
21                 switch (LOWORD(wParam))
22                 {
23                     case IDC_SALIR:
24                         PostQuitMessage (0);
25                         break;
26                     case IDC_AGREGAR:
27                         SendMessage (GetDlgItem(hWnd, IDC_EDIT1), WM_GETTEXT,
sizeof (Texto), (LPARAM) Texto);
28                         SendDlgItemMessage (hWnd, IDC_LIST1, LB_ADDSTRING, 0,
(LPARAM) Texto);
29                         break;
30                 }
31                 break;
32             case WM_CLOSE:
33                 /* Este mensaje se envía cuando el usuario presiona el botón
[X]
34                 para cerrar la ventana. Se debe devolver 0 si no
35                 se permite cerrar, y 1 en caso afirmativo */
36                 Ret = MessageBox (hWnd, "¿Realmente desea salir?",
"Saliendo...",
37                 MB_YESNO|MB_ICONQUESTION);
38                 if (Ret != IDYES)
39                     return 0;
40                 PostQuitMessage (0);
41                 return 1;
42                 break;
43         }
44         /* Una ventana debe devolver 0 si no procesa el mensaje */
45         return 0;
46     }
47
48
49     int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE prev_instance, LPSTR
cmdline, int cmdshow)
50     {
51         MSG msg;
52         inst = hInstance;
53
54         /* Crear una ventana de dialogo (Ventana fundamental de la aplicación)
*/
```

```
55         CreateDialog (inst, MAKEINTRESOURCE (IDD_DIALOG1), NULL, (DLGPROC)
FuncionPrincipal);
56
57         while (GetMessage (&msg, NULL, 0, 0))
58         {
59             TranslateMessage (&msg);
60             DispatchMessage (&msg);
61         }
62
63         return msg.wParam;
64     }
```

Archivo RESOURCE.H

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Ej01.rc
//
#define IDD_DIALOG1 101
#define IDC_EDIT1 1000
#define IDC_AGREGAR 1001
#define IDC_LIST1 1002

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 102
#define _APS_NEXT_COMMAND_VALUE 40001
#define _APS_NEXT_CONTROL_VALUE 1003
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

Archivo EJEMPLO1.RC

Este archivo contiene los recursos de la aplicación, en esta caso, un cuadro de diálogo. Este archivo se genera en forma visual, utilizando un editor de recursos, provisto junto con el entorno de desarrollo o integrado al mismo.

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by ejemplo1.rc
//
#define IDD_DIALOG1 101
#define IDD_DIALOG2 102
#define IDC_EDIT1 1001
#define IDC_LIST1 1002
#define IDC_BUTTON1 1003
#define IDC_AGREGAR 1003
#define IDC_SALIR 1004
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 103
#define _APS_NEXT_COMMAND_VALUE 40001
#define _APS_NEXT_CONTROL_VALUE 1005
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

```
#endif

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
/////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////
// Spanish (Mexican) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ESM)
#ifdef _WIN32
LANGUAGE LANG_SPANISH, SUBLANG_SPANISH_MEXICAN
#pragma code_page(1252)
#endif // _WIN32
/////////////////////////////////////////////////////////////////
//
// Dialog
//
IDD_DIALOG1 DIALOGEX 0, 0, 263, 95
STYLE DS_MODALFRAME | DS_CENTER | WS_MINIMIZEBOX | WS_MAXIMIZEBOX | WS_POPUP |
WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Ejemplo de un programa bajo Windows"
FONT 8, "MS Sans Serif"
BEGIN
DEFPUSHBUTTON "Salir",IDC_SALIR,7,74,50,14
EDITTEXT IDC_EDIT1,7,7,110,14,ES_AUTOHSCROLL
LISTBOX IDC_LIST1,133,7,123,82,LBS_SORT | LBS_NOINTEGRALHEIGHT |
WS_VSCROLL | WS_TABSTOP,WS_EX_RIGHT
PUSHBUTTON "-- Agregar ->",IDC_AGREGAR,7,33,60,14
END
IDD_DIALOG2 DIALOG DISCARDABLE 0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Dialog"
FONT 8, "MS Sans Serif"
BEGIN
DEFPUSHBUTTON "OK",IDOK,129,7,50,14
PUSHBUTTON "Cancel",IDCANCEL,129,24,50,14
PUSHBUTTON "Button1",IDC_BUTTON1,17,40,50,14
EDITTEXT IDC_EDIT1,73,45,40,14,ES_AUTOHSCROLL
END

/////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
IDD_DIALOG1, DIALOG
BEGIN
LEFTMARGIN, 7
RIGHTMARGIN, 256
TOPMARGIN, 7
BOTTOMMARGIN, 88
```

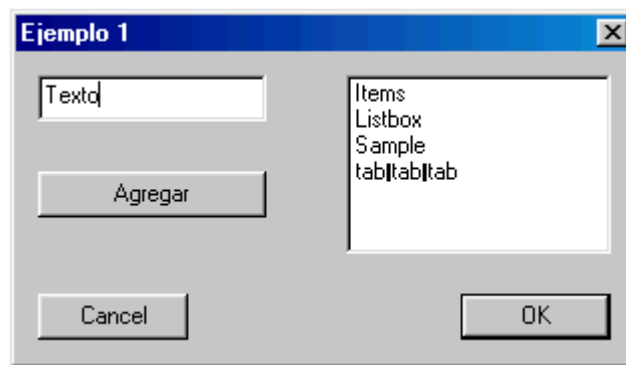
```
END
IDD_DIALOG2, DIALOG
BEGIN
LEFTMARGIN, 7
RIGHTMARGIN, 179
TOPMARGIN, 7
BOTTOMMARGIN, 88
END
END
#endif // APSTUDIO_INVOKED

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
"resource.h\0"
END
2 TEXTINCLUDE DISCARDABLE
BEGIN
"#include ""afxres.h""\r\n"
"\0"
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
"\r\n"
"\0"
END
#endif // APSTUDIO_INVOKED
#endif // Spanish (Mexican) resources
////////////////////////////////////

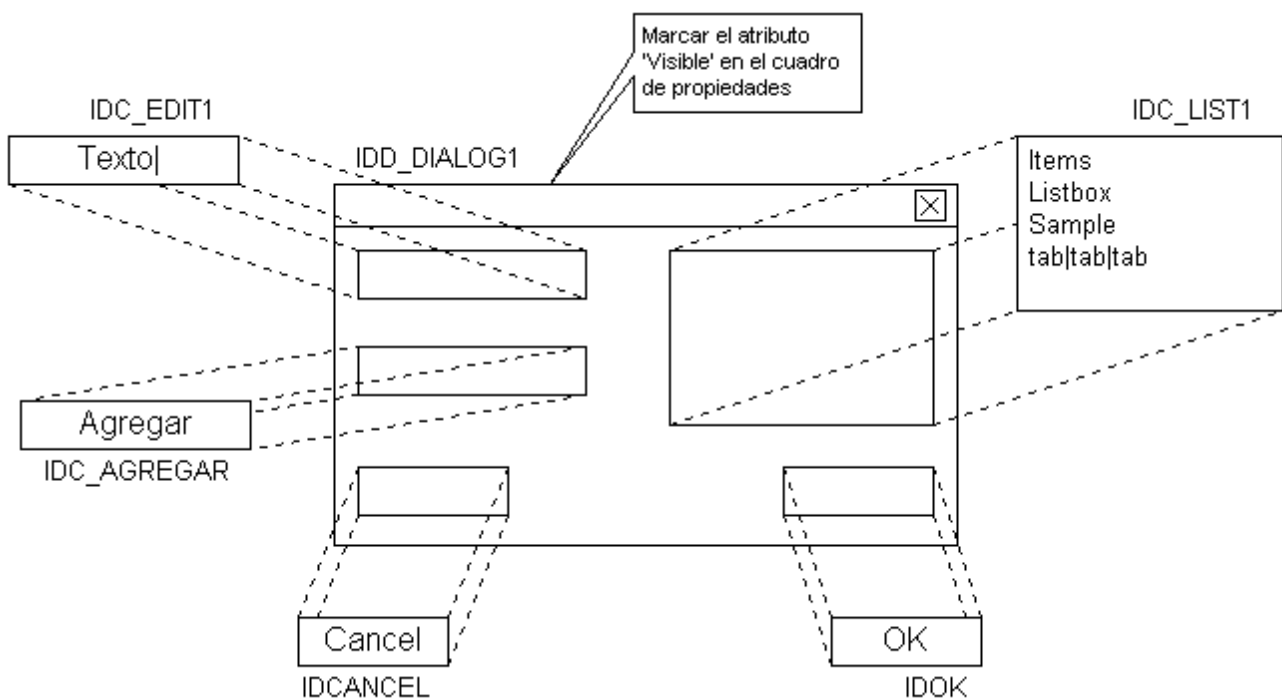
#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
////////////////////////////////////
#endif // not APSTUDIO_INVOKED
```

Analicemos el programa

En los archivos resource.h - ejemplo1.rc se crean utilizando un editor de recursos, que suele estar integrado con el entorno de desarrollo. En los mismos se definen los recursos del programa. Concretamente se define un diálogo como sigue:



Cada uno de los elementos de esta ventana es a su vez una ventana, y tiene asociado un ID, que permite identificarlos, tal como se muestra a continuación:



Estos ID son enteros de 32 bits, y deben ser especificados al crear los elementos de la ventana, en el entorno. La mayoría de los entornos o editores de recursos permiten hacer esto en un cuadro de propiedades del elemento, el cual suele abrirse presionando el botón derecho del mouse sobre el elemento.

Analicemos ahora el código C

En la línea 1 del archivo `ejemplo1.c` se incluye el archivo `windows.h`, un nuevo header donde se definen todas las funciones, constantes y estructuras fundamentales de Windows. En la línea 2 se incluye el archivo `resource.h`. Este archivo, generado automáticamente al generar `ejemplo1.rc`, contiene las identificaciones de todos los recursos almacenados en este archivo.

La ejecución de un programa Windows comienza su ejecución en la función `WinMain()`. Esta función recibe 4 parámetros, y devuelve uno sólo. Línea 49.

hInstance: Contiene el número de proceso del programa. Todo proceso (programa) bajo Windows se identifica unívocamente por un número (entero de 32 bits). Si un programa es ejecutado varias veces, cada "instancia" tendrá un número distinto. Este número es muy usado en todo el programa, razón por la cual se lo suele almacenar como una variable global (ver línea 52).

prev_instance: Definida por compatibilidad con Windows 16 bits. No tiene un valor útil en Windows 32bits (95/98/2000/NT).

cmdline: Línea de comandos con que se ejecutó el programa. Idem DOS.

cmdshow: Forma en que debe arrancar la aplicación. Este parámetro indica si la aplicación debe comenzar minimizada, maximizada, etc. Puede despreciárselo.

En la línea 55 se crea la primer ventana del programa. En esta línea se solicita la creación de una ventana de diálogo, utilizando la función `CreateDialog()`, que recibe 4 parámetros:

1. El parámetro `hInstance`, ya visto.
2. La ventana de diálogo a crear. La macro `MAKEINTRESOURCE()` se utiliza por compatibilidad. Antiguamente se identificaba a cada recurso con un string (`char *`), y hoy se lo hace con un entero. Ya que por compatibilidad no se puede cambiar la definición de la función, se utiliza esta macro que hace la conversión.
3. El padre de la ventana, en este caso `NULL`. Esto puede entenderse de dos formas, igualmente válidas:
 1. La ventana no tiene padre.
 2. (Lo correcto) El padre de la ventana es el escritorio de Windows.
1. La función que procesará los mensajes.

El ciclo implementado entre las líneas 57 y 61 lo veremos más adelante. Básicamente es idéntico en todos los programas, y simplemente se encarga de recibir mensajes y enviárselos a la función correspondiente.

Notar las palabras reservadas *CALLBACK* y *WINAPI* en las líneas 7 y 49. Estas palabras simplemente modifican la forma en que el compilador implementa la llamada a la función, ya que las mismas no serán llamadas directamente por nuestro programa sino por Windows, y deben ser codificadas de una forma especial.

En la línea 7 se define la función *callback* que procesará los mensajes de la ventana de dialogo creada, cuyo código se implementa entre las líneas 8 y 46. Simplemente se define el código para procesar 3 mensajes:

- **WM_INITDIALOG:** Este mensaje es enviado al crearse la ventana, justo antes de que se la muestre en pantalla. En este punto deberían inicializarse todos los datos de la ventana, si los hubiese. Por ejemplo, limitar el tamaño de los textos ingresables en los recuadros, cargar listados, etc.
- **WM_CLOSE:** Este mensaje se transmite a la ventana cuando se presiona el botón [x] que se encuentra a la derecha, en la barra de título de la ventana. La función *MessageBox()* crea y muestra una ventana muy sencilla, con el texto, título y botones especificados. Su funcionamiento es muy simple. En la ayuda de la función se describen todos los parámetros posibles.
- La función *PostQuitMessage()* finaliza el ciclo definido en las líneas 57 a 61, y con ello la ejecución del programa.
- **WM_COMMAND:** Agrupa una gran cantidad de eventos proveniente de ventanas hijas, particularmente todas las presiones de botones. El número de botón presionado se especifica en los 16 bits menos significativos de *wParam*. Por ello se define en la línea 21: `switch (LOWORD(wParam))` (que es equivalente a: `switch (wParam&0xFFFF)`) y que procesa los mensajes-comando emitidos por el botón Agregar (`IDC_AGREGAR`) y el botón Salir (`IDC_SALIR`).

Como vimos arriba, con el editor de recursos se asigna un ID o identificador a cada uno de los elementos constitutivos de la ventana. A su vez, al abrirse el diálogo en pantalla, Windows asignará un *handle* (que también es un entero de 32 bits).

F Una ventana tendrá asociados dos números enteros (de 32 bits). Un *handle*, asignado por Windows en tiempo de ejecución al crear la ventana en pantalla, y un ID, asignado por el programador en la etapa de desarrollo.

¿Qué sentido tiene que una ventana esté identificada por dos números enteros? La respuesta es la siguiente: Windows, identifica cada ventana creada por su *handle*. Este permite identificar cada una de las ventanas del sistema, y son creados en tiempo de ejecución. De hecho, si se ejecutasen dos instancias de la aplicación, los mismos elementos en ambas instancias tendrán *handles* diferentes.

Entonces ¿cómo puede el programador identificar un determinado elemento de la ventana de diálogo, por ejemplo, para enviarle un mensaje? La respuesta es, asignándole un ID o identificador, al momento de definirlo. Windows provee una función *GetDlgItem()*, que permite obtener el *handle* de un elemento, conocidos su ID y el *handle* de la ventana padre.

De esta forma puede realizarse una llamada como:

```
HWND hWndList = GetDlgItem (hWnd, IDC_EDIT1);
```

para obtener el handle de la ventana de edición de texto, y luego llamar a SendMessage() para enviarle un mensaje. Esta llamada podría escribirse también como sigue:

```
SendMessage(GetDlgItem(ventana,id),,,,,,....)
```

Ya que esta llamada es muy común, Windows define la función:

```
SendDlgItemMessage(ventana, id,,,,,....)
```

que hace esto mismo.

En la línea 27 se lee el texto de la ventana de edición de texto y en la línea 28 se inserta este texto en el listado de texto. Los parámetros concretos que recibe cada mensaje no hace falta memorizarlos. Windows define más de 5000 mensajes cantidad que aumenta día a día, cada uno de los cuales tiene sus propios parámetros. Pero afortunadamente no hace falta conocer todos los mensajes existentes. Simplemente basta saber que existen mensajes para prácticamente todo evento que se nos pueda ocurrir, pasando por movimientos de mouse, recepción de mensajes de red o modem, apagado de Windows, y sus nombres son bastante representativos, por lo que no es una tarea muy difícil identificar el mensaje requerido. Todo compilador para Windows trae un índice de los mensajes de Windows existentes.

Entre los tipos de mensajes más utilizados se encuentran:

LB_...	Mensajes enviables a un ListBox
LBN_...	Notificación proveniente de un ListBox
CB_...	Mensajes enviables a un ComboBox
CBN_...	Notificación proveniente de un ComboBox
EM_...	Mensajes enviables a un EditText
EN_...	Notificación proveniente de un EditText
STM_...	Mensajes enviables a una ventana Static
STN_...	Notificación proveniente de una ventana Static
SBM_...	Mensajes enviables a una ventana Static
SBN_...	Notificación proveniente de una ventana Static
WM_...	Mensajes genéricos de una ventana
WM_NOTIFY	Notificaciones varias provenientes de una ventana

Hay tres categorías principales de mensajes:

- **Mensajes de Windows:** incluye aquellos mensajes que comienzan con el prefijo WM_ excepto WM_COMMAND; estos mensajes tienen parámetros que son usados para identificar el mensaje y son manejados por Windows y las vistas.

- **Notificaciones de control:** incluye los mensajes de notificación WM_COMMAND desde controles y otras ventanas hijas a sus ventanas padres, por ejemplo: una ventana edit envía a su ventana padre un código de notificación de control cuando el usuario ha tomado una acción que puede alterar el texto en el control de edición; el identificador de ventana para el mensaje responde al mensaje de notificación de alguna manera apropiada, por ejemplo, retirando el texto en el control.
- **Mensajes de comandos:** incluye los mensajes de notificación WM_COMMAND desde los objetos de la interfaz de usuario: menús, botones de la barra de herramientas y teclas aceleradoras. El entorno de trabajo (framework) procesa estos comandos en forma diferente a otros mensajes ya que pueden ser manejados por una amplia variedad de objetos: documentos, vistas, la aplicación misma y, por supuesto, el sistema operativo.

Entre los mensajes WM_... más comunes se encuentran:

WM_INITDIALOG:

Enviado cuando se crea una ventana de dialogo, antes de mostrarla en pantalla.

WM_CREATE:

Enviado cuando se crea una ventana común, antes de mostrarla en pantalla.

WM_LBUTTONDOWN, WM_RBUTTONDOWN, WM_MBUTTONDOWN

WM_LBUTTONUP, WM_RBUTTONUP, WM_MBUTTONUP

WM_RBUTTONDOWNDBLCLK, WM_RBUTTONDOWNDBLCLK, WM_MBUTTONDOWNDBLCLK

Presión de los botones del mouse (izquierdo, derecho y medio).

WM_CLOSE:

Se presiona el botón de cerrar la ventana.

WM_PAINT:

Enviado cuando la ventana debe ser redibujada.

WM_COMMAND:

Mensajes provenientes de ventanas hijas, fundamentalmente presiones de botones o selección de comandos del menú. En caso de presiones de un botón, En los 16 bits menos significativos de wParam se especifica el ID del botón presionado.

¿Quién procesa los mensajes restantes?

Claramente la aplicación recibirá muchos más mensajes que no son procesados en este caso. Al crear ventanas de diálogo Windows se encargará de llamar (internamente) a una función de procesamiento genérico de mensajes, que proporciona el código necesario para procesar todos los mensajes para los cuales la función callback de la aplicación devuelva 0.

Múltiples ventanas

¿Qué pasaría si duplicásemos la línea 55? En este caso el código de esta línea quedaría:

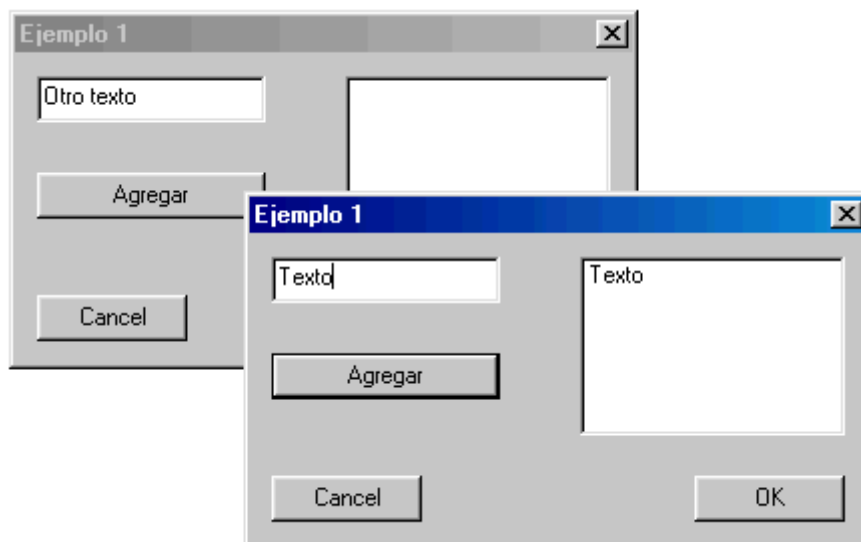
```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE prev_instance, LPSTR cmdline,
int cmdshow)
{
    MSG msg;
    inst = hInstance;

    /* Crear dos ventanas de dialogo */
    CreateDialog (inst, MAKEINTRESOURCE (IDD_DIALOG1), NULL, (DLGPROC)
FuncionPrincipal);
    CreateDialog (inst, MAKEINTRESOURCE (IDD_DIALOG1), NULL, (DLGPROC)
FuncionPrincipal);

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }

    return msg.wParam;
}
```

Pues bien, en este caso se crearían dos ventanas de diálogo, y el usuario podría trabajar indistintamente con cualquiera de ellas, tal como se muestra abajo:



En este caso ambas ventanas compartirán la misma función de procesamiento de mensajes, *FuncionPrincipal()*. El parámetro *hWnd* identificará a la ventana a la cual se esté enviando el mensaje. Como ya se vio, ambas ventanas tendrán los mismos elementos, pero los handles *HWND* tanto de las ventanas principales como de cada uno de sus elementos serán diferentes.

Ventanas modales

Las líneas 55 a 63 de nuestro primer ejemplo podrían haber sido reemplazadas en realidad por:

```
return DialogBox (inst, MAKEINTRESOURCE (IDD_DIALOG1),  
                NULL, (DLGPROC)FuncionPrincipal);
```

Notar que al hacer esto se elimina el ciclo de mensajes del programa. ¿Quiere esto no se utiliza un ciclo de mensajes? De ninguna manera. Simplemente, la función DialogBox ya implementa internamente un ciclo de procesamiento de mensajes como vimos arriba, sólo que el mismo está implementado de una forma particular, que imposibilita al usuario seleccionar la ventana padre. Esto es, si dentro de una ventana se abre una otra ventana utilizando DialogBox, la ventana previa no podrá ser seleccionada hasta que no se cierre la nueva ventana. A esto se conoce como ventana **modal**.

En nuestro primer ejemplo, ya que sólo se abre una ventana, es indiferente trabajarla en forma modal y no modal.

Un poco más sobre ventanas

Habiendo visto el funcionamiento general de un programa, podemos ahora analizar el funcionamiento de las ventanas con más detalle.

Las funciones de manejo de las ventanas proveen a las aplicaciones el medio de crear e implementar la interfaz de usuario. Se usan funciones de manejo de ventanas para crear y usar ventanas para desplegar salidas, proveer la manera en que el usuario ingresará a las entradas y efectuar las otras tareas necesarias para mantener la interacción con el usuario.

Las aplicaciones definen el comportamiento y apariencia generales de sus ventanas creando modelos de ventanas y los procedimientos de ventanas correspondientes. Los modelos, tipos o "clases" de ventanas (*window classes*) identifican las características por defecto, tales como, por ejemplo, la forma en que la ventana procesara un doble click del botón del mouse o si tiene o no un menú para selección de items. Los procedimientos de ventanas (*windows procedures*) contienen el código que define el comportamiento de las ventanas (esto es: cómo llevan a cabo las tareas requeridas) y procesan las entradas de usuario; estos procedimientos son funciones que se ocupan de todos los mensajes enviados y recibidos de y a todas las ventanas derivadas de ese modelo; todos los aspectos de la apariencia y comportamiento de esta ventanas dependen de las respuestas del procedimiento a los mensajes; por ejemplo, para computadoras con interfaz de usuario gráfica, los iconos, menús y cajas de dialogo deben sus características a estas funciones.

Debido a que todas las ventanas comparten el área de pantalla, las aplicaciones no reciben acceso a la pantalla entera. En lugar de eso, el sistema maneja todas las salidas, de forma tal que queden alineadas y, de ser necesario, recortadas para calzar dentro de la ventana correspondiente.

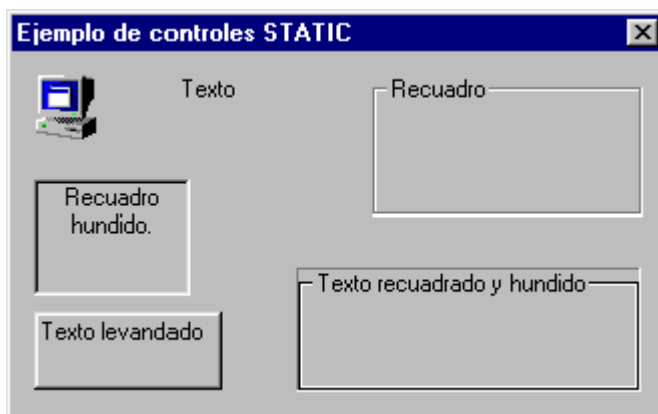
Las aplicaciones pueden dibujar ventanas en respuesta a requerimientos del sistema o mientras están procesando mensajes de entrada. Cuando el tamaño o porción de una ventana cambia el sistema típicamente envía un mensaje a la aplicación requiriendo que esta dibuje cualquier parte de sus ventanas no expuesta previamente.

Windows informa a las aplicaciones de eventos en el sistema, tales como movimientos del mouse, presión sobre los botones del mouse, o presión sobre las teclas en mensajes de entrada utilizando mensajes, los cuales almacena en una cola para cada aplicación, proveyendo automáticamente estas colas. Las aplicaciones usan las funciones de mensajes para extraer mensajes de la cola y despacharlos a los procedimientos de ventanas apropiados para su procesamiento.

Ya que toda aplicación con interfaz de usuario se compondrá sin duda de una serie de elementos básicos, tales como botones, cuadros de ingreso de texto, etc, cuya programación (sumamente compleja) será idéntica en todos los casos, Windows ya ofrece una serie de ventanas básicas, con su correspondiente código necesario para procesar una serie muy grande de mensajes. Estas ventanas están agrupadas en los siguientes tipos:

Ventanas **STATIC**:

textos, iconos, bitmaps, recuadros



Ventanas **BUTTON**:

Botones, en cualquiera de sus tipos: simple, check box, radio button



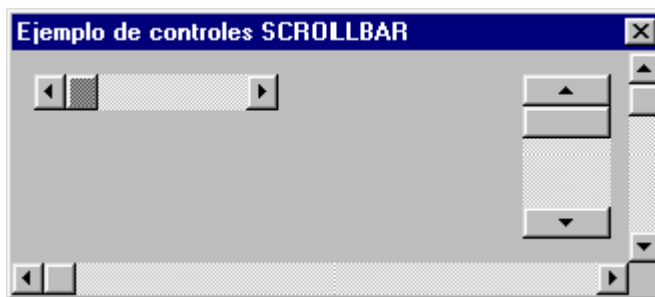
Ventanas **EDIT**:

Recuadros para la edición de texto



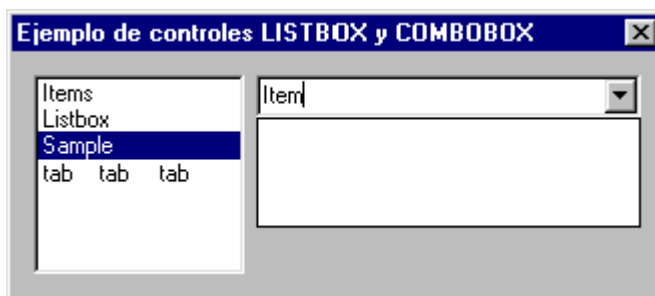
Ventanas SCROLLBAR:

Barras de desplazamiento



Ventanas LISTBOX y COMBOBOX:

Ventanas para la selección de elementos



Existe un segundo grupo de ventanas básicas, implementadas a partir de Windows 32 bits, que ayudan a darle a Windows la apariencia y comportamiento que lo caracterizan, conocidos como **Common Controls** y **Common Dialogs**. Estos son controles y ventanas de propósito general que representan listados con imágenes, árboles, animaciones, barras de progreso, y otros. Estos controles están agrupados dentro de librerías .DLL (dynamic link library o librería de enlace dinámico) como COMCTL32.DLL, las cuales, por ser parte del sistema operativo, están disponibles para todas las aplicaciones. Los controles son ventanas hijas que las aplicaciones usan juntamente con otras ventanas para realizar operaciones de entrada/salida.

Estas librerías se han ido extendiendo desde su aparición original en Windows 95/NT, agregando nuevos elementos y nuevas capacidades a las ya existentes, y son actualizadas en Windows por algunos programas, tales como Internet

Explorer. De allí que varios programas requieran para ejecutarse que se haya instalado previamente una determinada versión (o superior) de Internet Explorer.

Dibujando en pantalla

Ahora crearemos otro ejemplo con un tema interesante y muy aclarativo, el dibujar en pantalla. Nuestro programa dibujará una línea en pantalla al presionarse el botón [OK].

El código de nuestro programa será ahora el siguiente:

Ejemplo2.c

```
#include <windows.h>
#include "resource.h"

/* Instancia del programa */
HINSTANCE inst;

LONG CALLBACK FuncionPrincipal (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam)
{
    switch (wMsg)
    {
        case WM_INITDIALOG:
            /* Este mensaje se envía a la ventana en el momento de crearla,
             justo antes de mostrarla en pantalla */
            break;
        case WM_COMMAND:
            /* Este mensaje se envía a la ventana cada vez que se presiona un
             botón (entre otras cosas) */
            switch (LOWORD(wParam))
            {
                case IDOK:
                    {
                        HDC hdc;
                        hdc = GetDC (hWnd);
                        MoveToEx (hdc, 0, 0, NULL);
                        LineTo (hdc, 2000, 1000);
                        ReleaseDC (hWnd, hdc);
                    }
                    break;
            }
            break;
        case WM_CLOSE:
            /* Este mensaje se envía cuando el usuario presiona el botón [X]
             para cerrar la ventana. */
            PostQuitMessage (0);
    }
    /* Una ventana debe devolver 0 si no procesa el mensaje */
    return 0;
}
```

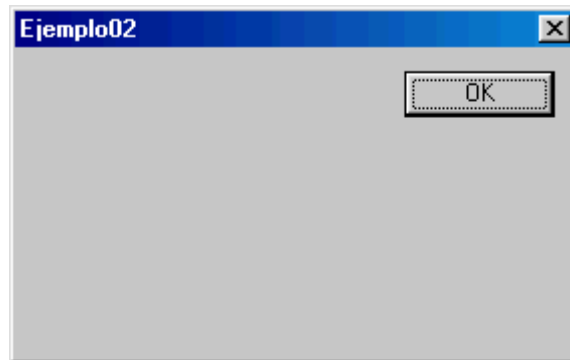
```
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE prev_instance, LPSTR cmdline,
int cmdshow)
{
    inst = hInstance;

    /* Crear una ventana de dialogo (Ventana fundamental de la aplicación) */
    DialogBox (inst, MAKEINTRESOURCE (IDD_DIALOG1), NULL, (DLGPROC)
FuncionPrincipal);

    return 0;
}
```

Ejemplo2.rc

Al igual que en el ejemplo anterior, tendremos un archivo de recursos, *ejemplo2.rc*, con su correspondiente *resource.h*. En estos archivos se define un diálogo como se muestra a continuación:



El código es el siguiente:

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// Spanish (Castilian) (unknown sub-lang: 0xB) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ESS)
#ifdef _WIN32
LANGUAGE LANG_SPANISH, 0xB
#pragma code_page(1252)
#endif // _WIN32

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
```

```
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

////////////////////////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 186, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "Ejemplo02"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON    "OK",IDOK,129,7,50,14
END

////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 179
        TOPMARGIN, 7
        BOTTOMMARGIN, 88
    END
END
#endif    // APSTUDIO_INVOKED

#endif    // Spanish (Castilian) (unknown sub-lang: 0xB) resources
////////////////////////////////////
```

```
#ifndef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif // not APSTUDIO_INVOKED
```

resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Ejemplo02.rc
//
#define IDD_DIALOG1 101

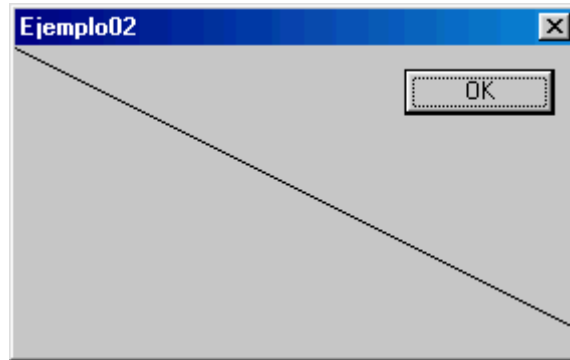
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifdef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE 102
#define _APS_NEXT_COMMAND_VALUE 40001
#define _APS_NEXT_CONTROL_VALUE 1000
#define _APS_NEXT_SYMED_VALUE 101
#endif
#endif
```

El archivo de recursos se generó visualmente, como se vió en el ejemplo anterior, por lo cual no será analizado nuevamente. No analizaremos todo el ejemplo nuevamente, sino que nos centraremos en las diferencias con el ejemplo anterior.

Como puede verse, al presionarse el botón [OK] se llamará al código:

```
case IDOK:
{
    HDC hdc;
    hdc = GetDC (hWnd);
    MoveToEx (hdc, 0, 0, NULL);
    LineTo (hdc, 2000, 1000);
    ReleaseDC (hWnd, hdc);
}
break;
```

Si el usuario prueba presionar el botón verá que se dibuja una línea en la ventana tal como se muestra a continuación:



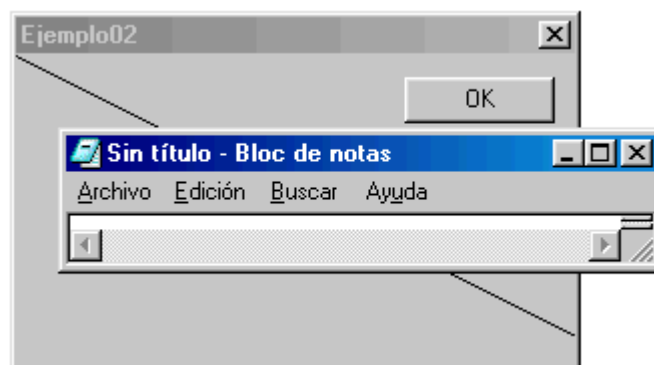
En este código se utiliza un nuevo *handle*, denominado **HDC** o *Handle to Device Context*. El mismo es un **identificador o handle de un contexto de un dispositivo**. Esta explicación resulta sin duda bastante confusa, así que trataremos de explicarla más claramente.

Windows controla todas las funciones de acceso a pantalla. Esto tiene, entre otros, dos propósitos básicos:

1) El área de dibujo está limitada a la ventana. Es decir que, el sistema de coordenadas tiene su origen en la posición superior izquierda de la ventana, y no puede salirse de él. Si bien los parámetros recibidos son enteros de 32 bits, Windows posee una limitación interna de tal forma que los números recibidos no deben superar los 16 bits de precisión.

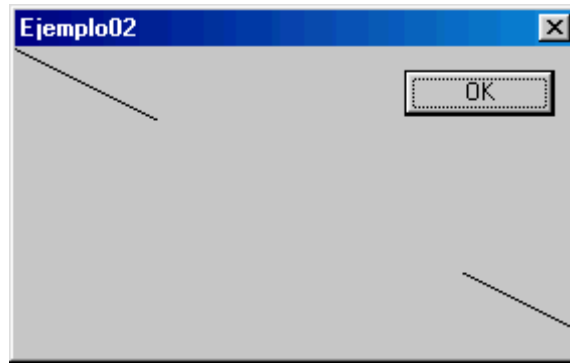
2) Windows controla el área del dispositivo (en este caso la ventana) que está visible en pantalla y debe ser dibujada. Por ejemplo, si la ventana estuviese tapada por otra y se dibujase algo, este dibujo no se verá en pantalla y no tendrá ningún efecto.

Como puede verse en la siguiente imagen, la línea no se dibuja sobre la ventana superior.



Redibujando la ventana

¿Qué pasaría si luego de presionar el botón [OK], la ventana queda oculta (total o parcialmente) y luego vuelve a ser visible en su totalidad?



La respuesta es que Windows no redibuja el área que quedó oculta, sino que esta es tarea de la aplicación. Cuando un área de la ventana vuelve a ser visible, Windows informa a la aplicación que un área de la ventana debe ser repintada, utilizando el mensaje WM_PAINT. Es posible para la aplicación saber qué área debe ser redibujada y actuar en consecuencia. Una solución sencilla es repintar toda la ventana.

Modifiquemos entonces la función *FuncionPrincipal()* para que el programa funcione correctamente:

```
LONG CALLBACK FuncionPrincipal (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM
lParam)
{
    static BOOL bDibujarLinea;

    switch (wMsg)
    {
        case WM_INITDIALOG:
            /* Este mensaje se envía a la ventana en el momento de crearla,
               justo antes de mostrarla en pantalla */
            bDibujarLinea = FALSE;
            break;
        case WM_PAINT:
            /* Windows enviará a la ventana un mensaje WM_PAINT cuando un área
               de la misma deba ser redibujada */
            if (bDibujarLinea)
            {
                HDC hdc;
                hdc = GetDC (hWnd);
                MoveToEx (hdc, 0, 0, NULL);
                LineTo (hdc, 2000, 1000);
                ReleaseDC (hWnd, hdc);
            }
            break;
        case WM_COMMAND:
            /* Este mensaje se envía a la ventana cada vez que se presiona un
               botón (entre otras cosas) */
            switch (LOWORD(wParam))
            {
                case IDOK:
                    bDibujarLinea = TRUE;
                    InvalidateRect (hWnd, NULL, FALSE);
                    break;
            }
            break;
        case WM_CLOSE:
            /* Este mensaje se envía cuando el usuario presiona el botón [X]
               para cerrar la ventana. */
```

```
        PostQuitMessage (0);
    }
    /* Una ventana debe devolver 0 si no procesa el mensaje */
    return 0;
}
```

Esta es la forma correcta de programar el dibujo en pantalla. **Todo el código de dibujo en pantalla debe codificarse como una respuesta al mensaje WM_PAINT, de Windows.** Esta es la única forma de garantizar que la ventana será redibujada si la misma queda oculta y luego vuelve a ser visible.

Al presionarse el botón [OK], simplemente "invalidamos la ventana", esto es, le informamos a Windows que la ventana tiene un estado inválido y que debe ser repintada, llamando a *InvalidateRect()*, tras lo cual Windows enviará un mensaje WM_PAINT, solicitando que la pantalla sea redibujada.

El primer parámetro de *InvalidateRect()* es el *handle* de la ventana que debe ser redibujada, el segundo es un puntero a un rectángulo (estructura *RECT*) que define el área que debe ser redibujada. En este caso se utiliza NULL, que identifica la ventana entera. Antes de ver el significado del tercer parámetro, veamos la siguiente pregunta:

Cuando la ventana queda oculta, y luego vuelve a ser visible, ¿quién repinta en fondo gris de la misma?. Algún programador inexperto podría pensar que esta es tarea de Windows, pero no es así. Cuando un área de la pantalla vuelve a ser visible (luego de haber quedado oculta), Windows envía a la aplicación un mensaje WM_ERASEBKGD. Al crear un cuadro de diálogo, cuando la función callback del mismo retorna 0 automáticamente se llama a una función genérica que procesa el mensaje. En este caso, si la aplicación no procesa el mensaje WM_ERASEBKGD lo hará esta función genérica.

Ahora sí, podemos ver qué es el tercer parámetro de la función *InvalidateRect()*. La documentación de esta función especifica que este tercer parámetro indica si la ventana debe ser borrada antes de ser redibujada o no. Desde el punto de vista operativo, lo que se especifica con este parámetro es si debe enviarse un mensaje WM_ERASEBKGD para que se repinte todo el fondo (en este caso de gris), antes de enviarse un mensaje WM_PAINT para que se redibuje la línea en la pantalla (en este caso una línea).

Capítulo III - Creación de una ventana

En el ejemplo del capítulo anterior utilizamos un tipo muy particular de ventana, que es la ventana de diálogo, junto con botones, un cuadro de edición y uno de listado, que son ventanas provistas por Windows. Sin embargo, pueden definirse nuevas ventanas, o incluso modificar las ya existentes. Para ello es necesario analizar la forma de crear una ventana.

La creación de una ventana genérica se realiza en dos pasos:

1º - Debe registrarse en Windows un modelo básico de ventana.

2º - Puede crearse una o más ventanas como variantes de algún modelo ya registrado.

Una ventana se compone de una gran cantidad de atributos, los cuales se especifican en estos dos puntos. Entre los parámetros que se definen en la primera etapa se encuentra la función necesaria para procesar los mensajes. En el segundo punto se especifican fundamentalmente atributos variables referentes al dibujo de la ventana en pantalla, tales como posición y tamaño.

Veamos ahora como registrar y mostrar un modelo de ventana propio. En un programa "tradicional" para Windows, esto es, sin usar librerías prehechas, nosotros debemos procesar todos los mensajes en el procesador de mensajes asociado con una ventana mediante la "registración de una clase ventana". Como mencionamos en el capítulo anterior, esto se realiza en dos pasos. En primer lugar hay que registrar un modelo de ventana básico, y luego se crea la ventana, a partir de un modelo básico ya registrado.

Paso 1: Registrar un modelo de ventana básico

Antes de crear una ventana en pantalla es necesario registrar un modelo básico, con las características fundamentales de la misma. Todas las ventanas creadas a partir de este modelo (denominado clase de ventana, no confundir con clase de C++) tendrán todas las características básicas, si bien pueden ser modificadas posteriormente. Uno de estos parámetros es la función callback, encargada de procesar los mensajes enviados a la ventana.

Para realizar esto es necesario cargar una estructura, *WNDCLASSEX*, con estos datos, y luego llamar a la función *RegisterClass* o su versión extendida *RegisterClassEx*. Esta función está definida como sigue:

```
ATOM RegisterClassEx (CONST WNDCLASSEX * lpwcx);
```

El valor retornado, *ATOM*, es un entero que identifica la clase creada. Generalmente su utilidad se reduce a verificar que la operación haya sido exitosa. Esta función reporta error retornando 0.

lpwcx es un puntero a la siguiente estructura:

```
typedef struct _WNDCLASSEX {  
    UINT cbSize;           // Debe tener el valor sizeof(WINDOWCLASSEX)
```

```
    UINT style;           // Estilo. Ver manual para lista de los tipos
posibles
    WNDPROC lpfnWndProc;  // Función callback que procesará los mensajes
    int cbClsExtra;       // Extra bytes a reservar a continuación de esta
estructura
    int cbWndExtra;       // Extra bytes a reservar a continuación de la
instancia de la ventana
    HANDLE hInstance;     // Instancia de la aplicación
    HICON hIcon;          // Icono de la ventana
    HCURSOR hCursor;      // Cursor de la ventana (por defecto)
    HBRUSH hbrBackground; // Fondo de la ventana
    LPCTSTR lpzMenuName;  // Menú de la ventana
    LPCTSTR lpzClassName; // Nombre de la clase
    HICON hIconSm;        // Icono pequeño (No utilizado en Windows NT)
} WNDCLASSEX;
```

Paso 2: Crear la ventana

La creación de la ventana se realiza llamando a la función `CreateWindow` o su versión extendida `CreateWindowEx`.

```
HWND CreateWindowEx(
    DWORD dwExStyle,       // Estilo extendido de la ventana
    LPCTSTR lpClassName,   // Puntero a un string con el nombre de la clase
    LPCTSTR lpWindowName,  // Texto de la ventana, generalmente (aunque no
siempre) el título.
    DWORD dwStyle,         // Estilo de la ventana
    int x,                 // Posición horizontal de la ventana (en pixels)
    int y,                 // Posición vertical de la ventana (en pixels)
    int nWidth,            // Ancho de la ventana (en pixels)
    int nHeight,           // Alto de la ventana (en pixels)
    HWND hWndParent,       // Identificador del padre de la ventana
    HMENU hMenu,           // Identificador del menu, o identificador de la
ventana en relación al padre
    HINSTANCE hInstance,   // Instancia actual del programa
    LPVOID lpParam         // Puntero para pasar datos a la aplicación. Ver
manual de la función
);
```

Esta función devuelve el identificador de la ventana creada, o `NULL` en caso de error.

Veamos el ejemplo completo:

```
/*
***          A R C H I V O S   A   I N C L U I R          ***
\*****/
#include <windows.h>
/*
Nota: La mayor parte del código referente a la verificación de errores
ha sido removido por claridad.
*/
/*
***          D E F I N I C I O N E S          ***
\*****/
#define OK 0
#define ERR 1
HINSTANCE inst;
```

```

/*****
***      P R O C E D I M I E N T O   D E   L A V E N T A N A      ***
*****/
LONG CALLBACK MainProc (HWND hWnd, UINT wParam, LPARAM lParam)
{
    switch (wParam)
    {
        case WM_CREATE:
            /* Aquí deben inicializarse las variables de la ventana */
            break;
        case WM_CLOSE:
            /* Cerrar la ventana */
            PostQuitMessage (0);
            break;
        default:
            /* Procesar todos los restantes mensajes */
            return DefWindowProc (hWnd, wParam, lParam);
    }
    return 0;
}

/*****
***      C R E A C I O N   D E   L A   V E N T A N A      ***
*****/
int Init (void)
{
    WNDCLASSEX wc;
    HWND Win;
    wc.cbSize = sizeof (wc);           // Debe contener el tamaño de la estructura
    wc.style = CS_DBLCLKS|CS_GLOBALCLASS; // Estilo de la ventana
    wc.lpfnWndProc = MainProc;         // Funcion que procesará los mensajes
    wc.cbClsExtra = 0;                 // Bytes adicionales al final de esta
estructura
    wc.cbWndExtra = 0;                 // Bytes adicionales en los datos de la
ventana creada
    wc.hInstance = inst;
    wc.hIcon = NULL;                  // Icono de la ventana
    wc.hCursor = NULL;                // Cursor que se usará en la ventana.
    wc.hbrBackground = COLOR_WINDOWFRAME;
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "Clase Ventana"; // Nombre de este tipo de ventana
    wc.hIconSm = NULL;                // Icono pequeño de la ventana
    /* Registrar la ventana */
    if (RegisterClassEx (&wc) == 0)
        return ERR;
    /* Crear la ventana principal */
    Win = CreateWindowEx (
        0,                          // Estilos extendidos de la ventana
        "Clase Ventana",             // Nombre de la clase registrada
        "Ventana de ejemplo",        // Título de la ventana
        WS_CAPTION|WS_SIZEBOX|WS_SYSMENU|WS_VISIBLE, // Estilo de la ventana
        CW_USEDEFAULT,               // Posición x en pixels (usar posición por
defecto)
        CW_USEDEFAULT,               // Posición y en pixels (usar posición por
defecto)
        CW_USEDEFAULT,               // Ancho en pixels (usar valor por defecto)
        CW_USEDEFAULT,               // Alto en pixels (usar valor por defecto)
        NULL,                        // Padre de la ventana (escritorio de Windows)
        NULL,                        // Menú de la ventana
        inst,                        // Instancia del programa
        NULL);                       // Información adicional
    if (!Win)

```

```
        return ERR;

    return OK;
}

/*****
***                               W I N M A I N                               ***
*****/

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nShowCmd)
{
    MSG msg;
    inst = hInstance;
    Init ();
    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}

/*****
***                               F I N                               ***
*****/
```

El mensaje WM_PAINT es enviado por Windows cada vez que se deba redibujar la ventana (en su totalidad o una parte de ella). Para dibujar dentro de la ventana debe obtenerse el área de contexto de la ventana (device-context), mediante la función GetDC(). El identificador devuelto (dc) nos garantizará que nuestros dibujos se harán dentro de la ventana especificada, y se dibujarán en ventanas que puedan estar superpuestas a la nuestra. El origen de coordenadas (0,0) es el pixel superior izquierdo de la ventana, y la posición se mide en pixels. Cualquier posición especificada fuera del área de la ventana (tanto valores positivos como negativos) no serán dibujados. De esta forma no es incorrecto, por ejemplo, dibujar en zonas negativas de la ventana, simplemente no se dibujará nada. Esto puede ser muy útil, ya que el sistema de coordenadas puede ser modificado, con lo cual podría trabajarse con una imagen muy grande, que pueda ser desplazada, ampliada o achicada con sólo cambiar el sistema de coordenadas. Desafortunadamente, el sistema de dibujo de Windows trabaja únicamente con enteros (con signo) de 16 bits.

En nuestro código redibujamos una línea desde la posición (0,0) hasta la posición (1000,1000), y a continuación imprimimos un número '1'. Es posible modificar todos los estilos de dibujo, como por ejemplo, utilizar diferentes estilos de línea, color, grosor, fuente de texto, itálica, negrilla, tamaño de letra, justificación, etc.

¿Qué sucedería si hubiésemos hecho nuestro dibujo en cualquier otro lugar del programa en vez de cómo respuesta a un mensaje WM_PAINT? Efectivamente se hubiese dibujado en pantalla, pero si por alguna razón la imagen de la ventana era tapada (por ejemplo por abrir otro programa), al volver a nuestra ventana el dibujo habría desaparecido.

Windows enviará un mensaje WM_PAINT cada vez que se deba redibujar la ventana. Esto puede resultar muy ineficiente, si el dibujo requiere tiempo para efectuarse. Para solucionar esto (en parte), es posible modificar este código para conocer la parte de la ventana que requiere ser dibujada, o para que sólo se redibuje aquella parte que fue modificada, pero lo dejaremos de lado por ahora.

El ciclo de mensajes y threads

Hasta ahora, en nuestros ejemplos hemos definido un ciclo de mensajes, sin reparar mucho en qué hace o cómo trabaja. El mismo se transcribe acá por claridad:

```
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
```

Es común que los programadores no profundicen en la forma en que se implementa el sistema de mensajería de Windows, y el resultado de ello son programas defectuosos. Algunos conceptos erróneos muy frecuentes son los siguientes:

1) No existe ningún peligro en enviar un mensaje a una ventana utilizando SendMessage().

Rta: Si la ventana a la cual se envía el mensaje no procesa los mensajes, por ejemplo por estar "colgada" la aplicación, esta función no retornará y por ende nuestra aplicación también quedará colgada. Existe una función llamada SendMessageTimeout() que puede ser utilizada si no se sabe el estado de la ventana destino del mensaje.

2) Todos los mensajes son encolados

Rta: No todos los mensajes son encolados. Se explicará esto más adelante.

3) Todo mensaje enviado utilizando PostMessage() a una ventana válida siempre es encolado.

Rta: Ya que, para encolar un mensaje debe existir una cola de mensajes, la cual tiene un tamaño finito, si la misma se llena el mensaje puede no ser encolado, en cuyo caso PostMessage() retornará un código de error.

Antes de ver la forma en que opera el sistema de mensajería de Windows es necesario hacer unas aclaraciones:

- Una aplicación o proceso Windows se puede componer de varias hebras de ejecución o threads, cada una de las cuales se ejecuta en forma independiente y tiene una cola propia. Inicialmente un proceso es creado con una única thread, y este puede crear nuevas threads.
- En Windows es posible definir parámetros de seguridad, que habilitan o bloquean ciertas operaciones. En Windows 95/98/Me, el sistema de seguridad es muy restringido, pero en NT el mismo es sumamente complejo. Así, cada thread tomará parámetros de seguridad que determinarán el permiso para realizar operaciones, que generalmente serán iguales para todas las threads de un mismo proceso.

Habiendo hecho estas aclaraciones, volvamos al tema de mensajería.

Como ya se vió, al crear un proceso, Windows crea la thread inicial del mismo. Esta thread o hebra de ejecución es la que comienza a ejecutarse en WinMain(). Una vez creada una o más ventanas deben procesarse los mensajes destinados a las mismas. La función GetMessage() lee y retira el primer mensaje enviado a la thread. Notar que los mensajes enviados a todas las ventanas de la aplicación son encolados en realidad en la cola de la thread que creó la ventana.

La función GetMessage() toma 4 parámetros:

El primero es una estructura MSG donde se almacenará el mensaje leído y retirado de la cola.

El segundo identifica la ventana de la cual se desea leer el mensaje. Si se especifica NULL, se leerán todos los mensajes. El campo hWnd de la estructura MSG contendrá el handle de la a la cual se envía el mensaje. Es posible enviar un mensaje a una thread, en cuyo caso este campo tendrá el valor NULL.

El tercer y cuarto parámetro permiten definir un rango de mensajes a leer. Si se especifica 0, 0 se leerán todos los mensajes.

La función TranslateMessage() modifica el mensaje, de acuerdo a ciertas reglas y aceleradores. Por ejemplo, se traduce la tacla [TAB] en un cambio de la ventana seleccionada, y la presión de una tecla en un mensaje identificando al carácter ASCII correspondiente.

Finalmente, la función DispatchMessage() se encarga de llamar a la función callback correspondiente.

Cabe destacar que no todos los mensajes son encolados. Por ejemplo, cuando en el ejemplo del capítulo 1 se llama a SendMessage para enviar un mensaje a un elemento de la ventana de diálogo, esta llamada se traduce directamente en una llamada a la función callback de elemento. Al llamar a SendMessage, si la ventana a la cual se envía el mensaje es creada por la misma tirad, el mensaje no es encolado sino que directamente se llama a la función. Por el contrario, si el mensaje fue enviado a una ventana de otra thread, el mensaje es encolado y procesado por el thread de la ventana destino, si es que lo hace. Esto garantiza que todos los mensajes enviados a una ventana sean procesados sean procesados por la ventana que creó la misma. Esto debe hacerse por varias razones, relacionadas principalmente con la seguridad del sistema. Por ejemplo, cuando el sistema operativo nos envía un mensaje como WM_MOUSEMOVE, el procesamiento del mismo debe hacerse con los parámetros de seguridad del usuario de la aplicación, no con los del sistema operativo. En caso contrario, la aplicación destino podría hacerse pasar por el sistema operativo y realizar cualquier operación. En segundo lugar, si una segunda thread enviase mensajes imprevisibles, podría causar problemas en la aplicación.

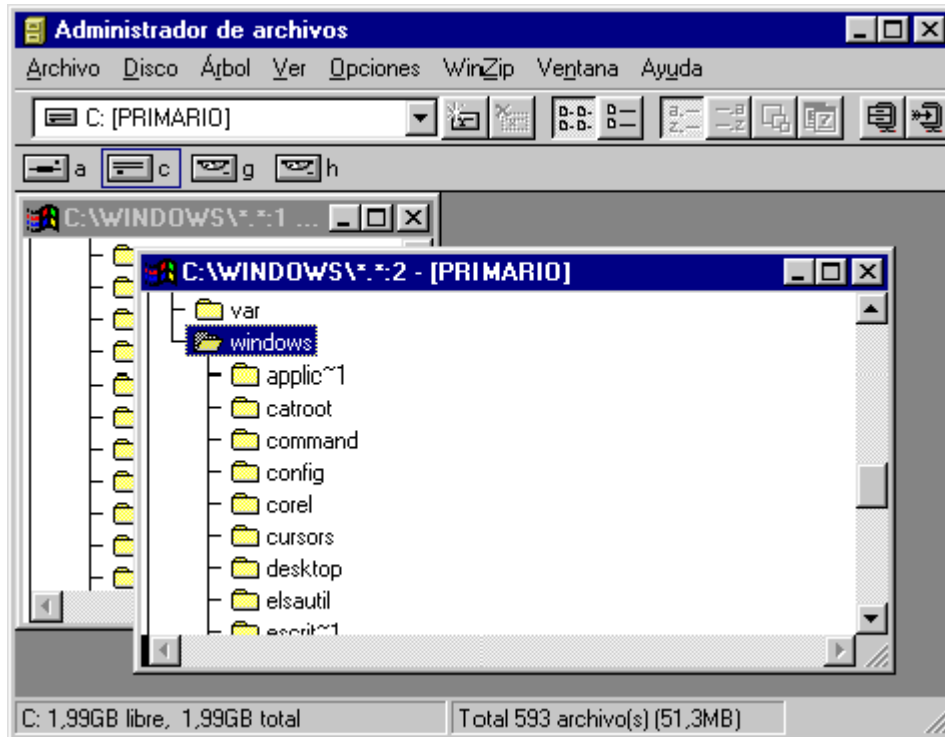
Los mensajes enviados llamando a PostMessage() siempre son encolados, salvo que la cola esté llena o no exista. Los mensajes encolados por una thread para sí misma tienen precedencia por sobre todos los mensajes enviados por otras threads.

Finalmente, debe aclararse que no necesariamente toda thread tiene una cola de mensajes. Esta cola se crea cuando la misma llama por primera vez a GetMessage(). Si no se llama a esta función, la cola no será creada y por ende no habrá forma de enviarle un mensaje a la misma.

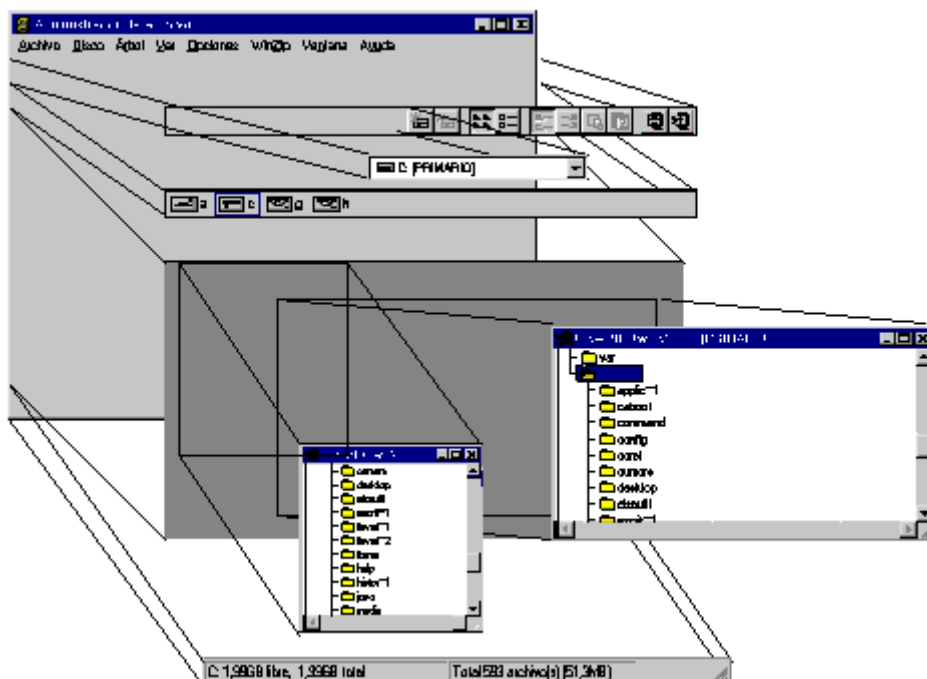
Es posible definir los parámetros prioridades tanto para las threads como para los procesos. La prioridad con la cual se ejecutará una thread es un resultado de la prioridad del proceso correspondiente y la prioridad propia de la thread. Para más información, ver documentación de Microsoft.

La necesidad de una nueva estructura de programación

Veamos un nuevo ejemplo, aún más complejo. Supongamos que se quiere desarrollar una aplicación que permita abrir múltiples ventanas. Un ejemplo de esto es el administrador de archivos de Windows:



Una ventana tan simple como esta, se compone en realidad de varias ventanas, unas dentro de otras. A continuación se muestra la forma en que se construyen estas ventanas. Concretamente, la aplicación se compone de una ventana principal, la cual contiene dos barras de tareas, una ventana "MDIClient", que contendrá las ventanas abiertas por el usuario, y una ventana para dibujar una barra de estado. Estas ventanas a su vez pueden contener otras más. De hecho, los botones suelen ser a su vez ventanas.



Se encuentra aquí un primer inconveniente. Vimos en el ejemplo anterior que la programación Windows se basa fundamentalmente en funciones callback. Estas funciones son llamadas por Windows, y puede definirse variables dentro de ellas, pero al salir de la función las mismas se destruyen. Entonces: ¿Dónde se definen las variables de las ventanas hija de un programa?

Hay tres respuestas posibles, igualmente inadecuadas.

La primera y más simple es definir todas las variables fundamentales del programa como globales. Esto es completamente inadecuado por todas las razones ya vistas.

La segunda solución consiste en declarar las variables dentro de cada función callback como estáticas (static), de forma tal que no se destruyan. Esta alternativa, si bien es viable y de hecho se utiliza para programas pequeños, presenta muchos inconvenientes. Uno de ellos es la imposibilidad (o dificultad) de abrir simultáneamente dos ventanas del mismo tipo, ya que se estará usando una única función callback para dos o más ventanas diferentes, y por lo tanto habrá que escribir un código para poder duplicar todas las variables para cada ventana.

La tercera, (y la que realmente se utiliza) es asociar a cada ventana una estructura con todas las variables de la misma. Toda ventana puede contener información adicional, definida por el usuario. Windows provee a tal fin la función: **SetWindowLong()** y **GetWindowLong()**, que entre sus opciones permiten asociar y leer un entero (datos del usuario) a la ventana (Ver opción: `GWL_USERDATA`). Este entero se suele utilizar para almacenar un puntero a todas las variables del programa.

A continuación se muestra un programa Windows básico para crear una aplicación de este tipo. La misma se compone de una ventana básica, la cual contendrá una ventana principal (comúnmente llamada MDI), la cual a su vez contendrá cuatro ventanas hijas. Estas ventanas hijas mostrarán en pantalla sus datos, los cuales serán simplemente un entero. Notar claramente como se crearon las variables de cada ventana. Notar también la forma en que se imprimen los datos en pantalla. Utilizando ya programación orientada a objetos, se utiliza un método de la clase para imprimir el contenido de la variable (a partir de ahora atributo) **a**.

Ya que las cuatro ventanas hijas deberán compartir la misma función de procesamiento de mensajes, y esta función no puede tener acceso a ninguna variable externa más que los cuatro parámetros que recibe (no quiero utilizar variables globales), ¿Cómo puedo conocer los datos de la ventana correspondiente al momento de procesar los mensajes?

Rta: Toda ventana tiene la capacidad de almacenar un número adicional, que puede ser utilizado para almacenar un puntero a datos de la ventana.

```
/* *****\
***                A R C H I V O S   A   I N C L U I R                ***
\*****/
#include
#include
#include
#include
/* Nota: La mayor parte del código referente a la verificación de errores
ha sido removido por claridad. */
/* *****\
***                D E F I N I C I O N E S                ***
\*****/
#define OK 0
```



```
#define ERR 1
HINSTANCE inst;
/*****
***                                P R O T O T I P O S                                ***
*****/
int RegistrarVentanaPrincipal (void);
int RegistrarVentanaHija (void);
HWND CrearVentanaHija (HWND Parent, int Numero);
LONG CALLBACK MainProc (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam);
LONG CALLBACK ChildProc (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam);
int Init (void);
/*****
int RegistrarVentanaPrincipal (void)
*****/
int RegistrarVentanaPrincipal (void)
{
    WNDCLASSEX wc;
    wc.cbSize = sizeof (wc);           // Debe contener el tamaño de la estructura
    wc.style = 0;                      // Estilo de la ventana
    wc.lpfnWndProc = MainProc;         // Funcion que procesará los mensajes
    wc.cbClsExtra = 0;                // Bytes adicionales al final de esta
estructura
    wc.cbWndExtra = 0;                // Bytes adicionales en los datos de la
ventana creada
    wc.hInstance = inst;
    wc.hIcon = NULL;                  // Icono de la ventana
    wc.hCursor = LoadCursor (NULL, MAKEINTRESOURCE (IDC_ARROW)); // Cursor que se
usará en la ventana.
    wc.hbrBackground = (void *)COLOR_APPWORKSPACE;
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "Clase Ventana"; // Nombre de este tipo de ventana
    wc.hIconSm = NULL;                // Icono pequeño de la ventana
    /* Registrar la ventana */
    if (RegisterClassEx (&wc) == 0)
        return ERR;
    return OK;
}
/*****
int RegistrarVentanaHija (void)
*****/
int RegistrarVentanaHija (void)
{
    WNDCLASSEX wc;
    wc.cbSize = sizeof (wc);           // Debe contener el tamaño de la estructura
    wc.style = 0;                      // Estilo de la ventana
    wc.lpfnWndProc = ChildProc;        // Funcion que procesará los mensajes
    wc.cbClsExtra = 0;                // Bytes adicionales al final de esta
estructura
    wc.cbWndExtra = 0;                // Bytes adicionales en los datos de la
ventana creada
    wc.hInstance = inst;
    wc.hIcon = NULL;                  // Icono de la ventana
    wc.hCursor = LoadCursor (NULL, MAKEINTRESOURCE (IDC_ARROW)); // Cursor que
se usará en la ventana.
    wc.hbrBackground = (void *)COLOR_WINDOWFRAME;
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "Clase Vent Hija"; // Nombre de este tipo de ventana
    wc.hIconSm = NULL;                // Icono pequeño de la ventana
    /* Registrar la ventana */
    if (RegisterClassEx (&wc) == 0)
        return ERR;
}
```

```

    return OK;
}

/*****\
int CrearVentanaHija (HWND Parent, int Numero)
\*****/
HWND CrearVentanaHija (HWND Parent, int Numero)
{
    HWND Win;
    /* Crear la ventana principal */
    Win = CreateWindowEx (
        WS_EX_WINDOWEDGE|WS_EX_CLIENTEDGE|WS_EX_MDICHILD,    // Estilos
        "Clase Vent Hija",    // Nombre de la clase registrada
        "Ventana Hija",    // Título de la ventana
        WS_CHILD|WS_VISIBLE|WS_CLIPSIBLINGS|WS_BORDER|WS_DLGFRAME|WS_THICKFRAME|
        WS_TABSTOP|WS_GROUP,    // Estilo de la ventana
        Numero*20,    // Posición x en pixels (usar posición por
        defecto)
        Numero*20,    // Posición y en pixels (usar posición por
        defecto)
        150,    // Ancho en pixels (usar valor por defecto)
        100,    // Alto en pixels (usar valor por defecto)
        Parent,    // Padre de la ventana (escritorio de
        Windows)
        NULL,    // Menú de la ventana
        inst,    // Instancia del programa
        NULL);    // Información adicional
    return Win;
}

/*****\
* P R O C E D I M I E N T O   D E   L A   V E N T A N A   P R I N C I P A L *
\*****/
LONG CALLBACK MainProc (HWND hWnd, UINT wParam, LPARAM lParam)
{
    /* Procesamiento de mensajes */
    switch (wParam)
    {
        case WM_CREATE:
            /* Aquí deben inicializarse las variables de la ventana */
            break;
        case WM_CLOSE:
            /* Cerrar la ventana */
            PostQuitMessage (0);
            break;
        default:
            /* Procesar todos los restantes mensajes */
            return DefWindowProc (hWnd, wParam, lParam);
    }
    return 0;
}

/*****\
***   P R O C E D I M I E N T O   D E   L A   V E N T A N A   H I J A   ***
\*****/
/* Datos de la ventana Hija */
class TDatosVentanaHija{
private:
    int a;
public:

```

```

    TDatosVentanaHija (void)
    {
        a = rand();
    }
    void EvPaint (HDC dc)
    {
        char Txt[20];
        sprintf (Txt, "%u", a);
        TextOut (dc, 10, 10, Txt, strlen(Txt));
    }
};

LONG CALLBACK ChildProc (HWND hWnd, UINT wMsg, WPARAM wParam, LPARAM lParam)
{
    /* Obtener el puntero a los datos de la aplicación */
    TDatosVentanaHija *p;
    if (wMsg == WM_CREATE)
    {
        p = new TDatosVentanaHija;
        SetWindowLong (hWnd, GWL_USERDATA, (int)p);
    }
    else
        p = (TDatosVentanaHija *)GetWindowLong (hWnd, GWL_USERDATA);
    /* Proceso de mensajes */
    switch (wMsg)
    {
        case WM_DESTROY:
            /* Destruir los datos de la ventana */
            delete p;
            break;
        case WM_PAINT:
            {
                HDC dc = GetDC (hWnd);
                p->EvPaint (dc);
                ReleaseDC (hWnd, dc);
                return DefWindowProc (hWnd, wMsg, wParam, lParam);
            }
            break;
        default:
            /* Procesar todos los restantes mensajes */
            return DefWindowProc (hWnd, wMsg, wParam, lParam);
    }
    return 0;
}

/*****
***          C R E A C I O N   D E   L A   V E N T A N A          ***
*****/
int Init (void)
{
    HWND Win;
    HWND MdiWin;
    CLIENTCREATESTRUCT ClientData;
    /* Registrar la ventana principal */
    RegistrarVentanaPrincipal ();
    /* Registrar la ventana Hija */
    RegistrarVentanaHija ();
    /* Crear la ventana principal */
    Win = CreateWindowEx (
        WS_EX_CLIENTEDGE,          // Estilos extendidos de la ventana
        "Clase Ventana",          // Nombre de la clase registrada
        "Ventana de ejemplo",     // Título de la ventana
        WS_TILEDWINDOW|WS_CLIPCHILDREN|WS_VISIBLE, // Estilo de la ventana

```

```

        CW_USEDEFAULT,          // Posición x en pixels (usar posición por
defecto)
        CW_USEDEFAULT,          // Posición y en pixels (usar posición por
defecto)
        CW_USEDEFAULT,          // Ancho en pixels (usar valor por defecto)
        CW_USEDEFAULT,          // Alto en pixels (usar valor por defecto)
        NULL,                    // Padre de la ventana (escritorio de
Windows)
        NULL,                    // Menú de la ventana
        inst,                    // Instancia del programa
        NULL);                  // Información adicional
/* Registrar la ventana MDI Cliente */
MdiWin = CreateWindowEx (
        WS_EX_CLIENTEDGE,        // Estilos extendidos de la ventana
        "MDIClient",            // Nombre de la clase registrada
        "",                      // Título de la ventana
        WS_CHILD|WS_VISIBLE|WS_CLIPCHILDREN|WS_CLIPSIBLINGS|WS_GROUP|WS_TABSTOP,
// Estilo de la ventana
        0,                      // Posición x en pixels (usar posición por
defecto)
        0,                      // Posición y en pixels (usar posición por
defecto)
        600,                    // Ancho en pixels (usar valor por defecto)
        400,                    // Alto en pixels (usar valor por defecto)
        Win,                    // Padre de la ventana (escritorio de
Windows)
        NULL,                    // Menú de la ventana
        inst,                    // Instancia del programa
        &ClientData);           // Información adicional
/* Crear cuatro ventanas hijas */
Win = CrearVentanaHija (MdiWin, 1);
Win = CrearVentanaHija (MdiWin, 2);
Win = CrearVentanaHija (MdiWin, 3);
Win = CrearVentanaHija (MdiWin, 4);

return OK;
}

/*****
***                               W I N M A I N                               ***
*****/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
lpCmdLine, int nShowCmd)
{
    MSG msg;
    inst = hInstance;
    Init ();
    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}
/*****
***                               F I N                               ***
*****/

```

Capítulo IV - Hacia un nuevo modelo de programación Windows

La necesidad de un nuevo modelo de programación Windows

El ejemplo del capítulo anterior comienza a presentar algunos de los inconvenientes de la programación Windows. La forma en que se crean, obtienen y destruyen las variables o atributos de una ventana resulta bastante confusa y complicada. Además, tan sólo para crear el entorno de la aplicación debieron escribirse muchas líneas de código, las cuales serán idénticas para todas las aplicaciones que trabajen con múltiples ventanas, además de que nuestro ejemplo fue corto y muy limitado. Pero afortunadamente se cuenta con la Programación Orientada a Objetos y una herramienta muy poderosa de C, que hasta ahora no se había utilizado mucho: las *macros*. Utilizando estas dos herramientas, varias empresas tales como Borland y Microsoft han desarrollado bibliotecas muy grandes, que simplifican enormemente la programación Windows. La biblioteca de Borland se conoce con el nombre de *Object Windows Library* (OWL) y la de Microsoft con el nombre de *Microsoft Foundation Classes* (MFC). Ambas son muy parecidas, y se basan en las mismas ideas.

Borland - OWL (Object Windows Library)

Para empezar, pocas veces se utilizan los parámetros de la función *WinMain()* (salvo *hInstance*, que suele almacenárselo como una variable global). Por el contrario, sí resultaría muy cómodo tener los parámetros de la línea de comandos separados en la forma que se tenía con la función *main()*. Por esta razón estas bibliotecas ya definen la función *WinMain()*. En el caso de OWL esta llamará a la función *OwlMain()*, (que será el nuevo punto de entrada al programa), que recibe los parámetros en la forma habitual que se tenía en DOS.

En el caso de OWL, se define una clase *TApplication*, la cual se encargará de crear la ventana principal (en el método *InitMainWindow*) y ejecutar el ciclo habitual de mensajes:

```
while (GetMessage (&msg, NULL, 0, 0))
{
    TranslateMessage (&msg);
    DispatchMessage (&msg);
}
```

Nosotros podemos ahora definir una nueva clase, heredera de *TApplication*, y sobrecargar el método *InitMainWindow*, para crear la ventana que nosotros querremos. Ya que este método es virtual, no se llamará el método original sino el nuevo.

```
class TmiAplicacion:public TApplication
{
    char *Parametro;
public:
    TmiAplicacion (char *Param):TApplication()
    {
        Parametro = Param;
    }
    virtual void InitMainWindow()
    {
        TFrameWindow *Win = new TFrameWindow (NULL, "Ejemplo");
        SetMainWindow (Win);
    }
};
```

Notar que en nuestro método *InitMainWindow*, se crea un nuevo objeto, de tipo **TFrameWindow**. Este objeto define una ventana básica, y para hacer algo útil nuestra aplicación deberá construir un nuevo objeto, como heredero de esta clase. Veamos como queda nuestro programa hasta ahora:

```
#include<owl\applicat.h>
#include<owl\framewin.h>

class TMiAplicacion:public TApplication
{
    private:
        char *Parametro;
    public:
        TMiAplicacion (char *Param):TApplication()
        {
            Parametro = Param;
        }
        virtual void InitMainWindow()
        {
            TFrameWindow *Win = new TFrameWindow (NULL, "Ejemplo");
            SetMainWindow (Win);
        }
};

int OwlMain(int argc, char *argv[])
{
    TMiAplicacion App(argv[1]);
    App.Run();
    return 0;
}
```

Muy bien, pero para poder desarrollar una aplicación interesante, no debemos utilizar un objeto de tipo **TFrameWindow**, sino uno nuevo derivado de este, con las modificaciones que nosotros querramos. Veamos como hacerlo:

```
/******\
*** # I N C L U D E S ***
\*****/
#include<owl\applicat.h>
#include<owl\framewin.h>
/******\
*** V E N T A N A P R I N C I P A L ***
\*****/
class TMiVentana:public TFrameWindow
{
    private:
        int x,y;
    public:
        TMiVentana (TWindow *Padre, char *Titulo):TFrameWindow(Padre,
Titulo)
        {
            x = y = 0;
        }
        LRESULT BotonIzquierdo (WPARAM, LPARAM lParam)
        {
            x = LOWORD (lParam);
            y = HIWORD (lParam);
            Invalidate ();
            return 0;
        }
        void Paint(TDC& dc, bool erase, TRect& rect)
```

```
        {
            dc.TextOut (x, y, "Hola mundo");
        }
        DECLARE_RESPONSE_TABLE (TMiVentana);
};
DEFINE_RESPONSE_TABLE1 (TMiVentana, TFrameWindow)
    EV_MESSAGE (WM_LBUTTONDOWN, BotonIzquierdo),
END_RESPONSE_TABLE;
/***** APLICACION *****/
class TMiAplicacion:public TApplication
{
    public:
        void InitMainWindow()
        {
            TMiVentana *Win = new TMiVentana (NULL, "Ejemplo");
            SetMainWindow (Win);
        }
};
/***** O W L M A I N *****/
int OwlMain(int /*argc*/, char /*argv*/[])
{
    TMiAplicacion App;
    App.Run();
    return 0;
}
```

Nuestra aplicación se basa en una ventana que imprime "Hola mundo", en la posición donde se presiona el botón izquierdo del mouse.

Notar la definición de una "tabla de respuestas". Esta tabla es en realidad un conjunto de macros que se expanden para formar una función y una tabla, encargada de procesar los mensajes y llamar a la función correspondiente. Lo que hacen es construir las funciones *callback*, que se vio en los capítulos anteriores, y llamar a la función correspondiente del objeto. Para más información, ver el ejemplo del capítulo anterior. La forma en que se define esta tabla de respuesta es muy simple:

En la clase debe incluirse la línea:

```
DECLARE_RESPONSE_TABLE (clase derivada);
```

Fuera de la clase deben escribirse las líneas:

```
DEFINE_RESPONSE_TABLE1 (Clase derivada, clase base)
    EV_MESSAGE (mensaje, Función a llamar),
    ...
    EV_COMMAND(comando, Función a llamar),
    ...
END_RESPONSE_TABLE;
```

En el caso de EV_MESSAGE, la función recibir dos enteros (WPARAM y LPARAM) y retornar un entero (LRESULT). En el caso de EV_COMMAND, la función no debe recibir ni retornar nada.



Importante: La declaración de la tabla de respuesta es obligatoria para todo objeto derivado de TWindow o sus derivados, aún cuando esta esté vacía.

Microsoft MFC

Microsoft simplificó enormemente la programación bajo Windows, agregando a su compilador un constructor automático de código fuente (Wizard). Las clases desarrolladas son similares a las de Borland, pero además se tiene una gran cantidad de herramientas para agregar y modificar código automáticamente. Veamos un ejemplo. Queremos desarrollar un programa capaz de mostrar en pantalla un texto ingresado por el usuario.

Paso 1: Creación del proyecto:

1. Seleccionar en el menú archivos la opción **new** (nuevo), y luego seleccionar la opción nuevo proyecto.
2. Seleccionar como tipo de proyecto la opción: MFC AppWizard (exe), indicando un nombre de directorio donde crear el proyecto, y como nombre del mismo: ejemplo1
3. En el paso 1 debe seleccionarse la opción **Single document**, es decir un único documento.
4. Todos los pasos siguientes pueden dejarse con las opciones por defecto.

Automáticamente se generará un proyecto con los siguientes archivos:

ejemplo1.cpp:

Contiene la definición de las clases CEjemplo1App, derivada de CWinApp y una ventana de dialogo "About", derivada de CDialog. La primera almacenará "la aplicación", es decir, es la base sobre la cual se construirá todo el programa.

ejemplo1.rc

Contiene los recursos del programa. Particularmente contiene la definición de la ventana CAbout, entre otras cosas.

ejemplo1Doc.cpp

Contienen la información del documento. En programas que pueden contener múltiples documentos, cada uno de ellos debe almacenar la información en objetos de este tipo.

ejemplo1View.cpp

La vista del documento. Un documento puede contener varias vistas.

MainFrm.cpp

Contiene el código referente al entorno del programa, tales como la barra de estado, etc.

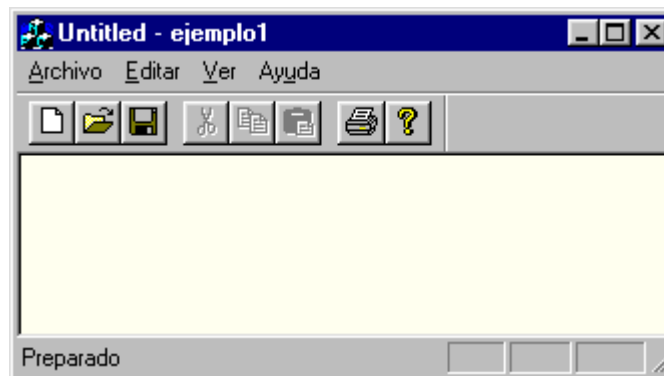
ReadMe.txt

Datos de la aplicación.

StdAfx.cpp

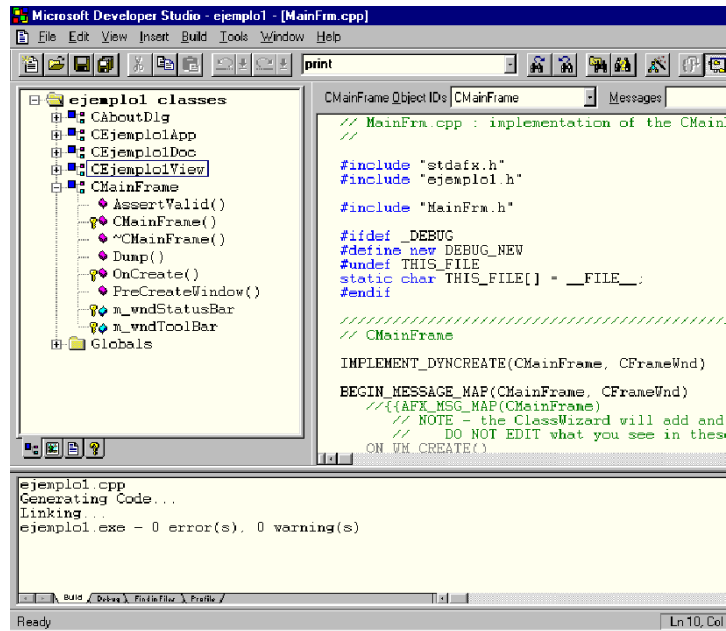
Contiene definiciones varias, y los archivos *header* a precompilar.

Este es un buen momento para compilar el programa. Pruebe el funcionamiento del programa e intente leer y entender el código.



Paso2: Agregar una variable String y mostrarla.

1. El entorno de desarrollo contiene una ventana con una etiqueta denominada ClassView, que muestra todas las clases con sus métodos y atributos. Selecciónela, y seleccione dentro de ella la clase CEjemplo1Doc.



2. Presione sobre ella el botón derecho del mouse, y seleccione la opción **agregar variable**.
3. Ingrese como tipo de variable: **CString**, como nombre **m_Info** y como acceso **Público**. Microsoft acostumbra llamar a las variables de una clase con el prefijo **m_**, para indicar que son miembro de la misma.
4. Abra el método **CEjemplo1Doc**, (el constructor de la clase)
5. Reemplace el comentario **// TODO** por la línea:

```
m_Info = "Hola mundo";
```
6. Seleccione y abra la clase **CEjemplo1View** en la etiqueta **ClassView**.
7. Haga doble click en el método **OnDraw()**. Este método será llamado cada vez que se deba dibujar el contenido de la ventana.
8. Reemplace el comentario **// TODO** por la línea:

```
pDC->TextOut (10,10,pDoc->m_Info);
```
9. Compile nuevamente el programa.

Paso3: Agregar un dialogo

1. Seleccione la segunda etiqueta de este mismo menú, donde se encuentran los recursos, y abra el menú del programa.
2. Agregue una opción en el menú View, con el ID: **ID_VIEW_TEXTO** y el texto: **Texto de la ventana**. Puede agregar un texto de ayuda si lo desea.
3. Agregue un dialogo al programa. Esto lo puede hacer seleccionando la opción **Insertar recurso** del menú **Insertar**, y seleccionando allí el recurso diálogo.
4. Inserte un recuadro de edición de texto, pulse sobre él el botón derecho del mouse y seleccione la opción **Propiedades**. Modifique el ID del recuadro a **IDC_TEXTO**.
5. Seleccione las propiedades del dialogo, y especifique un ID: **IDD_TEXTO**, Caption: **Ingrese el texto a mostrar**.
6. Haga click con el botón derecho del mouse sobre la ventana, y seleccione la opción **ClassWizard**.
7. En la ventana **Adding a Class** seleccione la opción: **Crear nueva clase**.
8. Ingrese **CDlgTexto** como nombre de la clase y presione ENTER. El entorno ha creado para usted una nueva clase CDlgTexto, derivada de CDialog, en los archivos DlgTexto.cpp y DlgTexto.h. Puede ver que esta información ha ya sido agregada en la etiqueta ClassView.
9. Seleccione la etiqueta **Member Variables**, en la cual se muestran las variables de la clase.
10. Seleccione la opción **IDC_TEXTO**, y presione el botón **Add Variables**. Ingrese como nombre de la variable: **m_Texto**, y presione el botón **OK**. Cierre el dialogo **MFC ClassWizard** presionando **OK**.
11. Abra el archivo **ejemplo1View.cpp**, y agregue la:


```
ON_COMMAND(ID_VIEW_TEXTO, ModificarTexto)
```


a continuación de la línea: `BEGIN_MESSAGE_MAP(CEjemplo1View, CView).`
12. Seleccione la clase **CEjemplo1View**, presione el **botón derecho** y seleccione la opción Agregar función.
13. Ingrese **void** como tipo de función y **ModificarTexto** como nombre.

14. Dentro del código de la función *ModificarTexto* recién creada ingrese:

```
15.CDlgTexto DlgTexto; // Creo el diálogo
16.CEjemplo1Doc* pDoc = GetDocument();
17.DlgTexto.m_Texto = pDoc->m_Info; // Cargo el texto inicial del diálogo
18.if (DlgTexto.DoModal()==IDOK) // Muestro el dialogo
19.{
20.    pDoc->m_Info = DlgTexto.m_Texto; // Leo el texto ingresado
21.    Invalidate (); // Redibujar la ventana
}
```

22. Vaya al comienzo del archivo y agregue la siguiente línea, debajo de los `#include` ya existentes:

```
#include "dlgTexto.h"
```

23. Ingrese este mismo `#include` al final del archivo `stdafx.h`. Esto hará que el archivo sea precompilado.

24. Ingrese la siguiente línea al comienzo del archivo `dlgTexto.h`.

```
#include "resource.h"
```

25. Compile el programa nuevamente.

Puede verse que en el menú *Ver* existe la opción *ModificarTexto*, que permite modificar el texto mostrado en pantalla.