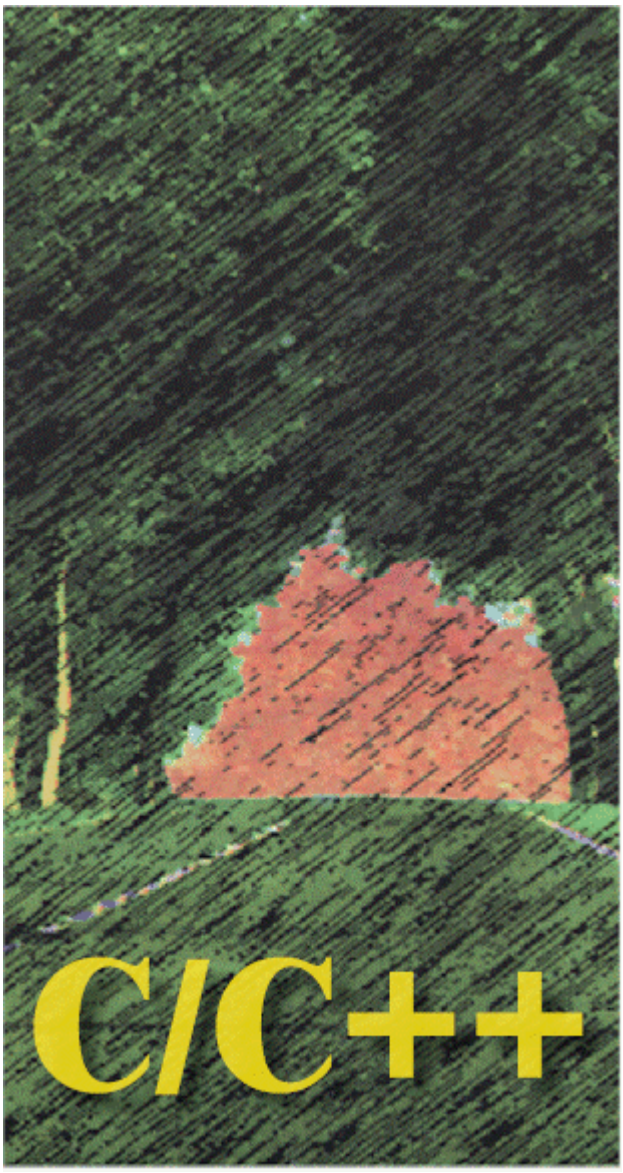
Texto diseñado para enseñar a desarrollar aplicaciones basadas en lenguaje C++ a través de Microsoft Visual C++ 6. El texto cubre los fundamentos del lenguaje, las técnicas de orientación a objetos y el modelo de trabajo con Visual C++ en su versión 6.

A lo largo del texto se desarrollan numerosos ejemplos listos para ser cargados desde Visual C++.

Se requiere tener como mínimo conocimientos de fundamentos de programación y conocer al menos un lenguaje, sea éste el que sea, así como conocer el S.O. Windows a nivel de usuario.



PROGRAMACIÓN EN C/C++ VISUAL C++ 6

GRUPO Eidos

Programación en C/C++, Visual C++
Versión 1.0.0

www.LaLibreriaDigital.com

Índice

1

Introducción

El lenguaje C de programación

Bienvenidos al curso de programación en C, C++ y Visual C++. En estos primeros capítulos pretendemos introducir al usuario en el lenguaje de programación C. El conocimiento de C no es necesario para manejar C++ y utilizar Visual C++, sin embargo, la estructura de base de C++ hace que el conocimiento de C se convierta en conveniente para un programador de C++, sobre todo porque la predominancia de C como lenguaje de programación de sistemas durante las última décadas hace que sea común la coexistencia de código escrito en ambos lenguajes.

Los orígenes de C datan de finales de los años 60 y principios de los años 70. En particular, se basó en el lenguaje **BCPL** desarrollado por Martin Richards, y en el lenguaje **B** desarrollado por Ken Thompson en 1970 (del que toma su nombre), en un sistema UNIX, sistema operativo inventado por Dennis Ritchie y Brian Kernighan a finales de los 60 en los Laboratorios Bell de AT&T. En 1972, y en los mismos Laboratorios Bell, **Dennis Ritchie** desarrolló un lenguaje de programación para sistemas operativos (UNIX, en particular) para máquinas Digital, en concreto DEC PDP-7 y DEC PDP-11.

La evolución posterior del lenguaje viene condicionada por una amplia difusión en el entorno docente norteamericano, en el que UNIX se convierte en el sistema operativo estándar, y C en su lenguaje de programación. Con ese éxito Brian Kernighan y Dennis Ritchie escriben en 1978 el primer estándar de facto para C, "*The C Programming Language*". Este es el primer intento de normalización formal del lenguaje, que se convierte en la base de los compiladores de C desarrollados en esa época.

De la amplia difusión de C, que ya alcanza a la industria, surge la necesidad de una norma más rígida. Así, el ANSI (*American National Standards Institute*, el Instituto Nacional Americano de Estándares)

crea un grupo de normalización en 1983, que finaliza sus trabajos en 1988, con el primer estándar oficial. Un año después (1989) aparece la segunda edición de "*The C Programming Language*", basado en el estándar ANSI, y se produce la aceptación generalizada de la norma ANSI en la industria. Como se verá en el capítulo correspondiente a la introducción a C++, durante los años 80 Bjarne Stroustrup desarrolla el lenguaje C++, para extender el lenguaje C a la Programación Orientada a Objetos (POO).

Requisitos iniciales del curso

Durante el desarrollo del temario del curso, se supondrá que el alumno

- Conoce algunos métodos para escribir y verificar programas en algún lenguaje de programación;
- Está familiarizado con los conceptos de variable, asignaciones, bucles y funciones (llamadas subrutinas en otros lenguajes); y
- Conoce los conceptos básicos de la programación estructurada.

En particular, recomendamos la lectura detallada del capítulo de introducción a Visual C++ para la realización de los ejemplos.

Características generales de C

C fue concebido como un lenguaje para crear sistemas operativos, aunque se ha convertido en un lenguaje de programación de propósito general. En particular, podemos mencionar que existen librerías desarrolladas para crear entornos gráficos en C para entornos UNIX (X-Window), y que las primeras versiones de Windows fueron desarrolladas con bibliotecas escritas en C, hasta la aparición de las Microsoft Foundation Classes (**MFC**) y de la Object Window Library (**OWL**) de Borland (ahora Inprise) escritas en C++. Debemos reseñar que C fue creado en un principio por una sola persona, por lo que está muy influido por su forma de trabajar y su entorno (utilización de UNIX y la filosofía de los sistemas abiertos). En particular, C es clave en el desarrollo del sistema operativo Linux.

C (al contrario que los lenguajes B y BCPL, que eran lenguajes sin tipos) posee tipos de datos y una jerarquía de datos derivados (punteros, arrays, estructuras y uniones) estricta. C maneja expresiones, como muchos otros lenguajes de programación. Una expresión se define del siguiente modo:

```
expresión = operadores + operandos
```

C posee además las construcciones propias del control estructurado:

```
if-else, switch, for, while, do-while, break, continue
```

que se describirán en detalle más adelante.

C es un lenguaje de "bajo nivel", aunque permite una estructura de programación de mayor nivel. Sin embargo, C no proporciona ni soporta:

- Operaciones con objetos compuestos: no existen cadenas de caracteres, clases y otros tipos de objetos compuestos.

- Operaciones de Entrada/Salida (E/S).
- Operaciones con ficheros.
- Multiprogramación.
- Operaciones paralelas (hilos) y su sincronización.
- Co-rutinas.

Sin embargo, las tres primeras operaciones sí son soportadas por la **biblioteca estándar**. También debemos señalar que C es independiente de la arquitectura de la máquina.

Proceso de elaboración de un programa de software

Cómo es conocido por el alumno, el proceso de elaboración de un programa de software consta de seis fases:

1. **Análisis**: en el que aparece la necesidad de un proceso de mecanización, se elaboran las especificaciones que desembocan en métodos y algoritmos a utilizar, y se escriben los diagramas de flujo y el pseudocódigo.
2. **Edición**: se desarrolla la lista de instrucciones en un editor de textos o en un Entorno de Desarrollo Integrado (IDE).
3. **Compilación**: traducción del código fuente a la arquitectura de la máquina que se utiliza. Por defecto, se obtiene un fichero con la extensión ".OBJ" (en Windows 95/98/NT).
4. **Enlazado** (linkado): unión del código elaborado por el programador con las bibliotecas estándar del sistema o con bibliotecas elaboradas por otros. Se obtiene un fichero ejecutable (extensión por defecto: ".exe" en Windows 95/98/NT).
5. **Depuración** (Debugging): eliminación de errores en el proceso de compilación (sintaxis, dimensionado, etc.) o en el proceso de enlazado.
6. **Ejecución** del programa.

En este curso, nos ocuparemos del desarrollo de las fases 2 a 5 utilizando C, C++ y Microsoft Visual C++. Reiteramos aquí la recomendación al alumno de la lectura del capítulo 1 de la parte sobre Microsoft Visual C++.

Un tutorial rápido

Vamos a escribir rápidamente algunos programas en C. Con ello el alumno podrá familiarizarse con las propiedades principales del lenguaje. Antes de proseguir, recordemos que los componentes principales de un programa en C son:

- Funciones: conjunto de instrucciones a computar (equivalen a las funciones y subrutinas en Fortran, o a las funciones y procedimientos en Pascal).
- Variables: elementos donde se almacenan valores para realizar determinadas operaciones.

Todo programa está compuesto por **instrucciones o sentencias**. Las sentencias terminan siempre en punto y coma, por ejemplo:

```
return 0;
```

Las sentencias que forman parten de una función, o las compuestas, se agrupan con llaves: { }

```
while (i == 0)
{
  printf ("Presiona una tecla");
  /* ... */
}
```

Código fuente 1

Los espacios en blanco, los tabuladores y los saltos de línea no tienen significado especial en la compilación del programa. Deben utilizarse únicamente por claridad en la presentación del código. Vemos un ejemplo en el Código fuente 2.

```
int i;
int j;
/*equivale a:*/
int i; int j;
```

Código fuente 2

Los **comentarios** comienzan con /* y terminan con */. Se pueden prolongar durante varias líneas, pero no se pueden anidar.

```
printf ("Presiona una tecla"); /* Esto es un comentario
                                  que continúa en la segunda línea */
```

Código fuente 3

Hola Mundo

Continuando la tradición establecida por Kernighan y Rithie en sus dos ediciones históricas, en el Código fuente 4 se encuentra nuestro primer programa.

```
/* Ejemplo 1 */
#include <stdio.h> /* Llamada a la librería estándar */
int main() /* Programa principal */
{
  printf ("Hola, mundo!\n"); /* Llamada a función */
  return 0;
}
```

Código fuente 4

Este simple programa que se limita a imprimir en pantalla el mensaje **Hola, mundo!** presenta ya diversos aspectos importantes del lenguaje. La primera sentencia llama a una de las bibliotecas estándar, en concreto a la biblioteca de entrada/salida (*input/output*) **stdio.h**. A continuación, sigue el programa principal, main(), cuyo cuerpo de instrucciones está contenido entre dos llaves. Finalmente, el programa invoca a la función de la biblioteca estándar printf que imprime por pantalla sus argumentos, para devolver posteriormente al sistema el valor 0, indicándole a éste que se ha completado con éxito el programa.

Otros ejemplos

Después de este ejemplo tan sencillo, elaboremos algunos ejemplos típicos más complejos. Consideremos un programa que crea una tabla de conversión de grados Celsius a grados Fahrenheit. La fórmula conocida de transformación aparece en el Código fuente 5.

```
&ordm;C = 5 / 9 * (&ordm;F - 32)
```

Código fuente 5

Que podríamos traducir a C como muestra el Código fuente 6.

```
/* Ejemplo 2 */
#include <stdio.h>
int main()
{
  int fahr, celsius; /* Declaración de variables */
  int lower, upper, step;

  lower = 0; /* Límite inferior */
  upper = 300; /* Límite superior */
  step = 20; /* Paso entre elementos */

  fahr = lower;
  while (fahr <= upper)
  {
    celsius = 5 * (fahr - 32) / 9;
    printf ("%d\t%d\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

Código fuente 6

En este segundo ejemplo, vemos otras peculiaridades de C. En primer lugar, se observa que es **obligatorio** declarar variables. Así, en este ejemplo declaramos diversos números enteros, y les asignamos posteriormente un valor. Es importante conocer previamente la representación en número de bits (o bytes) de los números enteros o reales de la máquina en la que trabajamos para evitar problemas de portabilidad. Por ejemplo, es típico que:

- Los números enteros (int) se representen con 16 bits (2 bytes), con lo que su rango varía desde -32768 hasta 32767.

- Los números reales (float) se representen con 32 bits (4 bytes), con lo que su rango varía desde 10^{-38} hasta 10^{+38}.

Es de interés la sentencia while, que ejecuta su cuerpo (entre llaves) mientras la expresión lógica entre paréntesis (es decir, fahr menor o igual a upper) sea cierta. La función de la biblioteca estándar printf ofrece ahora una lista de argumentos en lugar de un único argumento. El primer argumento es una cadena de caracteres que da formato a la impresión, y los términos siguientes son las variables a imprimir. Los símbolos %d indican impresión de enteros, y \t es el carácter tabulador.

Este ejemplo se puede elaborar un poco más. Observemos que la fórmula para obtener grados Celsius implica que no se va a obtener siempre un número entero. En el caso de que este error de redondeo nos preocupase, conviene definir la variable celsius como real.

```
 /* Ejemplo 3 */
#include <stdio.h>
int main()
{
  float celsius; /* Declaración de variables */
  int lower, upper, step, fahr;

  lower = 0; /* Límite inferior */
  upper = 300; /* Límite superior */
  step  = 20; /* Paso entre elementos */

  fahr = lower;
  while (fahr <= upper)
  {
    celsius = (5.0 / 9.0) * (fahr - 32.0);
    printf ("%d\t%6.1f\n", fahr, celsius);
    fahr = fahr + step;
  }
  return 0;
}
```

Código fuente 7

Observe que la representación de un número real implica el uso del punto decimal. Además, existen reglas de conversión entre enteros (int) y reales (float), y se utiliza el símbolo %f para imprimir números reales. El Código fuente 7 se puede simplificar aún más, como se puede ver en el Código fuente 8.

```
/* Ejemplo 4 */
#include <stdio.h>
int main()
{
  int fahr; /* Declaración de variables */

  for (fahr = 0; fahr <= 300; fahr = fahr + 20)
  {
    printf ("%d\t%6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32.0));
  }
  return 0;
}
```

Código fuente 8

Ahora hemos definido un bucle for, que parte de una variable con un valor inicial, e incrementa en saltos de 20, hasta el máximo de 300. La fórmula se incluye en el propio cuerpo de la función printf. El Código fuente 9 introduce nuevos elementos.

```
/* Ejemplo 5 */
#include <stdio.h>
#define LOWER 0
#define UPPER 300
#define STEP  20

int main()
{
  int fahr; /* Declaración de variables */
  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
  {
    printf ("%d\t%6.1f\n", fahr, (5.0 / 9.0) * (fahr - 32.0));
  }
  return 0;
}
```

Código fuente 9

En este caso, mediante la sentencia #define, definimos valores constantes que el compilador sustituye allá donde aparecen. Con ello se obtiene mayor flexibilidad a la hora de redefinir constantes en el código.

```
#define nombre_variable texto_que_reemplaza
```

Cadenas de caracteres

En este caso vamos a describir brevemente una función para manipular cadenas de caracteres.

```
/* Ejemplo 6 */
#include <stdio.h>

int main()
{
  int c;
  while ( (c = getchar()) != EOF )
    putchar (c);
  return 0;
}
```

Código fuente 10

En C, los caracteres se representan por un entero que corresponde a su código ASCII. Existe un carácter especial definido en la biblioteca estándar como EOF, que corresponde al fin de archivo. El programa presentado se limita a repetir por pantalla aquellos caracteres introducidos. La sentencia c = getchar(), lee el carácter introducido por pantalla, mientras sea distinto (!=) del fin de archivo. El carácter leído se imprime por pantalla con putchar.

Hemos introducido en este capítulo distintos elementos del lenguaje C. En los capítulos siguientes se desarrollarán con profundidad, pero esta introducción permitirá crear ejemplos más complejos en los siguientes capítulos.

Ejercicios

1. Compilar y Ejecutar el programa HolaMundo.c con Microsoft Visual C++.

2. Modificar el ejemplo 6 sustituyendo putchar por printf. (Nota: el código de formato para caracteres es %c).

3. Escribir un programa en C que imprima en pantalla el valor numérico de EOF. Comprobar que es -1.

4. Escribir un programa en C que cree una tabla de conversión de Euros a pesetas desde 1 a 100 Euros (Nota: 1 Euro = 166.386 pts).

5. Modificar el programa anterior para que el lector introduzca su nombre, y lo imprima de nuevo.

Respuestas a los ejercicios

1. Modificar el ejemplo 6 sustituyendo putchar por printf. (Nota: el código de formato para caracteres es %c).

```
#include <stdio.h>

int main()
{
  int c;
  while ( (c = getchar()) != EOF )
    printf("%c", c);
  return 0;
}
```

2. Escribir un programa en C que imprima en pantalla el valor numérico de EOF. Comprobar que es -1.

```
#include <stdio.h>

int main()
{
  printf("%d\n", EOF);
  return 0;
}
```

3. Escribir un programa en C que cree una tabla de conversión de Euros a pesetas desde 1 a 100 Euros (Nota: 1 Euro = 166.386 pts).

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

```
#define LOWER 0
#define UPPER 100
#define EURO 166.386

int main()
{
  int euro; /* Declaración de variables */
  for (euro = LOWER; euro <= UPPER; euro = euro + 5)
  {
    printf ("%3d\t%8.2f\n", euro, euro * EURO);
  }
  return 0;
}
```

4. Modificar el programa anterior para que el lector introduzca su nombre, y lo imprima de nuevo.

```
#include <stdio.h>

#define LOWER 0
#define UPPER 100
#define EURO 166.386

int main()
{
  int c; /* Declaración de variables */
  printf ("Como te llamas? ");
  while ((c = getchar()) != '\n') putchar(c);
  printf ("\n");
  return 0;
}
```

2

Tipos, operadores y expresiones

Introducción

Como el alumno con conocimientos de programación sabe, los programas manipulan variables y constantes. En C y C++ existen distintas unidades fundamentales que vamos a definir.

- Declaración: es una lista de las variables o funciones a usar, su tipo y opcionalmente, su valor inicial.

```
int i = 2;
```

- Operador: es un elemento de un programa que especifica combinaciones de variables.

```
x = a + b;
```

- Expresiones: es una combinación de variables, constantes y operadores para producir variables nuevas.

```
y = f (a+b) * (x - y);
```

En las secciones siguientes estudiaremos la "gramática" de C para representar estos elementos.

Nombres de variables

En C existen las siguientes convenciones sobre los nombres de variables:

- Los nombres de variables constan de letras, dígitos y el carácter "_" (*underscore*).
- El primer carácter debe ser una letra o "_". (No es aconsejable, sin embargo usar este carácter porque es utilizado frecuentemente por funciones de la biblioteca estándar).
- Las mayúsculas y minúsculas son diferentes a efectos de los nombres de las variables, al contrario de lo que ocurre con otros lenguajes (Fortran).
- El estándar garantiza al menos la singularidad de los 31 primeros caracteres de un nombre para las variables internas.
- Para las variables externas (usadas por otros lenguajes o ensambladores), el estándar sólo garantiza 6 caracteres, y no distingue entre mayúsculas y minúsculas.

Como consejos para la denominación de variables recomendamos:

- Usar nombres significativos para las variables.
- Usar una única letra para índices y variables locales.

Tipos de datos y sus tamaños

En el estándar de C no están definidos en detalle el número de bytes que cada implantación debe asignar a las variables por defecto. Típicamente, se tiene

- char: carácter (1 byte).
- int: entero (2 ó 4 bytes).
- float: números reales de precisión sencilla (coma flotante, 4 bytes).
- double: números reales de doble precisión (8 bytes).

Modificadores

Son elementos que aplicados a un tipo de variable cambian su tamaño, cuando la arquitectura lo permita. Por ejemplo:

- short: aplicable a enteros.
- long (l/L): aplicable a enteros (long int, 4 bytes) o números en doble precisión (long double, 12 bytes).
- unsigned (u/U) / signed: aplicables a enteros o caracteres. Cambian la aritmética de variable aplicable (con o sin signo).

Las bibliotecas <limits.h> y <float.h> contienen constantes simbólicas relacionadas con el tamaño y los límites. Existe la función sizeof(tipo) que devuelve como valor el número de bytes que un tipo de variable ocupa en memoria.

Constantes

Es muy frecuente encontrarse con la situación de tener que definir ciertas variables en un programa como constantes, para que no puedan ser modificadas a lo largo del programa. Para ello, C facilita una serie de herramientas que permiten definir estas constantes. Una constante puede ser:

- numérica: donde puede tomar un valor entero, entero long (sufijo "l" o "L"), sin signo (unsigned, "u" o "U"), octal (entero que comienza con "0"), hexadecimal ("0x" o "0X") y real (número con punto decimal: 123.4, o exponente: 12.0e-2).

- de caracteres: su valor es un entero que corresponde al conjunto de caracteres de la máquina utilizada (para un PC es el código ASCII), en el caso de un único carácter, o varios caracteres para formar una cadena constante, como veremos más adelante.

Además, existen una serie de caracteres con un significado especial, denominados caracteres de escape. Estos aparecen en la Tabla 1.

\a	Alarma	\\	barra invertida (\)
\b	Backspace	\?	interrogación
\f	alimentación de hoja	\'	comilla sencilla
\n	nueva línea	\"	comillas dobles
\r	retorno de carro	\0oo	número octal
\t	tabulador	\xhh	número hexadecimal
\v	tabulador vertical	\0	carácter nulo

Tabla 1

La sintaxis típica es:

```
const int i = 2;
```

El modificador const indica al compilador que la variable no puede ser modificada en el programa. También existe la posibilidad de definir expresiones constantes. Éstas son expresiones definidas con la macro #define que se evalúan en tiempo de compilación.

```
#define MAXLINE 1000
```

Cadenas de caracteres

Las cadenas de caracteres son uno de los tipos más utilizados en los lenguajes de programación. En la mayoría de los lenguajes son un tipo propio, pero en C se definen con un puntero (veremos más adelante en detalle qué es un puntero) al inicio de la cadena. Una cadena se define como cero o más caracteres delimitados por comillas dobles.

```
char *string = "Esto es una cadena de caracteres";
```

Código fuente 11

Las cadenas de caracteres terminan con el carácter nulo ('\0'). Por ello ocupan un byte más en memoria. Por ejemplo, 'x' ocupa un único byte, pero "x" ocupa dos: 'x'+'\0'. La función de la biblioteca estándar strlen devuelve el tamaño de la cadena, sin contar el carácter nulo de terminación.

Constantes de enumeración

Son listas de valores enteros constantes. Si no se especifican valores, comienzan en cero. En caso contrario, los valores sin especificar continúan la progresión del último valor fijado, en incrementos de uno. Los valores dentro de la misma enumeración no tienen que ser distintos. Vemos unos ejemplos en el Código fuente 12.

```
enum logica { NO, SI }; /* NO=0; SI=1 */
enum meses { ENE=1, FEB, MAR, ABR, MAY, JUN, JUL,
  AGO, SEP, OCT, NOV, DIC }; /* FEB=2, ..., DIC=12 */
```

Código fuente 12

Declaraciones

En C, todas las variables deben ser declaradas antes de ser utilizadas. La declaración se sitúa al comienzo del programa principal, de un bloque entre llaves { ... } o de la función correspondiente. Además, pueden ser inicializadas en la declaración.

```
int i = 0, lower, upper;
char c, line[1000];
```

Código fuente 13

Existen lo que se denomina variables automáticas, que se inicializan cada vez que se entra en una función o en un bloque. Si carecen de valor inicial, contienen valores basura. Las variables externas y estáticas se inicializan a cero por defecto.

Operadores Aritméticos

Los operadores aritméticos son: +, -, *, / y % (operador resto: x%y retorna el valor del resto de la división, no se puede aplicar a float o double). La división de enteros trunca la parte fraccionada.

Los operadores aritméticos se asocian de izquierda a derecha.

Operadores Relacionales y Lógicos

Por definición, los operadores relacionales y lógicos, evalúan una expresión lógica, y devuelven 0 ó 1, según sea falsa o verdadera:

- Expresión lógica verdadera = 1
- Expresión lógica falsa = 0

Los operadores relacionales son: > (mayor que), >= (mayor o igual que), < (menor que), y <= (menor o igual que). Los operadores de igualdad son: == (igual que), y != (distinto de). Los operadores lógicos son: && (*and*, y lógico), || (*or*, o lógico). El operador de negación es: !. Las expresiones se evalúan de izquierda a derecha, deteniéndose tan pronto como es conocida la falsedad o la certidumbre de la expresión. Tienen menor precedencia que los aritméticos.

Vemos unos ejemplos en el Código fuente 14.

```
a = 2; b = 3; i = a > b; /* i = 0 */
c = 0; d = 1; i = (c > 0) && (d == 1);
  /* i = 0, porque c no es mayor que cero. No se evalúa el resto */
x = 1.0; y = 0; i = (x > 0.0) || (y > 1); /* i = 1 */
```

Código fuente 14

Conversiones de tipo

Las conversiones de tipo ocurren cuando un operador tiene operandos de distinto tipo, con lo que en principio, no está claro el tipo del resultado. La regla general es que el operando de menor rango se convierte en el tipo del operando de mayor rango.

```
int i = 1;
float x = 1.2, y;
y = x + i; /* y = 2.2 */
```

Código fuente 15

Existen dos tipos de expresiones no permitidas, o en las que el compilador debe emitir advertencias (*warnings*). El primer caso corresponde al uso de números reales como contadores o índices. El segundo caso corresponde a expresiones que pierden información, por ejemplo, al asignar un número real a un entero. Para la manipulación de conversiones, se puede consultar la biblioteca estándar <ctype.h>. En los casos en que la conversión explícita de caracteres es necesaria, se utilizan los *casts*, cuya construcción es: (nombre_de_tipo) expresión. Por ejemplo, en el Código fuente 16 convertimos un entero en doble precisión.

```
int n; double x;
x = (double)n;
```

Código fuente 16

Operadores Incremento y Decremento

Existen dos tipos de operadores intrínsecos a C (y a C++). Estos son los operadores incremento (++) y decremento (--) a uno, que incrementan o decrementan la unidad cuando se aplican a enteros. Es importante para estos operadores su posición respecto a la variable:

- Como prefijo (++n, --n): incrementa / decrementa n antes de ser utilizado su valor.
- Como sufijo (n++, n--): incrementa / decrementa n después de ser utilizado su valor.

```
int i = 5, x, y;
x = ++i; /* x e i valen 6 */
y = x++; /* y vale 6; x vale 7 */
```

Código fuente 17

Expresiones y Operadores de Asignación

Los operadores de asignación se construyen de la siguiente forma, siendo op cualquiera de estos operadores: + - * / % << >> & ^ |:

```
expr1 op= expr2
```

que equivale a (los paréntesis son importantes):

```
expr1 = (expr1) op (expr2)
```

siendo expr1 y expr2 expresiones o variables. Vemos un ejemplo en el Código fuente 18.

```
int i = 1, j = 2;
j += i; /* j vale ahora 3 */
```

Código fuente 18

Expresión condicional

Existe en C un operador, denominado ternario condicional, porque su evaluación implica tres expresiones. Su construcción es:

```
expr1 ? (expr2) : (expr3)
```

Se procede del siguiente modo. Se evalúa la expresión 1; si es cierta, se evalúa la expresión 2 y éste es el valor de la expresión condicional. Si es falsa, se evalúa la expresión 3, y éste es el valor asignado. En el Código fuente 19, calculamos el máximo de dos enteros.

```
int a = 1, b = 2, z;
z = (a > b) ? a : b; /* z es igual a b */
```

Código fuente 19

Precedencia y orden de evaluación

En la ayuda de Microsoft Visual C++ se puede consultar la tabla de referencia para la precedencia y asociatividad de los operadores. En general, se recomienda el uso de paréntesis para agrupar elementos comunes, y se puede reseñar que los operadores aritméticos tienen precedencia sobre los operadores lógicos. Debemos señalar que C no especifica el orden en el que los operandos de un operador son evaluados.

```
x = f() + g(); /* ¿primero f o g? */
```

Código fuente 20

Ni el orden en el que se evalúan los argumentos de una función:

```
printf ("%d %d\n", ++n, power(2, n)); /* incorrecto */
...
++n; /* correcto */
printf ("%d %d\n", n, power(2, n));
```

Código fuente 21

Ejercicios

1. Escribir un programa en C que mediante la función sizeof(tipo) imprima por pantalla el tamaño en bytes de los tipos típicos de variables.

2. Comprobar el tamaño de las siguientes cadenas de caracteres: "Hola", y "Madrid".

3. En las siguientes enumeraciones, ¿cuánto vale dato?

```
enum { i = -2, dato };
enum { i, j = 4, dato };
```

4. ¿Qué ocurre cuando se aplican los operadores incremento y decremento a una variable tipo char? Realizar un programa de ejemplo.

5. Consultar la ayuda de Microsoft Visual C++ para ver la manipulación directa de bits en C, con sus correspondientes operadores.

6. ¿Cuál es el resultado del siguiente código?

```
int i = 3, j = 2;
j += ++i;
```

7. ¿Cuál es el resultado del siguiente código?

```
int x = 0, y = -1;
x += y++;
```

8. Consultar la ayuda de Microsoft Visual C++ para revisar la precedencia en el orden de evaluación de los distintos operadores.

9. Escribir un programa que manipule distintas variables con el operador incremento o decremento.

10. ¿Cuál es el resultado del siguiente código?

```
int x = 1, y = 1, z ;
z = (x++ > y) ? x : y;
```

11. ¿Cuál es el resultado del siguiente código?

```
int x = 1, y = 1, z ;
z = (++x > y) ? x : y;
```

Respuestas a los ejercicios

1. Escribir un programa en C que mediante la función sizeof(tipo) imprima por pantalla el tamaño en bytes de los tipos típicos de variables.

```
#include <stdio.h>

int main ()
{
  printf ("Tamaño en bytes de las variables tipicas en C\n");
  printf ("\tCaracter (char): \t\t%ld byte\n", sizeof(char));
  printf ("\tEntero (int): \t\t\t%ld bytes\n", sizeof(int));
  printf ("\tEntero grande (long int): \t%ld bytes\n", sizeof(long));
  printf ("\tNumero real (float): \t\t%ld bytes\n", sizeof(float));
  printf ("\tNumero real (double): \t\t%ld bytes\n", sizeof(double));
  printf ("\tNum. real grande (long double): %ld bytes\n", sizeof(long
double));
  return 0;
}
```

2. Comprobar el tamaño de las siguientes cadenas de caracteres: "Hola", y "Madrid".

```
#include <stdio.h>

int main ()
{
  char *s1 = "Hola";
  char *s2 = "Madrid";
```

```
  printf ("Tamaño de \"%s\": %d bytes\n", s1, strlen(s1));
  printf ("Tamaño de \"%s\": %d bytes\n", s2, strlen(s2));
  return 0;
}
```

3. En las siguientes enumeraciones, ¿cuánto vale dato?

```
enum { i = -2, dato };
```

Respuesta: dato = -1

```
   enum { i, j = 4, dato };
```

Respuesta: dato = 5

4. ¿Qué ocurre cuando se aplican los operadores incremento y decremento a una variable tipo char? Realizar un programa de ejemplo.

```
#include <stdio.h>

int main ()
{
  char a = 'a';
  printf ("Caracter %c: %d\n", a, a);
  ++a;
  printf ("Caracter %c: %d\n", a, a);
  return 0;
}
```

6. ¿Cuál es el resultado del siguiente código?

```
   int i = 3, j = 2;
   j += ++i;
```

Respuesta: i = 4; j = 6

7. ¿Cuál es el resultado del siguiente código?

```
   int x = 0, y = -1;
   x += y++;
```

Respuesta: x = -1; y = 0

9. Escribir un programa que manipule distintas variables con el operador incremento o decremento.

```
#include <stdio.h>

int main ()
```

```
{
  int x = 2, y = 7;
  int a, b;

  printf ("x vale %d - y vale %d\n", x, y);
  a = x++;
  printf ("x vale %d - a (=x++) vale %d\n", x, a);
  b = ++x;
  printf ("x vale %d - b (=++x) vale %d\n", x, b);
  a = y--;
  printf ("y vale %d - a (=y--) vale %d\n", y, a);
  b = --y;
  printf ("y vale %d - b (=--y) vale %d\n", y, b);
  return 0;
}
```

10. ¿Cuál es el resultado del siguiente código?

```
  int x = 1, y = 1, z ;
  z = (x++ > y) ? x : y;
```

Respuesta: x = 2, y = 1, z = 1

11. ¿Cuál es el resultado del siguiente código?

```
  int x = 1, y = 1, z ;
  z = (++x > y) ? x : y;
```

Respuesta: x = 2, y = 1, z = 2

3

Control de flujo en un programa

Introducción

El control del flujo de un programa es clave dentro de la programación estructurada porque especifica en que orden se realizan las operaciones y se ejecutan las sentencias o instrucciones.

Dentro de este capítulo vamos a analizar todas las estructuras existentes en C para el control del flujo. Muchas de estas estructuras se repiten tal cual en C++.

Instrucciones y Bloques

Una instrucción simple es una expresión sencilla o una llamada a una función seguida de punto y coma ";":

```
x = 'a';
```

Código fuente 22

Un bloque o instrucción compuesta son un conjunto de instrucciones simples. Se agrupan con llaves "{ ... }":

```
for (i = 0; i < 3; i++)
{
   a[i] = i + 1;
   b[i] = i;
}
```

Código fuente 23

En C el punto y coma actúa como un "finalizador" de instrucción, no como separador de instrucciones como en Pascal.

if-else

La estructura if ... else expresa formalmente la posibilidad de elección o decisión dentro de un programa. Su sintaxis es:

```
if (expresión)
   sentencias_1;
else
   sentencias_2;
```

donde, en todo lo que sigue, sentencias_1 ó 2 implican tanto instrucción sencilla como compuesta (entre llaves "{ ... }").

La forma de trabajar de esta sentencia es la siguiente. Se evalúa expresión, si ésta es cierta (valor distinto de cero), se ejecuta sentencias_1; si es falsa (la expresión tiene valor cero), se ejecuta sentencias_2. El bloque else es opcional. Debemos remarcar que else se agrupa con el if más cercano, por lo que en ifs anidados es conveniente el uso de llaves para agrupar instrucciones.

else if

Esta construcción es similar a la anterior, pero se ofrece la posibilidad de elegir entre varias alternativas. Su sintaxis es:

```
if (expresión1)
   sentencias_1;
else if (expresión2)
   sentencias_2;
else if (expresión3)
   sentencias_3;
/* ... */
else
   sentencias_n;
```

En este caso, se evalúan las expresiones en el orden el que aparecen hasta que una de ellas es cierta y se ejecuta el bloque correspondiente. Si ninguna es cierta, se ejecuta el bloque else. El bloque else es opcional también.

switch

Es un sentencia que facilita la posibilidad de elección múltiple, basándose en una expresión constante. Su sintaxis es:

```
switch (expresión)
{
   case expres_const_1:
      sentencias_1;
      break;
   case expres_const_2:
      sentencias_2;
      break;
   /* ... */
   default:
      sentencias_n;
}
```

Cada caso (case) es etiquetado con un número entero o por una expresión constante. Cuando la etiqueta corresponda con el valor de la expresión evaluada, la ejecución comenzará en ese punto. Al contrario de lo que ocurre en las sentencias if, la ejecución de cada case no es excluyente, sino que, una vez comenzada, ésta continúa en los case que siguen, salvo que cada bloque termine con la instrucción break; que sale del bucle switch. Debemos reseñar que break se puede ejecutar también dentro de otras sentencias de control de flujo, como los bucles while, for o do, para salir de ellos.

Bucles while, for y do...while

La sintaxis del bucle while es la siguiente:

```
while (expresión)
   sentencias;
```

En este caso, expresión es evaluada, y si es distinta de cero, se ejecuta el conjunto de sentencias. Al final de este conjunto, se re-evalúa de nuevo expresión. Si continúa siendo no nula, se ejecutan de nuevo sentencias hasta que expresión sea nula. El programador debe poner exquisito cuidado en no incurrir en bucles infinitos como aparece en el Código fuente 24.

```
int i = 1;
while (i) ++i;
```

Código fuente 24

La sintaxis de un bucle for es ligeramente distinta:

```
for (expr_1; expr_2; expr_3)
   sentencias;
```

expr_1 es la condición inicial del bucle, que suele contener los contadores. expr_3 contiene los incrementos o variación de los contadores. Ambas son comúnmente asignaciones o llamadas a funciones. Sin embargo, expr_2 es una expresión lógica que indica si la condición para detener el bucle es cierta. Cualquier parte se puede omitir, sin embargo, se debe mantener el punto y coma.

Si expr_2 no existe se toma permanentemente como cierta. Un bucle infinito sería el que muestra el Código fuente 25.

```
for ( ; ; )
   { /* ... */ }
```

Código fuente 25

Existe una equivalencia entre bucles while y for, veámoslo en el Código fuente 26.

```
expr_1;
while (expr_2)
{
   sentencias;
   expr_3;
}
```

Código fuente 26

Los índices tienen las siguientes propiedades:

- Pueden ser alterados dentro del bucle.
- Conservan su valor cuando el bucle termina por cualquier motivo.

Los límites pueden ser alterados dentro del bucle.

El operador coma "," separa dos expresiones dentro de las expresiones de control de un bucle for. Se evalúan de izquierda a derecha. Se suele utilizar para procesar dos índices en paralelo.

```
for (i = 0, j = 10; i <= 4; i++, j++)
   printf ("Contador j: %d\n", j);
```

Código fuente 27

La sintaxis de un bucle do-while tiene la forma:

```
do sentencias;
   while (expresión);
```

La diferencia fundamental con la sentencia while, es que en este caso, a diferencia del anterior, expresión es evaluada al final del bucle, por lo que sentencias son ejecutadas al menos una vez.

break y continue

Son las instrucciones que modifican los bucles o causan su terminación inmediata:

- break proporciona una salida inmediata de un bucle switch, for, while o do-while. En el caso de bucles anidados, recordar que se sale al bucle exterior.

- continue da lugar a que comience la siguiente iteración de un bucle for, while o do-while. En el caso de while ejecuta inmediatamente el test de expresión, en el caso for, el control pasa a los incrementos.

Goto y etiquetas

Las sentencias goto gozan de una merecida mala fama dentro de la programación estructurada. Implican un salto incondicional dentro del código que es frecuentemente difícil de rastrear. Sólo recomendamos su uso cuando sea totalmente ineludible. Un caso típico es salir de una estructura de bucles anidados, en los que serían necesarios varios break. En el código del programa, la etiqueta va seguida de dos puntos ":".

La sintaxis es:

```
/* ... */
goto etiqueta;
/* ... */
/* ... */
etiqueta:
/* ... */
```

En este tema hemos analizado en profundidad la sintaxis de los bucles y otras sentencias en C. En lo que sigue de curso se reusarán estos conceptos con frecuencia.

Ejercicios

1. Escribir un programa en C que lea un entero en pantalla en el rango 0-10. Para cualquier número fuera de ese rango imprimir un mensaje de error. (Nota: para leer enteros en pantalla usar scanf ("%d", &i);, siendo i un número entero). Usar sentencias if.

2. Un año es bisiesto cuando es múltiplo de 400, ó cuando es divisible entre 4, y no es divisible entre 100. Escribir un programa en C que pida al usuario un año, y le informe de si es bisiesto o no (No considerar años negativos. Recordar que no existe el año cero en el calendario cristiano).

3. Crear un programa que lea una opción por pantalla de 4 opciones ofrecidas, y le informe al usuario de cuál ha elegido.

4. Escribir un programa en C que cuente los caracteres de un nombre introducido por el teclado mediante un bucle while. (Nota: para leer una cadena de caracteres, usar scanf ("%s", c);, siendo c un puntero a una cadena de caracteres).

5. Repetir el ejercicio 3, de forma que el programa reimprima la lista de opciones cuando el usuario introduzca una opción errónea mediante goto.

6. Crear un bucle for de 1 a 10, que imprima sólo los números impares mediante una sentencia continue.

7. Repetir el ejercicio anterior cambiando el incremento en el bucle.

8. Crear un programa que cuente el número de vocales de una palabra con una sentencia switch.

9. Modificar el programa anterior para calcular el número de consonantes.

10. Crear un programa para invertir el orden de caracteres en una palabra. (Nota: usa strlen(s);).

Respuestas a los ejercicios

1. Escribir un programa en C que lea un entero en pantalla en el rango 0-10. Para cualquier número fuera de ese rango imprimir un mensaje de error. (Nota: para leer enteros en pantalla usar scanf ("%d", &i);, siendo i un número entero). Usar sentencias if.

```
#include <stdio.h>

int main()
{
  int d;
  printf ("Introduce un entero en el rango 0-10: ");
  scanf ("%d", &d);
  if (d > 10) printf ("Número demasiado grande\n");
  else if (d < 0) printf ("Número demasiado pequeño\n");
  else printf ("Número apropiado\n");
  return 0;
}
```

2. Un año es bisiesto cuando es múltiplo de 400, ó cuando es divisible entre 4, y no es divisible entre 100. Escribir un programa en C que pida al usuario un año, y le informe de si es bisiesto o no (No considerar años negativos. Recordar que no existe el año cero en el calendario cristiano).

```
#include <stdio.h>

int main ()
{
  int year;

  printf ("Introduce un año: ");
  if ( (scanf ("%d", &year)) != 1 )
  {
    perror ("Error en la lectura del dígito");
    exit (1);
  }
  else if (year <= 0)
  {
    perror ("Año no válido");
    exit (2);
  }
  if ( ( year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
    printf ("%d es un año bisiesto\n", year);
  else
    printf ("%d no es un año bisiesto\n", year);
  return 0;
}
```

3. Crear un programa que lea una opción por pantalla de 4 opciones ofrecidas, y le informe al usuario de cuál ha elegido.

```
#include <stdio.h>

int main ()
{
  char option;
  printf ("Introduce tu opción:\n"
          "1. Opción n°1\n"
          "2. Opción n°2\n"
          "3. Opción n°3\n"
          "4. Opción n°4\n");
  if ( (option = getchar() ) == EOF )
  {
    perror ("Error en la lectura de la opción\n");
    exit (1);
  }
  switch (option)
  {
    case '1':
      printf ("Elegiste la opción n°1\n");
      break;
    case '2':
      printf ("Elegiste la opción n°2\n");
      break;
    case '3':
      printf ("Elegiste la opción n°3\n");
      break;
    case '4':
      printf ("Elegiste la opción n°4\n");
      break;
    default:
      perror ("Error en la lectura de la opción\n");
      exit (1);
  }
  return 0;
}
```

4. Escribir un programa en C que cuente los caracteres de un nombre introducido por el teclado mediante un bucle while. (Nota: para leer una cadena de caracteres, usar scanf ("%s", c);, siendo c un puntero a una cadena de caracteres).

```
#include <stdio.h>

int main()
{
  char s[60];
  int i = 0;
  printf ("Introduce tu nombre completo sin espacios:\n");
  if ( (scanf ("%s", s)) != 1 )
  {
    perror ("Error en la lectura de la cadena");
    exit (1);
  }
  while (s[i] != '\0') ++i;
  printf ("Tu nombre es ... \'%s\'\n"
          "y tiene una longitud: %d\n", s, i);
  return 0;
}
```

5. Repetir el ejercicio 3, de forma que el programa reimprima la lista de opciones cuando el usuario introduzca una opción errónea mediante goto.

```
#include <stdio.h>

int main ()
{
  char option;
  inicio: /* Etiqueta del goto */
  printf ("Introduce tu opción:\n"
          "1. Opción n°1\n"
          "2. Opción n°2\n"
          "3. Opción n°3\n"
          "4. Opción n°4\n");
  if ( (option = getchar() ) == EOF )
    goto inicio;
  switch (option)
  {
    case '1':
      printf ("Elegiste la opción n°1\n");
      break;
    case '2':
      printf ("Elegiste la opción n°2\n");
      break;
    case '3':
      printf ("Elegiste la opción n°3\n");
      break;
    case '4':
      printf ("Elegiste la opción n°4\n");
      break;
    default:
      goto inicio;
  }
  return 0;
}
```

6. Crear un bucle for de 1 a 10, que imprima sólo los números impares mediante una sentencia continue.

```
#include <stdio.h>

int main()
{
  int i;

  for (i = 1; i <= 10; i++)
  {
    if (i % 2 == 0) continue;
    printf ("Bucle: índice %d\n", i);
  }
  return 0;
}
```

7. Repetir el ejercicio anterior cambiando el incremento en el bucle.

```
#include <stdio.h>

int main()
{
  int i;

  for (i = 1; i <= 10; i+=2)
  {
    printf ("Bucle: índice %d\n", i);
  }
  return 0;
}
```

8. Crear un programa que cuente el número de vocales de una palabra con una sentencia switch.

```
#include <stdio.h>

int main()
{
  char s[40];
  int i = 0, nv = 0;
  printf ("Introduce una palabra:\n");
  if ( (scanf ("%s", s)) != 1 )
  {
    perror ("Error en la lectura de la cadena");
    exit (1);
  }
  while (s[i] != '\0')
  {
    switch (s[i])
    {
      case 'a': case 'e': case 'i': case 'o': case 'u':
      case 'A': case 'E': case 'I': case 'O': case 'U':
        ++nv;
    }
    ++i;
  }
  printf ("La palabra es ... \'%s\'\n"
          "y tiene %d vocales.\n", s, nv);
  return 0;
}
```

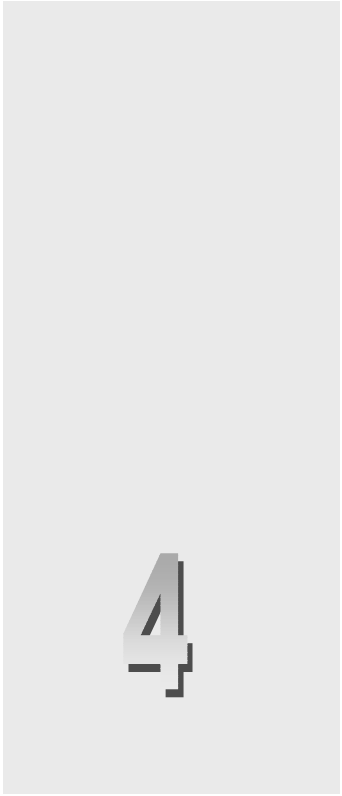
9. Modificar el programa anterior para calcular el número de consonantes.

```
#include <stdio.h>

int main()
{
  char s[40];
  int i = 0, nc = 0;
  printf ("Introduce una palabra:\n");
  if ( (scanf ("%s", s)) != 1 )
  {
    perror ("Error en la lectura de la cadena");
    exit (1);
  }
  while (s[i] != '\0')
  {
    switch (s[i])
```

```
    {
      case 'a': case 'e': case 'i': case 'o': case 'u':
      case 'A': case 'E': case 'I': case 'O': case 'U': break;
      default: ++nc;
    }
    ++i;
  }
  printf ("La palabra es ... \'%s\'\n"
          "y tiene %d consonantes.\n", s, nc);
  return 0;
}
```

10. Crear un programa para invertir el orden de caracteres en una palabra. (Nota: usa strlen(s);).

```
#include <stdio.h>

int main()
{
  char s[40], c;
  int i, j;
  printf ("Introduce una palabra:\n");
  if ( (scanf ("%s", s)) != 1 )
  {
    perror ("Error en la lectura de la cadena");
    exit (1);
  }
  printf ("La palabra es ... \'%s\'\n", s);
  for (i = 0, j = strlen(s) - 1; i < j; i++, j--)
  {
    c = s[i];
    s[i] = s[j];
    s[j] = c;
  }
  printf ("La palabra invertida es ... \'%s\'\n", s);
  return 0;
}
```



Funciones y estructura de los programas

Introducción

En este capítulo vamos a avanzar más en la descripción de la estructura de los programas escritos en C. Como es conocido por el alumno, dentro de cada lenguaje de programación existen una serie de elementos que facilitan la descomposición de tareas en otras más sencillas. Para ello, se dispone de subrutinas, también llamadas funciones. Además, la utilización de funciones va un paso más allá, permitiendo la creación de bibliotecas de funciones que pueden ser reutilizadas por el propio programador, o por otros programadores.

C ha sido diseñado para un uso eficiente de las funciones porque:

- Un programa puede residir en varios ficheros fuente;
- Los ficheros fuente pueden ser compilados por separado; y
- Estos ficheros pueden ser cargados conjuntamente, mediante la utilización de bibliotecas.

Una mejora importante de ANSI C es que se declaran los tipos de los argumentos de las funciones (lo que permite su verificación en tiempo de compilación), cosa que no ocurría con versiones anteriores del lenguaje. También se han clarificado las reglas de alcance de las funciones y se ha mejorado el preprocesador.

Fundamentos de funciones

La sintaxis general de una función es la siguiente

```
tipo_retorno nombre_función (declaración_de_argumentos)
{
   declaraciones;
   sentencias;
   return (expresión);
}
```

Siendo tipo_retorno, el tipo de variable que devuelve la función a la función que la llama. La declaración de argumentos no era obligatoria en algunos compiladores (aunque si es recomendable y es una buena práctica de programación) por compatibilidad con anteriores versiones de C. Esa situación ha ido desapareciendo.

Una función "minimalista" que no hiciera absolutamente nada sería la mostrada en el Código fuente 28.

```
void no_hago_nada( )
{
}
```

Código fuente 28

Return es el mecanismo que utilizan las funciones para devolver el control al programa principal, o a otra función que la llamó. El valor de expresión es convertido al tipo de retorno especificado. Los paréntesis en expresión son optativos.

El tipo por defecto de una función es entero (int). Si deseamos que una función no devuelva ningún valor, se usa el tipo void. Son válidos todos los tipos de variable estándar y los que se puedan definir con estructuras: float, double, char, long, etc.

En el Código fuente 29 vemos un ejemplo simple. Tratemos de construir la función de biblioteca atof, que dada una cadena de caracteres que representa un número, devuelve el número de coma flotante equivalente. Para ello, usamos la función isdigit, de la biblioteca estándar <ctype.h>, que indica si un carácter es un dígito o no. El código es simple, y recomendamos estudiarlo en detalle:

```
/* Ejemplo 1 */
#include <ctype.h>

double atof (char s[]) /* Convierte cadena a doble */
{
  double val, potencia;
  int i, signo;
  for (i = 0; isspace(s[i]); i++); /* Salta espacios en
                                      blanco iniciales */
  signo = (s[i] == '-') ? -1 : 1;
  if (s[i] == '+' || s[i] == '-') i++;
  for (val = 0.0; isdigit(s[i]); i++)
    val = 10.0 * val + (s[i] - '0');
  if (s[i] = '.') i++;
  for (potencia = 1.0; isdigit(s[i]); i++)
  {
```

```
    val = 10.0 * val + (s[i] - '0');
    potencia *= 10.0;
  }
  return signo * val / potencia;
}
```

Código fuente 29

Variables externas e internas

En C existen dos tipos de variables, externas e internas con respecto a una función. Las variables internas describen argumentos y variables definidos dentro de las funciones. Las variables externas están definidas fuera de cualquier función y disponibles potencialmente para muchas funciones. Las propias funciones son externas, puesto que una función no puede ser definida dentro de otra función.

Por defecto, todas las referencias a variables y funciones externas, incluso de funciones compiladas por separado, son las mismas. (Equivalen a los bloques COMMON de Fortran). Por ello, son el mecanismo típico de comunicación entre funciones.

Reglas de alcance

Un tema muy importante en C es dónde y cómo interaccionan las variables de un programa y sus funciones. Si consideramos que los componentes de un programa en C no tienen porqué estar en el mismo fichero:

- ¿Cómo se escriben las declaraciones adecuadamente para la compilación?
- ¿Cómo se disponen las declaraciones para una conexión adecuada durante el enlazado del programa (link)?
- ¿Cómo se organizan las declaraciones para que exista sólo una copia?
- ¿Cómo se inicializan las variables externas?

Estos elementos no introducen el concepto de alcance. ¿Qué es el alcance de un nombre (función o variable)? Es la parte del programa dentro de la cual ese nombre puede ser utilizado. Se denomina variable automática a aquella definida al comienzo de una función, y su alcance es la propia función. Debemos destacar que variables locales del mismo nombre en distintas funciones no tienen ninguna relación.

Se denomina variable externa o función a aquella cuyo alcance está definido desde el punto de declaración hasta el final del fichero. Veamos el Código fuente 30.

```
int main( ) { ... }
double x;
int funcion1 ( ) { ... }
int funcion2 ( ) { ... }
```

Código fuente 30

La variable x es externa a funcion1 y funcion2, y puede ser usada dentro de ellas, en cambio main no puede utilizarla. Si una variable externa tiene que ser utilizada antes de ser definida, o está definida en otro fichero fuente, es obligatorio el uso de extern.

```
extern double x;
```

Código fuente 31

Es muy importante para un programador en C distinguir entre declaración y definición de una variable externa:

- Declaración: anuncia las propiedades de una variable (su tipo, principalmente).
- Definición: además origina la reserva de un espacio de almacenamiento. La definición es única, pero la declaración es necesaria en todos los puntos donde se vaya a utilizar la variable.

Por ejemplo:

- fichero fuente1 (definición).

```
int sp = 0;
double val[MAXVAL];
```

Código fuente 32

- fichero fuente2 (declaración).

```
extern int sp;
extern double val[];
```

Código fuente 33

Archivos de encabezado

A lo largo de los distintos ejemplos del curso, nos hemos familiarizado con el concepto de biblioteca estándar y como acceder a ella. Hemos introducido elementos de la biblioteca de entrada/salida, <stdio.h>. Pero en C existen más elementos de la biblioteca estándar, e incluso el usuario puede crear su propia biblioteca de funciones, y todos los accesos se declaran a través de archivos de encabezado (*header files*). Estos archivos se incluyen en el archivo fuente en tiempo de compilación mediante la instrucción del preprocesador #include.

```
#include <archivo_de_biblioteca_estándar>
#include "nombre_archivo_de_usuario"
```

Los archivos contienen:

- Encabezados de biblioteca estándar con funciones a utilizar.
- Definiciones de variables externas y funciones.

Variables estáticas

Las variables estáticas se declaran con la palabra clave static. Para una variable externa, el uso de static limita el alcance del objeto al resto del archivo fuente en uso, y no a otros archivos que pudieran tener acceso a las funciones definidas en éste.

```
static int buffer = 0;
funcion1 ( ) { ... }
funcion2 ( ) { ... }
```

Código fuente 34

La variable buffer es accesible a funcion1 y funcion2, pero ninguna otra función podrá acceder a ella, y no entrará en conflicto con otra variable del mismo nombre en otro archivo fuente del programa.

Static es también aplicable a una función, que sería sólo visible a funciones contenidas dentro del mismo fichero. Para una variable interna o local, el uso de static haría que la variable siga siendo local a la función, pero mantendría su valor en llamadas sucesivas.

```
funcion1( )
{
   static int i = 0;
   /* punto_1 */
   ...
   i = 1;
}
```

Código fuente 35

La segunda vez que se llame a funcion1, en el punto_1, la variable i valdrá 1, no cero.

Variables de registro

Una declaración register advierte al compilador que la variable en cuestión va a ser utilizada muy frecuentemente, y le aconseja situarla en "registros de máquina", de acceso más rápido. El compilador puede ignorar el "consejo".

```
register int x;
register char c;
```

Código fuente 36

Esta declaración depende mucho del hardware y del sistema operativo.

Estructura de bloques

En C las funciones no pueden ser definidas dentro de otras funciones. Sin embargo, las variables pueden ser definidas dentro de los bloques de cualquier sentencia compuesta (dentro de las llaves { }).

```
if (n > 0)
{
   int i;
   for (i = 0; i < n; i++) { ... }
}
```

Código fuente 37

La variable i existe en el interior del bloque if y no está relacionada con cualquier otra variable i de la función.

Inicialización

La inicialización de las variables es un tema muy importante en el desarrollo de software. Si no hay una inicialización explícita, el estándar indica que:

- Las variables externas y estáticas tienen "garantizado" el valor cero.
- Las variables automáticas y de registro tienen valores "basura".

En C, se puede definir un inicializador para las variables:

- Variables externas y estáticas: siempre debe ser una expresión constante, y se hace una única vez, cuando el programa comienza a ejecutarse.
- Variables automáticas y de registro: admiten una expresión implicando valores definidos previamente. Se definen cada vez que se entra en la función o en el bloque.

Un array no necesita en su declaración el tamaño, el compilador calculará el tamaño del array con los valores iniciales.

```
int dias[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Inicializa dias como un array de doce elementos. Si el número de valores iniciales es menor que la dimensión del array, el resto de los elementos serán cero para variables externas, estáticas y automáticas. No hay modo de especificar la repetición de un valor inicial. Es un error especificar mayor número de valores que la dimensión del array.

Recursión

Las funciones en C pueden ser usadas recursivamente, es decir, una función puede llamarse a sí misma directa o indirectamente.

El preprocesador de C

El preprocesador de C consiste en una serie de facilidades incluidas en el lenguaje, para automatizar algunos elementos en el proceso de compilación. Estas facilidades tienen el nombre de directivas, y van precedidas del símbolo #.

La primera facilidad, ya mencionada con anterioridad es la inclusión de archivos en tiempo de compilación a través de la directiva #include:

```
#include "nombre_de_archivo"
#include <nombre_de_archivo>
```

Código fuente 38

La línea de código es sustituida por el contenido del archivo, sin ninguna modificación. Además del uso típico para incluir declaraciones de funciones o variables, esta directiva ofrece otros usos. Debemos recordar que si por algún motivo cambia el contenido del archivo, el programa debe ser compilado de nuevo.

La siguiente directiva está relacionada con la sustitución de macros:

```
#define nombre texto_para_reemplazar
```

Cada ocurrencia de nombre dentro del archivo fuente es sustituida por texto_para_reemplazar. Si texto_para_reemplazar es muy largo, se puede continuar en la siguiente línea con un '\' al final. El alcance de #define es desde el punto de definición hasta el final del archivo fuente. La sustitución no tendría lugar si apareciera nombre entre comillas ("nombre").

Si el nombre de un parámetro está precedido de '#' en el texto de reemplazo, la combinación se expandirá en una cadena de caracteres entrecomillada con el argumento. Veamos por ejemplo el Código fuente 39.

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

Código fuente 39

Entonces,

```
dprint (x/y);
```

Código fuente 40

equivale a:

```
printf ("x/y" " = %g\n", x/y);
```

Código fuente 41

Una macro o nombre, se puede des-definir:

```
#undef nombre
```

Este mecanismo se utiliza para asegurar que una rutina es realmente una función y no una macro. El operador ## proporciona una forma de concatenar argumentos:

```
#define pegar(anterior, posterior) anterior##posterior
```

La inclusión condicional controla la inclusión de sentencias en la compilación mediante las instrucciones que aparecen en el Código fuente 42.

```
#if defined (nombre1)
   sentencias y macros
#elif defined (nombre2)
   sentencias y macros
#else
   sentencias y macros
#endif
```

Código fuente 42

Cuya sintaxis recuerda a los bucles if. También existe la comparación con igualdad: == y la negación: !. Una utilización muy frecuente de las directivas es la comprobación de la existencia de variables o macros.

```
#if !defined (HDR) /* o #ifndef HDR */
   #define HDR
#endif
```

Código fuente 43

Ejercicios

1. Escribe un código para una función int isdigit (char c), que devuelve 1 ó 0 según el carácter c sea un dígito o no.

2. ¿Qué significa el fragmento de código siguiente (ver ejemplo 1)?

```
val = 10.0 * val + (s[i] - '0');
```

3. Escribir un programa que pruebe la función atof.

4. El factorial de un número entero positivo se define como el producto de sí mismo por todos los enteros positivos menores que él. Crear una función factorial recursiva para calcular el factorial de un entero positivo introducido por el teclado.

5. Modificar el ejemplo anterior para comprobar que el usuario introduce efectivamente un número entero positivo.

6. Crear un programa en C con tres variables i: una variable en el programa principal main, una variable en una función nada con distinto valor, y una variable estática dentro de un bucle for. Comprobar las reglas de alcance.

7. Crear un programa para comprobar la concatenación de argumentos con la directiva ##.

8. Crear una directiva para definir una macro que calcule el máximo de dos números.

9. Vamos a realizar un ejercicio simple. Supongamos que queremos realizar una versión simplificada del programa grep de Unix, que extrae de un fichero un determinado grupo de líneas que contienen los caracteres del patrón de comparación. El algoritmo, escrito en pseudo-código, tiene una estructura muy simple.

```
while (EXISTA_OTRA_LÍNEA)
   if (LA_LÍNEA_CONTIENE_EL_PATRÓN)
      IMPRÍMELA;
```

   Primero crear una función strindex que busca una cadena de caracteres t[] en otra s[]. Como valor devuelve el índice de s[] donde comienza t[], o -1 si no existe la subcadena. A continuación crear una función getline que lee cada línea de la entrada (teclado o archivo).

10. Realizar un ejemplo para las funciones del ejercicio anterior. Aplicarlo a distintos archivos usando el redireccionador de entrada desde el prompt de MS-DOS: ex10.exe < nombre-archivo

Respuestas a los ejercicios

1. Escribe un código para una función int isdigit (char c), que devuelve 1 ó 0 según el carácter c sea un dígito o no.

```
int isdigit (char c)
{
  switch (c)
  {
    case '0': case '1': case '2': case '3':
    case '4': case '5': case '6': case '7':
    case '8': case '9':
      return 1;
  }
  return 0;
}
```

2. ¿Qué significa el fragmento de código siguiente (ver ejemplo 1)?

```
val = 10.0 * val + (s[i] - '0');
```

   Respuesta:

   Si el carácter s[i] es un dígito, implica que val es multiplicado por 10, y se le suma un entero que equivale al propio dígito (como resultado de la diferencia del código ASCII de '0').

3. Escribir un programa que pruebe la función atof.

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

double atof (char s[]) /* Convierte cadena a doble */
{
  double val, potencia;
  int i, signo;
  for (i = 0; isspace(s[i]); i++); /* Salta espacios en
                                      blanco iniciales */
  signo = (s[i] == '-') ? -1 : 1;
  if (s[i] == '+' || s[i] == '-') i++;
  for (val = 0.0; isdigit(s[i]); i++)
    val = 10.0 * val + (s[i] - '0');
  if (s[i] = '.') i++;
  for (potencia = 1.0; isdigit(s[i]); i++)
  {
    val = 10.0 * val + (s[i] - '0');
    potencia *= 10.0;
  }
  return signo * val / potencia;
}

int main()
{
  char s[50];
  double x;
  printf ("Introduce un dígito real:\n");
  if ( (scanf ("%s", s)) != 1 )
  {
    perror ("Error en la lectura de la cadena");
    exit (1);
  }
  x = atof (s);
  printf ("El dígito es %f\n", x);
  return 0;
}
```

4. El factorial de un número entero positivo se define como el producto de sí mismo por todos los enteros positivos menores que él. Crear una función factorial recursiva para calcular el factorial de un entero positivo introducido por el teclado.

```
#include <stdio.h>
long int factorial (long int i)
{
  return (i > 1) ? i * factorial (i - 1) : 1;
}
int main()
{
  long int i;
  printf ("Introduce un número entero: ");
  if ( (scanf ("%ld", &i)) != 1 )
  {
    perror ("Error en la lectura del número");
    exit (1);
  }
  printf ("El factorial de %ld es %ld.\n", i, factorial (i));
  return 0;
}
```

5. Modificar el ejemplo anterior para comprobar que el usuario introduce efectivamente un número entero positivo.

```
#include <stdio.h>

long int factorial (long int i)
{
  return (i > 1) ? i * factorial (i - 1) : 1;
}

int main()
{
  long int i;
  printf ("Introduce un número entero: ");
  if ( (scanf ("%ld", &i)) != 1 || i < 0)
  {
    perror ("Error en la lectura del número");
    exit (1);
  }
  printf ("El factorial de %ld es %ld.\n", i, factorial (i));
  return 0;
}
```

6. Crear un programa en C con tres variables i: una variable en el programa principal main, una variable en una función nada con distinto valor, y una variable estática dentro de un bucle for. Comprobar las reglas de alcance.

```
#include <stdio.h>

void nada() /*      Función void nada */
{
  int i = 20;
  printf ("función nada: i = %d\n", i);
}

int main()
{
  int i = 0, j; /* Variable del programa principal */

  printf ("main: i = %d\n", i);
  nada();
  for (j = 1; j < 6; j++)
  {
    static int i = 1;
    printf ("bucle for: i = %d\n", i++);
  }
  printf ("main: i = %d\n", i);
  return 0;
}
```

7. Crear un programa para comprobar la concatenación de argumentos con la directiva ##.

```
#include <stdio.h>
#define pegar(anterior, posterior) anterior##posterior

int main()
```

```
{
  char *cd = "Hola Mundo!";

  printf ("%s\n", pegar(c, d));
  /* equivale a: printf ("%s\n", cd); */
  return 0;
}
```

8. Crear una directiva para definir una macro que calcule el máximo de dos números.

```
#include <stdio.h>
#define max(A,B) ((A) > (B) ? (A) : (B))

int main()
{
  int i = 3, j = 4;
  float x = 1.12, y = 1.11;

  printf ("Máximo enteros (i,j): %d\n", max(i, j));
  printf ("Máximo reales (x,y): %f\n", max(x, y));
  return 0;
}
```

9. Vamos a realizar un ejercicio simple. Supongamos que queremos realizar una versión simplificada del programa grep de Unix, que extrae de un fichero un determinado grupo de líneas que contienen los caracteres del patrón de comparación. El algoritmo, escrito en pseudo-código, tiene una estructura muy simple.

```
   while (EXISTA_OTRA_LÍNEA)
      if (LA_LÍNEA_CONTIENE_EL_PATRÓN)
         IMPRÍMELA;
```

Primero crear una función strindex que busca una cadena de caracteres t[ ] en otra s[ ]. Como valor devuelve el índice de s[ ] donde comienza t[ ], o -1 si no existe la subcadena. A continuación crear una función getline que lee cada línea de la entrada (teclado o archivo).

```
#include <stdio.h>
#define MAXLINEA 1000 /* Tamaño máximo de línea */

char cad[] = "if";

int strindex (char s[], char t[])
{
  int i, j, k;
  for (i = 0; s[i] != '\0'; i++)
  {
    for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++);
    if (k > 0 && t[k] == '\0') return i;
  }
  return -1;
}

int getline (char s[], int lim)
{
  int c, i = 0;
```

```
  while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
    s[i++] = c;
  if (c == '\n') s[i++] = c;
  s[i] = '\0';
  return i;
}

int main()
{
  char linea[MAXLINEA];
  int encontrados = 0;
  while (getline (linea, MAXLINEA) > 0)
    if (strindex (linea, cad) >= 0)
    {
      encontrados++;
      printf ("%d:%s", encontrados, linea);
    }
  return encontrados;
}
```

10. Realizar un ejemplo para las funciones del ejercicio anterior. Aplicarlo a distintos archivos usando el redireccionador de entrada desde el prompt de MS-DOS: ex10.exe < nombre-archivo

```
-El código de este ejercicio es idéntico al anterior. Veámoslo de nuevo-
#include <stdio.h>
#define MAXLINEA 1000 /* Tamaño máximo de línea */
char cad[] = "if";
int strindex (char s[], char t[])
{
  int i, j, k;
  for (i = 0; s[i] != '\0'; i++)
  {
    for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++);
    if (k > 0 && t[k] == '\0') return i;
  }
  return -1;
}
int getline (char s[], int lim)
{
  int c, i = 0;
  while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
    s[i++] = c;
  if (c == '\n') s[i++] = c;
  s[i] = '\0';
  return i;
}
int main()
{
  char linea[MAXLINEA];
  int encontrados = 0;
  while (getline (linea, MAXLINEA) > 0)
    if (strindex (linea, cad) >= 0)
    {
      encontrados++;
      printf ("%d:%s", encontrados, linea);
    }
  return encontrados;
}
```

5

Punteros y arrays

Introducción

Los punteros son una de las características más importantes de C. No sólo eso, podemos decir que es una de las características únicas de C. Existe gran controversia entre programadores en C y otros lenguajes sobre el uso de los punteros. Si bien su abuso conduce a programas crípticos, se ha de reconocer que muchas veces conduce a una gestión eficiente de los recursos. Un **puntero** no es más que una variable que contiene la dirección de memoria de otra variable. Sus características son:

- Crean código más compacto y eficiente.
- En muchas ocasiones son el único camino para un cálculo, o para manipular recursos del ordenador.
- Pueden ser utilizados para crear código "imposible de comprender".

En el ANSI C se contempla la creación del puntero nulo (tipo void *) como el puntero adecuado a un puntero genérico.

Punteros y direcciones

Los punteros responden a una necesidad en la manipulación de recursos del ordenador. En particular, si consideramos la organización de la memoria de un ordenador, nos encontramos con un conjunto de "celdas" de memoria numeradas consecutivamente, o direccionadas, que pueden ser manipuladas

individualmente o en grupos contiguos. Un puntero es un grupo de celdas (frecuentemente dos o cuatro) que pueden contener una dirección.

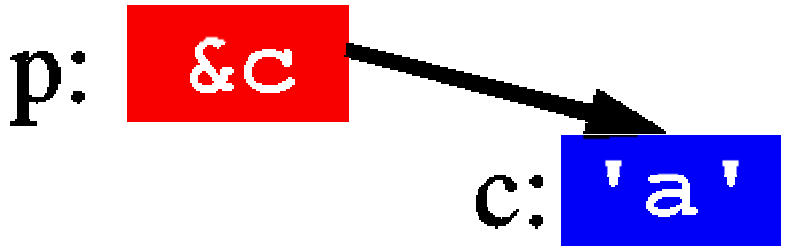


Figura 1

Por ejemplo, si consideramos que c es un char, y p un puntero a c, la figura anterior representa gráficamente el significado de los punteros. El operador unitario & proporciona la dirección de un objeto. Así:

```
p = &c;
```

Asigna la dirección de la variable c a la variable p. Existen restricciones al uso del operador &. Sólo puede ser aplicado a objetos en memoria, no puede ser aplicado a:

- expresiones;
- constantes; o
- variables de registro.

El operador unitario * (operador de indirección o dereferencial), cuando se aplica a un puntero, accede al objeto al cual apunta éste.

```
int x = 1, y = 2; int *ip; /* ip es un puntero a entero */
ip = &x; /* ip apunta ahora a x */
y = *ip; /* y es ahora 1 */
*ip = 0; /* x es ahora 0 */
```

Código fuente 44

Un puntero está restringido a un tipo específico de dato (Excepto el puntero void, que puede apuntar a cualquier tipo). Además, los punteros pueden usarse en cualquier contexto en el que se puede usar la variable a la que apunta. Veamos un ejemplo en el Código fuente 45.

```
y = *ip + 1; /* incrementa la variable apuntada por ip en 1, y
                lo asigna a y */
```

Código fuente 45

Debe prestarse especial cuidado a la manipulación de punteros y los operadores incremento y decremento.

```
   ++*ip; /* incrementa en 1 la variable apuntada por ip */
   (*ip)++; /* incrementa en 1 la variable apuntada por ip */
   *ip++; /* incrementa en 1 el puntero, y no la variable apuntada */
```

Código fuente 46

Punteros y argumentos de funciones

Es importante recordar que C pasa los argumentos a las funciones por valor. Es decir, las variables pasan su valor a la función, y cualquier modificación que realice ésta no se refleja en el programa principal. Veamos el ejemplo que muestra el Código fuente 47.

```
/* Ejemplo 1 */
#include <stdio.h>

void swap (int, int);

int main ()
{
  int a = 28, b = 20;
  printf ("main (inicio): a = %d - b = %d\n", a, b);
  swap (a, b);
  printf ("main (fin): a = %d - b = %d\n", a, b);
  return 0;
}

void swap (int a, int b)
{
  int temp;
  printf ("función swap (inicio): a = %d - b = %d\n", a, b);
  temp = a;
  a = b;
  b = temp;
  printf ("función swap (intercambiadas): a = %d - b = %d\n", a, b);
}
```

Código fuente 47

En él hemos definido una función swap que intercambia los valores de dos enteros. Después de los comentarios realizados, es evidente que la función swap tal como está definida es inoperante. Para ello, hay que redefinir swap de forma que manipule punteros. De este modo, los cambios realizados en los argumentos se reflejan en el programa principal.

```
/* Ejemplo 2 */
#include <stdio.h>

void swap (int*, int*);

int main ()
{
  int a = 28, b = 20;
  printf ("main (inicio): a = %d - b = %d\n", a, b);
  swap (&a, &b);
  printf ("main (fin): a = %d - b = %d\n", a, b);
```

```
  return 0;
}

void swap (int *a, int *b)
{
  int temp;
  printf ("función swap (inicio): a = %d - b = %d\n", *a, *b);
  temp = *a;
  *a = *b;
  *b = temp;
  printf ("función swap (intercambiadas): a = %d - b = %d\n", *a, *b);
}
```

Código fuente 48

Punteros y arrays

En C hay una relación muy estrecha entre punteros y arrays. Recordemos que los arrays son un conjunto de elementos del mismo tipo agrupados, y accesibles a través de un índice que comienza en cero. Cualquier operación con arrays puede ser realizada con punteros.

```
int a[7]; /* array de enteros de 7 elementos: a[0],a[1], ... , a[6] */
int *pa; /* puntero a entero */
pa = &a[0]; /* pa apunta al elemento cero de a */
pa = a; /* equivale a la asignación anterior */
```

Código fuente 49

Esta relación va más allá, transmitiéndose a los elementos del array.

```
pa + i /* dirección de a[i] */
*(pa + i) /* contenido de a[i] */
*(pa + i) /* equivale a pa[i] */
*(a + i) /* contenido de a[i] */
```

Código fuente 50

Muchas de las relaciones anteriores no son recíprocas. Por ejemplo, son ilegales las que vemos en el Código fuente 51. Pero son legales, las del Código fuente 52, ya que un puntero es una variable.

```
a = pa; a++;
```

Código fuente 51

```
pa = a; pa++;
```

Código fuente 52

Aritmética de direcciones

Uno de los problemas más criticados en el diseño de programas en C es permitir ciertas operaciones aritméticas con punteros. Si p es un puntero a un array: p++ apunta al siguiente elemento del array y p+=i apunta al elemento i posiciones más allá de la actual. Los problemas surgen porque C permite situarnos más allá de los elementos del array, con lo que la última construcción sería legal, pero su contenido es basura. De hecho, no es posible detectar en tiempo de compilación este tipo de errores.

Se debe recordar que punteros y enteros no son intercambiables, con la excepción de la constante cero (o NULL, definida en <stdio.h>), que puede ser asignada a un puntero, o un puntero puede ser comparado con la constante cero. Si p y q son **punteros a miembros del mismo array**, las relaciones: ==, !=, <, >, <=, >=, funcionan adecuadamente. Para punteros de distinto array, estas operaciones no están definidas. La sustracción de punteros del mismo array también es válida. La adición, división o multiplicación no son válidas.

Punteros a caracteres y funciones

Las cadenas de caracteres son arrays de caracteres terminados con el carácter nulo '\0'. Las cadenas de caracteres se definen de dos modos:

- Como array de caracteres.

```
char amessage[] = "ya es hora";
```

Código fuente 53

- Como puntero a cadena de caracteres.

```
char *pmessage = "ya es hora";
```

Código fuente 54

Observe que por su propia naturaleza, las cadenas de caracteres nunca pasan por valor como argumentos de funciones, sino por referencia.

Arrays de punteros; punteros a punteros

Como los punteros son variables, se pueden almacenar en arrays, con lo que podemos crear arrays de punteros.

```
int *p[4];
int i = 1, j = 2;
p[0] = &i;
p[1] = &j;
```

Código fuente 55

Estos elementos son de gran utilidad, especialmente para almacenar cadenas de distinta longitud cuya gestión originaría problemas en el caso de tener que almacenarlas en arrays de caracteres. No es necesario dar explícitamente la dimensión del array de punteros, pudiendo generarse dinámicamente con una inicialización con { ... }:

```
char *diasem[] = { "lunes", "martes", ..., "domingo" };
```

Arrays multidimensionales

C facilita también la creación de arrays multidimensionales. La forma de explicitar un array bidimensional es con la construcción a[i][j]. Aunque de muy raro uso en la práctica pueden existir arrays de dimensión mayor que 2. Se pueden inicializar también con { ... }, recordando que debe existir un par { ... } por fila.

```
long ventas[2][3] = { { 1000, 1400, 1300 }, { 3000, 3200, 1400 } };
```

Código fuente 56

Argumentos en la línea de comandos

Cualquier entorno que soporta C permite pasar argumentos desde la línea de comandos al programa main(). Así desde el prompt de MS-DOS podemos pasar a nuestro programa valores de ciertas variables, nombres de archivos, etc. Existen dos argumentos:

- argc (**argument count**): número de argumentos de la línea de comandos. Cuando argc == 1, no hay argumentos en la línea de comandos.

- argv[] (**argument vector**): puntero a un array de cadena de caracteres que contiene los argumentos (uno por cadena). argv[0] es el nombre por el que se ha invocado el propio programa.

Por ejemplo, pensemos en el programa más sencillo, disponible en la mayoría de los sistemas operativos, echo, que repite por pantalla sus argumentos. Su código sería algo tan sencillo como el Código fuente 57.

```
/* Ejemplo 3 */
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;
  for (i = 1; i < argc; i++)
    printf ("%s%s", argv[i], (i < argc - 1) ? " ":"");
  printf ("\n");
  return 0;
}
```

Código fuente 57

Punteros a funciones

Las funciones en C no son variables, sin embargo es posible definir punteros a funciones, que se pueden pasar como argumentos a otras funciones, almacenar en arrays, etc. Veamos un ejemplo simple en el Código fuente 58, creemos una función cambio que modifica el orden de dos números enteros pasados como argumentos del programa sólo sí el primero es mayor que el segundo. La función para cambiar los números pasa como argumento a cambio.

```
/* Ejemplo 4 */
#include <stdio.h>

void cambio (int *i, int *j, int (*funcion) (int*, int*))
{
  if (*i > *j) (*funcion) (i, j);
}

int swap (int *i, int *j)
{
  int tmp;
  tmp = *i;
  *i = *j;
  *j = tmp;
  return 0;
}

int main(int argc, char *argv[])
{
  int i, j;
  if (argc != 3)
  {
    perror ("Uso del programa: swapmax entero1 entero2");
    exit (1);
  }
  i = atoi (argv[1]);
  j = atoi (argv[2]);
  printf ("Enteros: i = %d - j = %d\n", i, j);
  cambio (&i, &j, &swap);
  printf ("Enteros: i = %d - j = %d\n", i, j);
  return 0;
}
```

Código fuente 58

Hemos analizado en este capítulo distintos elementos relacionados con punteros y arrays. El alumno puede observar que el uso de estos elementos concede gran potencia al lenguaje. Sin embargo, deben ser utilizados con precaución para optimizar recursos, y no para realizar programas crípticos.

Ejercicios

1. Comprobar que los punteros void pueden apuntar a cualquier variable, asignando el puntero creado a la dirección de un int y a la dirección de un float. (Nota: para imprimir direcciones de memoria usar %p en printf).

2. Desarrollar una versión propia de strlen que cuente el número de caracteres de una cadena. Utilizar un contador entero. Comprobar el funcionamiento con un ejemplo.

3. Rescribir la función anterior usando únicamente punteros a cadenas de caracteres, y la aritmética de punteros.

4. Crear su propia versión de la función de biblioteca strcpy que copia una cadena en otra.

5. Crear un array de punteros a cadenas de caracteres que almacene los nombres de los 12 meses del año. Crear un programa que lea un número entero por pantalla (1-12), e imprima al usuario el nombre del mes.

6. Basándose en el ejercicio 2 del capítulo 3, crear una función lógica que determine si un año es bisiesto. Con esa función crear un programa que devuelva el número de días dado un año y mes. Utilizar un array doble para almacenar los días del mes de años bisiestos y no bisiestos.

7. Rescribir el ejemplo 3 de forma que no se manipulen arrays, sino punteros a los argumentos del programa.

8. Rescribir el ejemplo 3 para que imprima al final de la lista de argumentos el nombre del ejecutable del programa entre paréntesis.

9. En el ejercicio 10 del capítulo anterior, donde buscábamos líneas de texto con una cadena dentro de ella, codificamos la cadena a buscar dentro del propio programa. Modificar este ejercicio para que lea la cadena a buscar desde la propia línea de comandos.

10. En Unix y Linux existe el programa cal que imprime el calendario al mes correspondiente al elegido por el usuario. Una ejecución típica conduce a ($ cal 2 2000):

    February 2000

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29				

    Realizar el programa C correspondiente para imprimir estos calendarios a partir del año 1900. Exigir al usuario la entrada por la línea de comandos de mes y año (4 dígitos). (Nota: el uno de enero de 1900 fue Lunes).

11. La fórmula para pagar las cuotas de un préstamo de cantidad p, con tasa anual t%, a n años es:

```
r = 12 / (12 + t/100);
cuota = p * (1 - r) / (r - pow(r,a*12+1)); /* usar math.h */
```

    Escribir un programa que lea de la línea de comandos una cantidad en pesetas, y un tipo de interés, e imprima las cuotas de 5 a 30 años.

Respuestas a los ejercicios

1. Comprobar que los punteros void pueden apuntar a cualquier variable, asignando el puntero creado a la dirección de un int y a la dirección de un float.

   (Nota: para imprimir direcciones de memoria usar %p en printf).

```
#include <stdio.h>

int main()
{
  int i = 2;
  float x = 2.1;
  void *p;
  p = &i;
  printf ("Puntero a entero: %p\n", &i);
  printf ("Puntero void:     %p\n", p);
  p = &x;
  printf ("Puntero a real: %p\n", &x);
  printf ("Puntero void:   %p\n", p);
  return 0;
}
```

2. Desarrollar una versión propia de strlen que cuente el número de caracteres de una cadena. Utilizar un contador entero. Comprobar el funcionamiento con un ejemplo.

```
#include <stdio.h>

int mi_strlen (char *s)
{
  int n;
  for (n = 0; *s != '\0'; s++) n++;
  return n;
}

int main()
{
  char *s = "Curso de C";
  char t[] = "Curso de C++";
  printf ("Cadena \'%s\': %d caracteres\n", s, mi_strlen(s));
  printf ("Cadena \'%s\': %d caracteres\n", t, mi_strlen(t));
  return 0;
}
```

3. Rescribir la función anterior usando únicamente punteros a cadenas de caracteres, y la aritmética de punteros.

```
#include <stdio.h>

int mi_strlen (char *s)
{
  int n;
  for (n = 0; *s != '\0'; s++) n++;
  return n;
}

int main()
{
  char *s = "Curso de C";
  char t[] = "Curso de C++";
  printf ("Cadena \'%s\': %d caracteres\n", s, mi_strlen(s));
  printf ("Cadena \'%s\': %d caracteres\n", t, mi_strlen(t));
  return 0;
}
```

4. Crear su propia versión de la función de biblioteca strcpy que copia una cadena en otra.

```
#include <stdio.h>

void mi_strcpy (char *dest, char *orig)
{
  while ((*dest = *orig) != '\0')
  {
    dest++;
    orig++;
  }
}

int main()
{
  char *s = "Curso de C";
  char *t;
  mi_strcpy (t, s);
  printf ("Cadena \'%s\'.\n", s);
  printf ("Cadena \'%s\'.\n", t);
  return 0;
}
```

5. Crear un array de punteros a cadenas de caracteres que almacene los nombres de los 12 meses del año. Crear un programa que lea un número entero por pantalla (1-12), e imprima al usuario el nombre del mes.

```
#include <stdio.h>

int main()
{
  char *mes[] = { "enero", "febrero", "marzo", "abril",
                  "mayo", "junio", "julio", "agosto",
                  "septiembre", "octubre", "noviembre",
                  "diciembre" };
  int i;
  printf ("Introduce el número del mes: ");
  if ( (scanf ("%d", &i)) != 1 || i < 1 || i > 12 )
  {
    perror ("Error en la lectura del mes");
    exit (1);
  }
  printf ("Corresponde al mes de \'%s\'.\n", mes[i-1]);
  return 0;
}
```

6. Basándose en el ejercicio 2 del capítulo 3, crear una función lógica que determine si un año es bisiesto. Con esa función crear un programa que devuelva el número de días dado un año y mes. Utilizar un array doble para almacenar los días del mes de años bisiestos y no bisiestos.

```
#include <stdio.h>

int bisiesto (int year)
{
  if ( ( year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
    return 1;
```

```
  return 0;
}

int main()
{
  char *mes[] = { "enero", "febrero", "marzo", "abril",
                  "mayo", "junio", "julio", "agosto",
                  "septiembre", "octubre", "noviembre",
                  "diciembre" };
  char dias[2][12] = {
    { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
    { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } };
  int i, j;
  printf ("Introduce el número del mes: ");
  if ( (scanf ("%d", &i)) != 1 || i < 1 || i > 12 )
  {
    perror ("Error en la lectura del mes");
    exit (1);
  }
  printf ("Introduce el número del año: ");
  if ( (scanf ("%d", &j)) != 1 || j < 0 )
  {
    perror ("Error en la lectura del año");
    exit (1);
  }
  printf ("El mes de %s del año %d tiene %d días.\n", mes[i-1], j,
    dias[bisiesto (j)][i-1]);
  return 0;
}
```

7. Rescribir el ejemplo 3 de forma que no se manipulen arrays, sino punteros a los argumentos del programa.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf ("%s%s", *(++argv), (argc > 1) ? " ":"");
  printf ("\n");
  return 0;
}
```

8. Reescribir el ejemplo 3 para que imprima al final de la lista de argumentos el nombre del ejecutable del programa entre paréntesis.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;
  for (i = 1; i < argc; i++)
    printf ("%s%s", argv[i], (i < argc - 1) ? " ":"");
  printf (" (%s)\n", argv[0]);
  return 0;
}
```

9. En el ejercicio 10 del capítulo anterior, donde buscábamos líneas de texto con una cadena dentro de ella, codificamos la cadena a buscar dentro del propio programa. Modificar este ejercicio para que lea la cadena a buscar desde la propia línea de comandos.

```
#include <stdio.h>
#define MAXLINEA 1000 /* Tamaño máximo de línea */

int strindex (char s[], char t[])
{
  int i, j, k;
  for (i = 0; s[i] != '\0'; i++)
  {
    for (j = i, k = 0; t[k] != '\0' && s[j] == t[k]; j++, k++);
    if (k > 0 && t[k] == '\0') return i;
  }
  return -1;
}

int getline (char s[], int lim)
{
  int c, i = 0;
  while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
    s[i++] = c;
  if (c == '\n') s[i++] = c;
  s[i] = '\0';
  return i;
}

int main(int argc, char *argv[])
{
  char linea[MAXLINEA];
  int encontrados = 0;

  if (argc != 2)
  {
    perror ("Uso del programa: ex9 argumento");
    exit (1);
  }
  while (getline (linea, MAXLINEA) > 0)
    if (strindex (linea, argv[1]) >= 0)
    {
      encontrados++;
      printf ("%d:%s", encontrados, linea);
    }
  return encontrados;
}
```

10. En Unix y Linux existe el programa cal que imprime el calendario al mes correspondiente al elegido por el usuario. Una ejecución típica conduce a ($ cal 2 2000):

February 2000

Su	Mo	Tu	We	Th	Fr	Sa
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29				

Realizar el programa C correspondiente para imprimir estos calendarios a partir del año 1900. Exigir al usuario la entrada por la línea de comandos de mes y año (4 dígitos). (Nota: el uno de enero de 1900 fue Lunes).

```
#include <stdio.h>


int bisiesto (int year)
{
  if ( ( year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
    return 1;
  return 0;
}


int main (int argc, char* argv[])
{
  int i, dia_Ene1, mes, agno, fecha;
  int dias_mes[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  char* nombre_mes[] = { "enero", "febrero", "marzo", "abril",
                 "mayo", "junio", "julio", "agosto",
                 "septiembre", "octubre", "noviembre",
                 "diciembre" };
  char _string[60];

  if (argc != 3)
  {
    perror ("Uso: ex10 mes año\n");
    exit (1);
  }
  mes = atoi (argv[1]);
  agno = atoi (argv[2]);
  if (mes < 1 || mes > 12)
  {
    perror ("Error en la lectura del mes.\n");
    exit (1);
  }
  if (agno < 1900)
  {
    perror ("Error en la lectura del año.\n");
    exit (1);
  }
  /* Averiguando el dia de la semana para el año */
  for (fecha = 1900, dia_Ene1 = 1; fecha < agno; fecha++)
  {
    if (bisiesto (fecha)) dia_Ene1 += 2;
    else dia_Ene1++;
    if (dia_Ene1 >= 7) dia_Ene1 -= 7;
  }
  /* 366 dias para un año bisiesto */
  if (bisiesto (agno)) ++dias_mes[1];
  for (i = 1; i < mes; i++)
    dia_Ene1 = (dia_Ene1 + dias_mes[i-1]) % 7;
  printf ("%15s %5d\n", nombre_mes[mes-1], agno);
  printf ("Dom Lun Mar Mie Jue Vie Sab\n");
  printf ("--- --- --- --- --- --- ---\n");
  for (fecha = 1; fecha <= dia_Ene1 * 4; fecha++)
    printf (" ");
  for (fecha = 1; fecha <= dias_mes[mes-1]; fecha++)
  {
    printf ("%3d", fecha);
    if ((fecha + dia_Ene1) % 7 > 0)
      printf (" ");
    else printf("\n");
```

```
  }
  printf ("\n");
  return 0;
}
```

11. La fórmula para pagar las cuotas de un préstamo de cantidad p, con tasa anual t%, a n años es:

```
r = 12 / (12 + t/100);
cuota = p * (1 - r) / (r - pow(r,a*12+1)); /* usar math.h */
```

Escribir un programa que lea de la línea de comandos una cantidad en pesetas, y un tipo de interés, e imprima las cuotas de 5 a 30 años.

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

double pago (double prestamo, double tasa, int agnos)
{
  double r = 12.0 / (12.0 + tasa / 100.0);
  return prestamo * (1.0 - r) / (r - pow (r, agnos * 12.0 + 1.0));
}

int main (int argc, char *argv[])
{
  double tasa, prestamo;
  int i;
  if (argc != 3)
  {
    perror ("Uso: ex11 tasa prestamo\n");
    exit (1);
  }
  tasa = atof (argv[1]);
  prestamo = atof (argv[2]);
  if (prestamo < 0.0)
  {
    perror ("Error en la lectura del préstamo.\n");
    exit (1);
  }
  printf ("Préstamo: %ld Pts - Interés %6.2f\%\n", (long)prestamo, tasa);
  printf ("Años:   ");
  for (i = 0; i < 6; i++)
    printf ("%8d", 5*(i+1));
  printf ("\nCuotas: ");
  for (i = 0; i < 6; i++)
    printf ("%8ld", (long)pago (prestamo, tasa, 5*(i+1)));
  printf ("\n");
  return 0;
}
```

6

Estructuras, gestión de archivos y otros elementos del lenguaje

Introducción

En este capítulo vamos a describir dos elementos fundamentales de la programación en C. Por un lado, vamos a analizar cómo crear elementos complejos con datos a través de las estructuras, y por otro lado, vamos a estudiar en detalle las bibliotecas de entrada/salida en C. Con este capítulo cerramos la parte dedicada a la programación en C, para adentrarnos en las siguientes secciones en la programación orientada a objetos y C++.

Estructuras

Una estructura (struct) es simplemente una colección de una o más variables, posiblemente de distinto tipo, agrupadas bajo un nombre común para facilitar su manipulación. En otros lenguajes, por ejemplo en Pascal, se denominan registros (*records*), porque son muy utilizados para manipular bases de datos. De hecho, los ejemplos tradicionales de creación de estructuras están íntimamente ligados a la gestión de bases de datos. Hasta la creación del estándar ANSI, las reglas para el uso de las estructuras por los compiladores no estaban definidas, y dependían del compilador.

Imaginemos una estructura simple para la gestión de coordenadas en un plano. La sintaxis es:

```
struct punto {
   int x;
```

```
    int y;
};
```

Las variables definidas dentro de la estructura se denominan miembros. Una variable normal y una variable miembro de una estructura pueden tener el mismo nombre, porque es posible distinguir una de otra por el contexto. Una estructura equivale a la definición de un tipo, por lo que es posible definir variables, en la propia declaración o a a.

```
struct punto {
   int x;
   int y;
} puntoA, puntoB;
struct punto pA, pB, pC;
```

Código fuente 59

Las estructuras se pueden inicializar con una lista de valores iniciales.

```
struct punto A = { 200, 110 };
```

Código fuente 60

Los miembros de una estructura son accesibles con el operador ".": estructura.miembro. Por ejemplo, para imprimir las coordenadas de un punto escribiremos el Código fuente 61.

```
printf("(%d, %d)", A.x, A.y);
```

Código fuente 61

Las estructuras pueden contener como variables miembro otras estructuras, es decir, se puede anidar.

Estructuras y funciones

Las únicas operaciones legales con estructuras son:

- copiar o asignar como una unidad (incluyendo el paso como argumento de funciones, y el uso como valor de retorno de funciones,
- recoger su dirección de memoria con el operador &, y
- acceder a sus miembros.

Por ejemplo, creemos una función sumaP que suma las coordenadas de dos puntos. Su sintaxis será:

```
struct punto sumaP (struct punto p1, struct punto p2)
{
   struct punto tmp;
```

```
    tmp.x = p1.x + p2.x;
    tmp.y = p1.y + p2.y;
    return tmp;
}
```

Por supuesto, también es posible pasar estructuras por referencia a funciones, en lugar de por valor. En el caso de utilizar punteros, la referencia a las variables miembro se puede realizar de dos formas:

- (*puntero_estructura).miembro; o
- puntero_estructura->miembro.

Por ejemplo, la función anterior se puede rescribir como muestra el Código fuente 62.

```
struct punto sumaP (struct punto *p1, struct punto *p2)
{
   struct punto tmp;
   tmp.x = p1->x + p2->x;
   tmp.y = p1->y + p2->y;
   return tmp;
}
```

Código fuente 62

La función sumaP se debe invocar como vemos en el Código fuente 63.

```
puntoX = sumaP (&p1, &p2);
```

Código fuente 63

Estructuras y arrays

Como cualquier otro tipo en C, se pueden crear arrays de estructuras. Por ejemplo, podemos crear una estructura apropiada para contar el número de veces que cierta palabra clave aparece en un texto. Veámoslo en el Código fuente 64.

```
struct clave {
   char *palabra;
   int contador;
};
```

Código fuente 64

A partir de ahí creamos una estructura para contar artículos (Código fuente 65)

```
struct clave articulos[] = {
   "el", 0,
```

```
    "ella", 0,
    "ellos", 0,
    "ellas", 0
};
```

Código fuente 65

Observe el lector, como se inicializa el array con pares palabra, contador. El uso de llaves para cada estructura es opcional. Así, podríamos rescribir el caso anterior como vemos en el Código fuente 66.

```
struct clave articulos[] = {
    {"el", 0},
    {"ella", 0},
    {"ellos", 0},
    {"ellas", 0}
};
```

Código fuente 66

Typedef

C proporciona una instrucción denominada typedef para crear nuevos tipos a partir de otros existentes. Estos nuevos tipos se pueden usar en declaraciones, definiciones, etc. al igual que los tipos estándar de C. La sintaxis es:

```
typedef int Distancia;
typedef struct punto Punto;
typedef *double PDouble;
```

Otros tipos de datos

En C existen otros tipos de datos avanzados como son las uniones (union) y los campos de bits (bit-fields). Su uso es para aplicaciones muy concretas, y creemos que su alcance está fuera de un curso introductorio. Recomendamos al lector interesado la lectura de la excelente ayuda de Microsoft Visual C++.

Biblioteca estándar

Como ya mencionamos en el capítulo introductorio, las facilidades de entrada y salida no son una parte del lenguaje C en sí mismo. Sin embargo, el estándar ANSI define estas bibliotecas muy concretamente, de forma que cualquier programa que las utilice será portable de un sistema a otro. En este capítulo describiremos la biblioteca estándar que es un conjunto de funciones para gestión de archivos, manipulación de cadenas, gestión de almacenamiento, rutinas matemáticas y otros servicios.

Entrada y salida estándar

Como ya hemos visto en diversos ejemplos y ejercicios, la entrada y salida de texto se realiza en C mediante secuencias de texto, que no son más que una sucesión de líneas, cada una terminada con un carácter de salto de línea o nueva línea ('\n').

La función más simple que podemos describir la vemos en el Código fuente 67.

```
int getchar (void);
```

Código fuente 67

Esta función lee un carácter de la entrada estándar (teclado) cuando es invocada. Devuelve como valor de retorno el siguiente carácter o EOF (fin de archivo, constante definida en <stdio.h>, típicamente -1) cuando se encuentra con el fin del archivo. En muchos entornos (MS-DOS o Windows 9x/NT también), el teclado puede ser sustituido por un archivo con el signo de redirección de entrada '<'.

```
programa < archivo_de_entrada
```

La función:

```
int putchar (int c);
```

Código fuente 68

Sitúa el carácter en la salida estándar (pantalla). Devuelve el carácter escrito o EOF si ocurre un error. Utilizando el signo de redirección de salida ('>'), la pantalla puede ser dirigida a un archivo.

```
programa > archivo_de_salida
```

Las funciones de entrada y salida están incluidas en <stdio.h>. Cualquier archivo que pretenda utilizar estas funciones debe incluir el Código fuente 69.

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

Código fuente 69

Salida con formato: printf

La función printf traduce valores internos (variables, en general) a caracteres. Su sintaxis es:

```
int printf (char *format, arg1, arg2, ...);
```

El valor de retorno es el número de caracteres imprimidos. format es la cadena de formateado. Contiene dos tipos de caracteres:

- Caracteres ordinarios, copiados tal como aparecen en la secuencia de salida.

- Especificaciones de conversión, cada una de las cuales se asocia al siguiente argumento de la lista: arg1, arg2, etc. Cada conversión comienza con el signo '%'. El formato de estas especificaciones es (en ese orden):

  ```
  %[-]w.p[h|l]c
  ```

  Siendo:

  - -: indicador de alineación a la izquierda
  - w: ancho del campo
  - p: precisión. Este término indica el número máximo de caracteres a imprimir de una cadena, o el número de dígitos después del punto decimal en un número real, o el mínimo número de dígitos de un entero. También se puede expresar con un asterisco (*) que es sustituido por el siguiente argumento de la lista de argumentos.
  - h ó l. h indica que es un número entero de tipo short, y l que es un número entero long, o un real de tipo double.
  - c: carácter de conversión. La Tabla 2, nos muestra como puede ser.

Carácter	**Tipo de Argumento**
d,i	int: número entero
O	int: número octal sin signo (sin cero al principio)
x, X	int: número hexadecimal sin signo (sin 0x, ó 0X)
U	int: número entero sin signo
C	int: carácter
S	char *: cadena de caracteres
F	double: [-]m.*dddd* donde el número de d's viene dado por la precisión
e, E	double: [-]m.*dddd* e±xx ó [-]m.*dddd* E±xx, donde el número de d's viene dado por la precisión
g, G	double: usa %e si el exponente es menor que -4 o mayor o igual que la precisión; si no, usa %f
P	void *: puntero
%	imprime el símbolo %

Tabla 2

El Código fuente 70 muestra la cadena "hola mundo" (10 caracteres) con varios formatos.

```
:%8s:          :hola mundo: /* muestra todo */
:%.8s:         :hola mun: /* sólo 8 caracteres */
:%-12s:        :hola mundo  :
```

Código fuente 70

Lista de argumentos variable

Durante muchos ejemplos y ejercicios de los realizados con anterioridad hemos visto como la función printf admite una lista de argumentos de longitud variable. Este hecho puede parecer en principio un caso excepcional. Sin embargo, C ofrece la posibilidad de diseñar funciones con una lista variable de argumentos. Esta posibilidad se ofrece usando la biblioteca estándar <stdarg.h>. La sintaxis de la función será:

```
tipo funcion (argumento1, ...);
```

El número de y los propios argumentos se determinan con las siguientes macros de <stdarg.h>:

- va_list: declara una variable que se referirá a cada argumento por turno. Por ejemplo, se puede llamar ap a esta variable:

  ```
  va_list ap;
  ```

- va_start: macro que inicializa ap apuntando al primer argumento "sin nombre". Debe ser llamado una vez antes de usar ap. Al menos debe haber un argumento con nombre:

  ```
  va_start (ap, &argumento1);
  ```

- va_arg: devuelve un argumento y sitúa ap en el siguiente argumento:

  ```
  va_arg (ap, tipo);
  ```

- va_end: realiza la "limpieza" necesaria antes del return de la función:

  ```
  va_end (ap);
  ```

Entrada con formato: scanf

Aunque ya hemos usado en algunos ejemplos scanf, conviene detenernos unos instantes en su sintaxis. Esta función tiene un funcionamiento similar a printf. Su sintaxis es:

```
int scanf (char *format, ...);
```

La lista de argumentos después del formato está formada por punteros. Uno de los errores más frecuentes en la programación en C es olvidarse de este hecho, y no incluir como argumentos los punteros a las variables, sino las propias variables. Finaliza al terminar la lista de argumentos o cuando la entrada no concuerda con la especificación de control. Devuelve el número de entradas que concuerdan con éxito. Devuelve EOF cuando encuentra el fin de archivo. Devuelve cero cuando el primer argumento no concuerda con la primera especificación del formato. Existen funciones con características similares cuya sintaxis se puede consultar en la ayuda de Microsoft Visual C++: sscanf, fscanf.

La cadena de formato puede contener:

- Blancos o tabuladores que son ignorados.
- Caracteres ordinarios (no %) que deben concordar con el siguiente carácter no blanco de la entrada.
- Especificaciones de conversión:
    - %
    - *: Carácter de supresión (opcional)
    - w: Especificación de ancho de campo (opcional)
    - h: Conversión a short (opcional)
    - l, L: Conversión a long o double (opcional)
    - c: Carácter de conversión (consultar la Tabla 3).

Carácter	**Tipo de Argumento**
d	int *: número entero decimal
i	int *: número entero octal 0*ddd* ó hexadecimal 0x*ddd*
o	int *: número octal con o sin 0 al inicio
u	unsigned int *: número entero decimal sin signo
x	int *: número hexadecimal con o sin 0x al inicio
c	char *: lee el siguiente carácter cualquiera que sea, incluidos blancos. Para leer el siguiente carácter no blanco usar %1s
s	char *: cadena de caracteres
e, f, g	float *: número real en cualquier formato
%	imprime el símbolo %

Tabla 3

Acceso a archivos

En todos los ejemplos y ejercicios realizados hasta ahora, la entrada y salida de datos se realizaban por teclado y pantalla. Obviamente, no son procedimientos muy cómodos para manipular grandes cantidades de datos. Por eso, C implanta facilidades para gestionar archivos para lectura y escritura de datos. El primer paso es abrir el archivo. Abrir un archivo es una interacción con el sistema operativo que da como resultado un puntero a una estructura de tipo archivo (FILE), definida con typedef. La sintaxis es:

```
FILE *fp;
```

Donde fp es el puntero al archivo. El siguiente paso es invocar la función de apertura.:

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

donde nombre es el nombre del archivo y modo es el modo de apertura, como veremos más adelante.

```
fp = fopen ("resultados.dat", "w");
```

La estructura FILE contiene la siguiente información:

- La localización del búfer.
- Posición del carácter actual en el búfer.
- Si el archivo se abre para lectura o escritura.
- Errores e información sobre fin de archivo.

Los modos de apertura son:

- "r": lectura
- "w": escritura
- "a": añadir

En algunos sistemas se dispone también de estas opciones:

- "t": texto
- "b": binario

Si un archivo no existe, con "w" o "a" se creará si es posible. Si existe el archivo, con "w" se destruye su contenido anterior; con "a" se preserva su contenido, y se añade la información al final del archivo. Cuando hay error, fopen devuelve NULL.

Para leer y escribir datos carácter a carácter, C dispone de dos funciones. Para leer, contamos con:

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

que lee el siguiente carácter del archivo de entrada fp. Cuando hay error, esta función devuelve EOF. Para escribir debemos usar:

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

que escribe el carácter c en el archivo de salida fp. Devuelve EOF cuando hay error.

Debemos recordar que el sistema operativo abre los siguientes archivos al arrancar el sistema (definidos en <stdio.h>), que pueden ser redirigidos:

- Entrada estándar (teclado): stdin
- Salida estándar (pantalla): stdout

- Salida de error estándar (pantalla sin búfer): stderr

La diferencia entre stdout y stderr, es que en el primer caso la salida es dirigida a la pantalla, pero se guarda en un búfer de salida que acumula datos hasta determinado tamaño, y en el segundo caso, no existe tal búfer. La forma de redireccionar ambas salidas depende del sistema operativo. En base a estas funciones, se definen las macros que aparecen en el Código fuente 71, ya utilizadas en capítulos anteriores.

```
#define getchar() getc(stdin)
#define putchar(c) putc((c), stdout)
```

Código fuente 71

Además, C dispone de funciones para entrada y salida formateada en archivos. Son las funciones que muestra el Código fuente 72.

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

Código fuente 72

Que se corresponden respectivamente con scanf y printf, con un primer argumento adicional que corresponde al archivo. Nunca se debe olvidar que la creación y utilización de archivos consume recursos del sistema, que deben ser liberados cuando su uso no sea necesario. Para ello, debemos cerrar los archivos con:

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

Manipulación de errores

La gestión de errores es un aspecto fundamental de la programación en cualquier lenguaje. C no proporciona una gestión de errores compleja, como puede ser el caso de C++ con la manipulación de excepciones, pero sí facilita herramientas para que el programador desarrolle sus propias herramientas. Para empezar, se define la salida estándar de errores stderr, que coincide normalmente la pantalla, salvo que el usuario la redireccione a un archivo.

El programador dispone además de la función exit que termina la ejecución de un programa cuando es invocada. Esta función envía un valor al sistema operativo. El convenio que se sigue es el Código fuente 73.

```
exit (0); /* No hay errores de ejecución */
exit (no_0); /* Error en la ejecución */
```

Código fuente 73

Dentro de main, "exit (expresion)" equivale a "return expresion". exit puede ser llamada desde cualquier función.

El sistema dispone además de la función ferror para detectar errores en la gestión de archivos:

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

que devuelve un valor no nulo si hay un error en fp. Los errores de salida son poco frecuentes, pero conviene comprobarlos en cualquier caso (por ejemplo, disco duro lleno, etc.). También se dispone de la función feof para comprobar si estamos en el fin de un archivo:

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

Devuelve un valor no nulo si es fin de archivo en fp.

Entrada y salida por líneas

Existen dos funciones muy utilizadas que manipulan líneas (o cadenas de caracteres). Para lectura, disponemos de la función fgets:

```
char *fgets (char *linea, int maxlinea, FILE *fp);
```

que lee la siguiente línea de entrada (incluido el carácter de nueva línea) del archivo fp. La línea debe contener como máximo maxlinea-1 caracteres. La cadena resultante es terminada con el carácter nulo \0. Devuelve linea si no hay error; si hay error devuelve NULL. Para escritura podemos utilizar la función fputs:

```
int *fputs (char *linea, FILE *fp);
```

que escribe linea en el archivo de salida fp. Devuelve cero cuando no hay error y EOF cuando hay error.

Funciones misceláneas

Existen multitud de funciones adicionales definidas en la biblioteca estándar, sobre las que vamos a detenernos brevemente. En la ayuda de Microsoft Visual C++ existen descripciones detalladas de todas ellas.

Operaciones con cadenas de caracteres

Definidas en la biblioteca <string.h>, encontramos las que aparecen en la Tabla 4.

strcat (s, t)	Concatena t al final de s
strncat (s, t, n)	Concatena n caracteres de t al final de s
strcmp (s, t)	Devuelve negativo, cero, o positivo sis<t, s==t, s>t
strncmp (s, t, n)	igual que strcmp, pero en los primeros n caracteres

strcpy(s, t)	copia t en s
strncpy (s, t, n)	copia al menos n caracteres de t en s
strlen (s)	longitud de s
strchr (s, c)	puntero a la primera ocurrencia de c en s; NULL si no existe
strrchr (s, c)	puntero a la última ocurrencia de c en s; NULL si no existe

Tabla 4

donde t y s son char*, y n y c son int.

Comparación del tipo de carácter y conversión

Definidas en la biblioteca <ctype.h>, encontramos las mostradas en la Tabla 5.

isalpha(c)	no-cero si c es una letra; 0 si no lo es
isupper(c)	no-cero si c es una letra mayúscula; 0 si no lo es
islower(c)	no-cero si c es una letra minúscula; 0 si no lo es
isdigit(c)	no-cero si c es un dígito; 0 si no lo es
isalnum(c)	no-cero si c es un carácter alfanumérico; 0 si no lo es
isspace(c)	no-cero si c es un blanco, tabulador, nueva línea, retorno de carro, salto de página, o tabulador vertical; 0 si no lo es
toupper(c)	convierte c en mayúscula
tolower(c)	convierte c en minúscula

Tabla 5

donde c es un int que se puede representar como unsigned char o EOF. Estas funciones siempre devuelven int.

Ejecución de comandos del sistema operativo

C dispone de la función system para ejecutar comandos del sistema operativo. No recomendamos su uso.

```
system ("comando_de_sistema_operativo");
```

Funciones matemáticas

Definidas en la biblioteca <math.h>, encontramos las que muestra la Tabla 6.

sin(x)	Seno de x, x en radianes
Cos(x)	Coseno de x, x en radianes
atan2(y, x)	Arco tangente de y/x; y, x en radianes
Exp(x)	Función exponencial e^x
Log(x)	Logaritmo natural (base e) de x > 0
Log10(x)	Logaritmo común (base 10) de x > 0
pow(x, y)	x^y
Sqrt(x)	Raíz cuadrada de x (x>=0)
fabs(x)	Valor absoluto de x

Tabla 6

donde x e y son double.

Ejercicios

1. Crear un programa en C que manipule coordenadas en un plano, y que calcule la distancia entre dos puntos. (Nota: la distancia entre dos puntos se define como sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)); usar la biblioteca <math.h>).

2. Calcular con sizeof el tamaño en bytes de la estructura punto, y compararlo con el tamaño de sus variables miembro.

3. Crear un programa en C que defina una estructura de agenda para nombres y direcciones de correo electrónico. Pidiendo datos al usuario, e imprimiéndolos por pantalla.

4. Definir una estructura rectángulo definida por dos puntos (su vértice inferior izquierdo y su vértice superior derecho). Crear una función lógica que determine si dado un punto en el plano, está en el interior del rectángulo o no.

5. Definir una estructura círculo definida por su centro y su radio. Leer dos círculos desde el teclado, y comprobar si intersectan o no.

6. Crear un programa que clasifique y cuente los caracteres introducidos por la entrada estándar en 3 grupos: dígitos (cada dígito individualmente), espacios en blanco (blancos, tabuladores, retornos de carro o saltos de línea) y el resto.

7. Escribir un programa para probar todas las opciones de formateado con cadenas de caracteres y enteros.

8. Escribir un programa para recordar la impresión de los caracteres especiales con printf.

9. Crear un programa en C que concatene todos los archivos introducidos por la línea de comandos en la salida estándar. Gestionar adecuadamente los posibles errores.

10. Crear un programa en C que cuente todas las líneas y el número de caracteres de todos los archivos introducidos por la línea de comandos. Gestionar adecuadamente los posibles errores.

Respuestas a los ejercicios

1. Crear un programa en C que manipule coordenadas en un plano, y que calcule la distancia entre dos puntos. (Nota: la distancia entre dos puntos se define como sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)); usar la biblioteca <math.h>).

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

struct punto {
  double x;
  double y;
};

int main()
{
  struct punto pA = { 1.0, 2.0 };
  struct punto pB = { 2.0, 1.0 };
  double dist;

  dist = sqrt ((pB.x - pA.x) * (pB.x - pA.x) +
               (pB.y - pA.y) * (pB.y - pA.y));
  printf ("Distancia A-B: %f\n", dist);
  return 0;
}
```

2. Calcular con sizeof el tamaño en bytes de la estructura punto, y compararlo con el tamaño de sus variables miembro.

```
#include <stdio.h>

struct punto {
  double x;
  double y;
};

int main()
{
  struct punto pA = { 0.0, 0.0 };
  printf ("Tamaño punto: %d\n", sizeof(pA));
  printf ("Tamaño punto.x: %d\n", sizeof(pA.x));
  printf ("Tamaño punto.y: %d\n", sizeof(pA.y));
  return 0;
}
```

3. Crear un programa en C que defina una estructura de agenda para nombres y direcciones de correo electrónico. Pidiendo datos al usuario, e imprimiéndolos por pantalla.

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

#define MAXITEMS 10
#define LINE 20
typedef struct agenda A;
struct agenda {
  char nombre[LINE];
  char email[LINE];
};

int main()
{
  A a[MAXITEMS];
  int i;

  for (i = 0; i < MAXITEMS; i++)
  {
    printf ("Nombre: ");
    scanf ("%s", a[i].nombre);
    if (!strcmp (a[i].nombre, "quit")) break;
    printf ("Correo electrónico: ");
    scanf ("%s", a[i].email);
  }
  while ((--i)+1)
  {
    printf ("Nombre: %s - Correo electrónico: %s\n",
      a[i].nombre, a[i].email);
  }
  return 0;
}
```

4. Definir una estructura rectángulo definida por dos puntos (su vértice inferior izquierdo y su vértice superior derecho). Crear una función lógica que determine si dado un punto en el plano, está en el interior del rectángulo o no.

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

typedef struct punto Punto;
typedef struct rect Rect;
struct punto {
  double x;
  double y;
};
struct rect {
  struct punto p1;
  struct punto p2;
};

int p_en_rect (Punto, Rect);

int main()
{
  Punto p;
  Rect r;

  printf ("Define el rectángulo con sus dos puntos (x,y): ");
  scanf ("%lf %lf %lf %lf", &r.p1.x, &r.p1.y, &r.p2.x, &r.p2.y);
  printf ("Define el punto a analizar: ");
  scanf ("%lf %lf", &p.x, &p.y);
```

```
  printf ("Rectángulo: (%lf, %lf) -  (%lf, %lf)\n",
          r.p1.x, r.p1.y, r.p2.x, r.p2.y);
  printf ("Punto: (%lf, %lf)\n", p.x, p.y);
  if (p_en_rect (p, r)) printf ("Punto en Rectángulo.\n");
  else printf ("Punto fuera de rectángulo.\n");
  return 0;
}

int p_en_rect (Punto p, Rect r)
{
  return p.x >= r.p1.x && p.x <= r.p2.x &&
         p.y >= r.p1.y && p.y <= r.p2.y;
}
```

5. Definir una estructura círculo definida por su centro y su radio. Leer dos círculos desde el teclado, y comprobar si intersectan o no.

```
#include <math.h>
#include <stdio.h>
typedef struct punto Punto;
typedef struct circulo Circ;
struct punto {
  double x;
  double y;
};
struct circulo {
  struct punto centro;
  double radio;
};

double distancia (Punto, Punto);
int intersecta (Circ, Circ);

int main()
{
  Circ c1, c2;

  printf ("Define el círculo 1 con su centro y su radio (x,y,r): ");
  scanf ("%lf %lf %lf", &c1.centro.x, &c1.centro.y, &c1.radio);
  printf ("Define el círculo 2 con su centro y su radio (x,y,r): ");
  scanf ("%lf %lf %lf", &c2.centro.x, &c2.centro.y, &c2.radio);
  printf ("Círculo 1: (%lf, %lf) - radio = %lf\n",
          c1.centro.x, c1.centro.y, c1.radio);
  printf ("Círculo 2: (%lf, %lf) - radio = %lf\n",
          c2.centro.x, c2.centro.y, c2.radio);
  printf ("Distancia entre centros = %lf\n",
           distancia (c1.centro, c2.centro));
  if (intersecta (c1, c2)) printf ("Intersectan.\n");
  else printf ("No intersectan.\n");
  return 0;
}

double distancia (Punto p1, Punto p2)
{
  return sqrt((p1.x-p2.x)*(p1.x-p2.x)+
              (p1.y-p2.y)*(p1.y-p2.y));
}

int intersecta (Circ c1, Circ c2)
{
  return (distancia (c1.centro, c2.centro) < c1.radio+c2.radio);
}
```

6. Crear un programa que clasifique y cuente los caracteres introducidos por la entrada estándar en 3 grupos: dígitos (cada dígito individualmente), espacios en blanco (blancos, tabuladores, retornos de carro o saltos de línea) y el resto.

```
#include <stdio.h>

int main()
{
  int c, i, nwhite, nother, ndigit[10];

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

7. Escribir un programa para probar todas las opciones de formateado con cadenas de caracteres y enteros.

```
#include <stdio.h>

int main()
{
  char *string = "Esto es una cadena de caracteres";
  int n = 125;

  printf ("\tCADENAS DE CARACTERES\n");
  printf ("Impresión %%s                 :%s:\n", string);
  printf ("Impresión %%40s               :%40s:\n", string);
  printf ("Impresión %%-40s izquierda    :%-40s:\n", string);
  printf ("Impresión %%.25s              :%.25s:\n", string);
  printf ("Impresión %%40.25s            :%40.25s:\n", string);
  printf ("Impresión %%-40.25s izquierda :%-40.25s:\n", string);
  printf ("Impresión %%*s                :%*s:\n",
          strlen(string), string);

  printf ("\n\tNUMEROS ENTEROS\n");
  printf ("Impresión %%d               :%d:\n", n);
  printf ("Impresión %%6d              :%6d:\n", n);
  printf ("Impresión %%-6d izquierda   :%-6d:\n", n);
```

```
  printf ("Impresión %%.4d             :%.4d:\n", n);
  printf ("Impresión %%6.4d            :%6.4d:\n", n);
  printf ("Impresión %%-6.4d izquierda :%-6.4d:\n", n);
  printf ("Impresión %%06d             :%06d:\n", n);
  printf ("Impresión %%*d              :%*d:\n", 4, n);
  printf ("Impresión %%6o octal        :%6o:\n", n);
  printf ("Impresión %%6x hexadecimal  :%6x:\n", n);

  return 0;
}
```

8. Escribir un programa para recordar la impresión de los caracteres especiales con printf.

```
#include <stdio.h>

int main()
{
  int i;

  for (i = 0; i < 5; i++)
    printf ("CARACTER DE ALARMA\a\n");
  printf ("\tESTE TEXTO COMIENZA CON UN TABULADOR\n");
  printf ("\tESTE ES UN RETORNO DE CARRO SIN NUEVA LINEA\rVES!\n");
  printf ("IMPRIMIMOS UNAS COMILLAS DOBLES: \"\n");
  printf ("IMPRIMIMOS UNAS COMILLAS SIMPLES: \'\n");
  printf ("IMPRIMIMOS UN BACKSLASH: \\\n");
  return 0;
}
```

9. Crear un programa en C que concatene todos los archivos introducidos por la línea de comandos en la salida estándar. Gestionar adecuadamente los posibles errores.

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

void filecopy (FILE *, FILE *);

int main (int argc, char *argv[])
{
  FILE *fp;
  char *program_name = argv[0];

  if (argc == 1) /* sin argumentos copia la entrada estándar */
    filecopy (stdin, stdout);
  else
    while (--argc > 0)
    {
      if ( (fp = fopen (*++argv, "r")) == NULL )
        {
          fprintf (stderr, "%s: error abriendo %s\n",
                   program_name, *argv);
          exit (1);
        }
      else
        {
          filecopy (fp, stdout);
          fclose (fp);
        }
    }
  if (ferror (stdout))
```

```
    {
      fprintf (stderr, "%s: error escribiendo en stdout\n",
               program_name);
      exit (2);
    }

  return 0;
}

/* Función void filecopy: copia el archivo input en output */
void filecopy (FILE *input, FILE *output)
{
  int c;
  while ( (c = getc (input)) != EOF ) putc (c, output);
}
```

10. Crear un programa en C que cuente todas las líneas y el número de caracteres de todos los archivos introducidos por la línea de comandos. Gestionar adecuadamente los posibles errores.

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

void filecount (char *, char *);

int main (int argc, char *argv[])
{
  char *program_name = argv[0];

  if (argc == 1) /* sin argumentos: mensaje de error */
  {
    fprintf (stderr, "%s: Llamada sin argumentos\n"
             "Uso: \'f_count lista_ficheros\'\n", program_name);
    exit (1);
  }
  else
    while (--argc > 0)
      filecount (program_name, *++argv);

  return 0;
}

/* Función void filecount: cuenta los caracteres y
   las líneas del fichero introducido */
void filecount (char *program_name, char *file_name)
{
  FILE *input;
  int nc = 0, nl = 1, c;
  if ( (input = fopen (file_name, "r")) == NULL )
  {
    fprintf (stderr, "%s: error abriendo %s\n",
             program_name, file_name);
    exit (1);
  }
  else
  {
    while ( (c = getc (input)) != EOF )
    {
      ++nc;
      if ( c == '\n' ) ++nl;
    }
  }
  fclose (input);
  fprintf (stdout, "%s:\n\t%d caracteres\n\t%d líneas\n",
```

```
            file_name, nc, nl);
  if (ferror (stdout))
  {
    fprintf (stderr, "Error escribiendo en stdout\n");
    exit (2);
  }
}
```

7

Introducción a C++

C++ es un lenguaje de programación orientado a objetos inventado a principios de los años 80 por Bjarne Stroustrup en los laboratorios de AT&T en Nueva Jersey (EE.UU.), famosos por ser donde trabajaban los inventores del lenguaje C y del sistema operativo UNIX, Dennis Ritchie y Brian Kernighan. La idea original de Stroustrup era desarrollar un lenguaje para uso interno en el laboratorio, que fuera una mejora al C, una especie de «C con clases». Debido a ello, C++ incorporó características de diversos lenguajes como Ada, Simula67 o Algol68. Sin embargo, el lenguaje no se redujo a estas incorporaciones, y además de conservar la versatilidad y potencia de C, ha crecido con las necesidades de sus usuarios hasta convertirse en uno de los lenguajes de programación más utilizados en la actualidad. En general, podemos decir que *C++* es un lenguaje de programación de propósito general con cierta desviación hacia la programación de sistemas que:

- es un C mejorado,
- soporta la abstracción de datos,
- soporta la programación orientada a objetos (POO), y
- soporta otros tipos de programación genérica.

La herencia de C es vista por muchos puristas como una debilidad del lenguaje. Sin embargo, el hecho de que C++ conserve una compatibilidad total con C, le dota de una potencia que no tienen otros lenguajes orientados a objetos «puros» como Java o Smalltalk, y además no «fuerza» ningún estilo de programación.

Este curso está orientado a enseñar al alumno la utilización del entorno de desarrollo Microsoft Visual C++ en su versión 6.0. Recomendamos al alumno que se familiarice con el entorno de Microsoft para

realizar los ejemplos propuestos, y se familiarice con las técnicas de programación descritas. Para ello es conveniente que lea el capítulo « Introducción al entorno de Microsoft Visual C++». Asimismo, supondremos que el alumno posee ciertos conocimientos de POO, aunque revisaremos los conceptos fundamentales a lo largo del curso.

Por otro lado, debemos recordar que C++ es un lenguaje estandarizado, que ha sido adoptado por la ISO (International Organization for Standardization, Organización Internacional de Estandarización) y varias organizaciones nacionales como ANSI (The American National Standards Institute, el Instituto Nacional Americano de Estándares), BSI (The British Standards Institute, el Instituto Británico de Estándares), y DIN (la Organización Alemana de Estándares Nacionales) el 14 de Noviembre de 1997. El comité ANSI-C++ es conocido por "X3J16". El grupo del estándar ISO C++ es conocido como "WG21". Entre las compañías que han contribuido al estándar figuran AT&T, Ericsson, Digital, Borland, Hewlett Packard, IBM, Mentor Graphics, Microsoft, Silicon Graphics, Sun Microsystems, y Siemens.

Un recorrido rápido por C++

Desde los tiempos en que Kernighan y Ritchie publicaron su famoso « The C Programming Language», es tradicional que todos los libros y cursos de programación comiencen con su versión del programa « Hola Mundo», pero antes de adentrarnos en esos detalles, vamos a ver un recorrido rápido sobre las capacidades del lenguaje. El alumno debe tener presente que no vamos a describir todas y cada una de las características del lenguaje, porque ese objetivo va más allá de un curso introductorio, pero sí vamos a facilitar las herramientas que permitan al alumno mejorar sus prestaciones en el uso del lenguaje.

En primer lugar, C++ facilita todas las herramientas típicas de lo que se denomina programación procedural, clásica de lenguajes no orientados a objetos como Fortran, C o Pascal. Para ello tenemos

- tipos y su aritmética correspondiente: cada nombre de variable tiene que ser declarado, y conlleva una serie de operaciones aritméticas y lógicas asociadas a él, por ejemplo:

```
int i;
char c;
```

- comparaciones y bucles: existen elementos de repetición como switch, while y for, que repiten un grupo de sentencias hasta que se cumple determinada condición o igualdad,

- posee punteros y arrays para gestionar conjuntos de objetos y su acceso desde la memoria.

El siguiente paso en los lenguajes de programación es la programación modular. En este caso, descomponemos el programa en distintos módulos, funciones o subrutinas (según el lenguaje de programación utilizado) que procesan las variables de nuestro programa basándose en algoritmos diseñados previamente para obtener el resultado deseado. Con estos módulos obtenemos:

- ocultación de datos, que facilitan la descomposición del código y su reutilización;

- compilación por separado del código y la creación de bibliotecas; y

- manejo de excepciones para el procesamiento de errores.

Con las características mencionadas anteriormente, C++ no pasaría de ser una réplica de C, con algunas mejoras. Sin embargo, C++ va un paso más allá, implantando la abstracción de datos. Con ello, el paradigma de la POO puede ser realizado bajo C++ de manera eficiente, facilitando tipos

definidos por el usuario (clases), también conocidos como tipos de datos abstractos. Una clase no es más que una abstracción de un objeto que el programador realiza a partir de un modelo que representa un elemento de la realidad. Una clase contiene no sólo sus elementos, sino que también contiene el interfaz con el que el objeto interacciona con otros objetos del « mundo».

La introducción de las clases facilita la introducción de otros conceptos inherentes a la POO como son la herencia y el polimorfismo. Como se verá en los siguientes capítulos, C++ posee los mecanismos que hacen de este lenguaje uno de los más potentes que existen en la actualidad.

La Biblioteca Estándar de C++ STL, (Standard Template Library)

La biblioteca o librería estándar de C++, conocida como STL (Standard Template Library) es una adición relativamente reciente al C++ (data de su último proceso de estandarización en 1998). Tras la introducción de la STL, C++ ha alcanzado una gran madurez, ya que facilita al programador un conjunto de tipos de datos y algoritmos de gran potencia. Se estudiará con cierto detalle más adelante, pero vamos a ver brevemente algunas de sus características. Esta introducción permitirá realizar ejemplos y ejercicios con cierto sentido a lo largo de las siguientes lecciones.

Cada implantación de C++ viene complementada por un conjunto de funciones que facilitan la creación de Interfaces Gráficos de Usuario (GUIs, Graphical User Interfaces). Al contrario de lo que ocurre con Java con las clases Swing o AWT, la parte del GUI no forma parte del estándar del lenguaje C++. Así, existen distintas bibliotecas para crear GUIs en Linux (Qt, GTK++), en UNIX (Motif), y en PC's (MFC). En esta parte del curso nos centraremos en el estándar de C++, dejando la discusión sobre Interfaces de usuario para la siguiente parte dedicada a Microsoft Visual C++.

Hola Mundo

Como prometimos, vamos a introducir nuestro primer programa en C++, con nuestra versión del conocido Hola Mundo! El listado del programa aparece en el Código fuente 74.

```
#include <iostream> // ejemplo 1

int main()
{
  // imprime en pantalla el saludo Hola Mundo!
  std::cout << "Hola Mundo!\n";
}
```

Código fuente 74

Vamos a analizar cada parte del programa. Cada programa ejecutable de C++, al igual que en C, tiene que tener una función main. El programa comienza ejecutando esta función. El valor entero (int) devuelto por main es capturado por el sistema operativo. Si no se devuelve ningún valor o se devuelve 0, el sistema interpreta que el programa se ha ejecutado con éxito. Las llaves { ... } indican agrupamiento en C++. En este caso agrupa el contenido de la función main. El alumno observará una gran similitud entre C y C++. Como ya hemos señalado no es accidental, y responde al deseo de mantener compatibilidad « hacia atrás» con código escrito en C.

La línea #include <iostream> indica al compilador que incluya la biblioteca estándar de entrada/salida de C++. Estos archivos se denominan en C y C++ archivos de encabezado, que incluyen las declaraciones de las funciones, y que tradicionalmente en C eran archivos con la extensión ".h". En C++, la biblioteca estándar no suele incluir extensión en los nombres de los archivos, aunque es tradicional que los archivos creados por el usuario sí la lleven. Observe que el nombre de la biblioteca está rodeado de los signos <...>. Observe también que en C++

- cada sentencia es terminada por un punto y coma ";"
- los espacios en blanco entre los elementos de las sentencias carecen de importancia
- // indican un comentario hasta el final de la línea; los comentarios « multi-línea» son como en C: /* .... */

La única línea del programa imprime en pantalla el saludo «Hola Mundo» seguido de un salto de línea (\n). El operador << envía su segundo argumento al primero, la pantalla en este caso, representada por el elemento de la biblioteca estándar iostream cout. Se observa que los elementos y funciones de la biblioteca estándar van precedidos de «std::». En este caso, std define el espacio de nombres (namespace, en lo que sigue) de esta biblioteca. Los namespaces son una técnica de C++ para agrupar interfaces y elementos con ciertos aspectos comunes. Si se incluye la línea que vemos en el Código fuente 75, después de las sentencias #include, no se necesitará incluir el prefijo std:: en los elementos de la biblioteca estándar. Usaremos esta técnica en los ejemplos de este curso, para acortar la longitud de los ejemplos, pero no recomendamos su utilización en ejercicios reales.

```
  using namespace std;
```

Código fuente 75

La salida por pantalla

Como ya hemos visto en nuestro primer ejemplo, la salida por pantalla o consola se realiza a través del elemento cout. Básicamente, cout convierte cualquier variable que se le envíe en una cadena apta para ser visualizada en la pantalla.

```
#include <iostream> // ejemplo 2
using namespace std;
int main()
{
  int i = 2;

  cout << "Hola";
  cout << ',';
  cout << " como estas? ";
  cout << i << '\n';
}
```

Código fuente 76

Se observa que el operador << acepta cualquier tipo de variable, y que se pueden concatenar varias variables con elementos << contiguos.

Cadenas en C++

Un elemento nuevo dentro de la biblioteca estándar de C++ son las cadenas de caracteres (strings), definidas en la biblioteca <string>. Las cadenas de caracteres que existían con anterioridad, se reducían a las cadenas tipo C, que consistían en un array de caracteres, terminados por el elemento 0. Veamos en el Código fuente 77 un ejemplo.

```
#include <iostream> // ejemplo 3
#include <string>
using namespace std;
int main()
{
  string s1 = "Hola ", s2;
  s2 = "carlos";
  string s = s1 + s2;
  cout << s1 << s2 << '\n';
  cout << s << '\n';
  s += '\n';
  cout << s;
}
```

Código fuente 77

En este ejemplo introducimos diversas propiedades de las cadenas. Una característica importante de C++ es que, al contrario que en otros lenguajes, las variables pueden ser declaradas en cualquier parte del código, por ejemplo, inmediatamente antes de su utilización, como ocurre en el ejemplo con la cadena s. Esta flexibilidad en la declaración de variables, denostada por muchos, debe utilizarse con moderación.

Vemos que las cadenas se pueden concatenar con el operador +. Además, se pueden añadir caracteres al final de una cadena con el operador +=, que equivale a:

```
s = s + '\n';
```

Finalmente, queremos señalar que las cadenas en C++ no pueden ser utilizadas en funciones de C, como printf. Sin embargo, las cadenas de C++ poseen una función miembro de su clase (string.c_str()), que transforma la cadena en una cadena de C.

```
#include <stdio.h> // ejemplo 4
#include <string>
using namespace std;
int main()
{
  string s1 = "Hola carlos\n";
  printf("%s", s1.c_str());
}
```

Código fuente 78

En el Código fuente 78 puede verse como hemos incluido en este caso la biblioteca estándar de C <stdio.h>, que declara printf.

La entrada por teclado

El elemento de <iostream> que acepta entradas del usuario por teclado se denomina cin. En este caso, el operador de entrada es >>. El Código fuente 79 lee el nombre del usuario.

```
#include <iostream> // ejemplo 5
#include <string>
using namespace std;

int main()
{
  string nombre;
  cout << "Introduce tu nombre: ";
  cin >> nombre;
  cout << "Bienvenido " << nombre << '\n';
}
```

Código fuente 79

El alumno podrá comprobar que si introducimos en el ejemplo anterior el nombre y los apellidos, el sistema sólo captura el nombre. Esto es debido a que el espacio en blanco actúa como separador de elementos. Para leer una línea entera hasta el retorno de carro hay que usar la función

```
getline(cin, nombre);
```

Otros elementos de la biblioteca estándar

Existen otros elementos muy importantes de la biblioteca estándar que no vamos a ver en detalle ahora. En particular, podemos destacar los contenedores (containers), los algoritmos para manipular los datos, y los iteradores (iterators) para « navegar» a través de los contenedores. Dentro de estos contenedores existen vectores (vector), listas (list), listas asociadas (map), y otros tipos que veremos más adelante.

Brevemente, un vector es una serie de elementos consecutivos que ocupan una zona de memoria. En el Código fuente 80 vamos a ver la creación de una guía telefónica elemental a partir de un vector de estructuras.

```
#include <iostream> // ejemplo 6
#include <string>
#include <vector>
using namespace std;

struct Entrada { string nombre; int tel; };

int main()
{
   vector<Entrada> guiaTel(100);
   for(int i = 0; i <= 5; i++)
   {
```

```
      cout << "Nombre y telefono: ";
      cin >> guiaTel[i].nombre >> guiaTel[i].tel;
   }
   for(int i = 0; i <= 5; i++)
   {
      cout << guiaTel[i].nombre << '+' << guiaTel[i].tel << '\n';
   }
}
```

Código fuente 80

En este ejemplo, definimos primero una estructura de datos «Entrada», constituida por una cadena de caracteres y un entero que representa un número de teléfono. A continuación, tenemos un vector de «Entradas» que denominamos guiaTel, y al que asignamos un tamaño de 100. El resto del programa consiste en dos bucles for, basados en un contador i. El lector puede observar que el contador se inicia por separado en cada bucle, ya que al declararse el entero i dentro del propio bucle, su alcance se reduce a ese bucle (NOTA: si por ejemplo, imprimiésemos el valor de i entre los dos bucles for, algunos compiladores lanzan un warning, y permiten el uso de i fuera del bucle, sin embargo, su uso debería conducir a un error de compilación. Este es el comportamiento definido por defecto para el estándar. Microsoft Visual C++ no considera el bucle for como un ente separado, por lo que el ejemplo no compilará a menos que se retire int, del segundo bloque).

En C++ un bucle for tiene esta sintaxis:

```
for (contador = valor inicial; comparación; incremento_contador)
{
   sentencias_a_ejecutar;
}
```

En nuestro ejemplo, el contador i se incrementa con el operador ++, que equivale a incrementar en 1 el valor del contador. Se observa, que al igual que en C, los arrays de elementos inician su contador en 0. Desde el punto de vista de la gestión de memoria, la utilización de vector para un ejemplo de estas características es poco eficiente porque hemos asignado un tamaño fijo de partida (100). Aunque es posible asignar memoria dinámicamente a los vectores (se verá más adelante), existen otro tipo de estructuras más eficientes para elementos de estas características, como listas, e incluso listas asociadas. Para estos elementos, el concepto de índice no existe, y se introduce el concepto de iterador que equivale a un puntero que se desplaza a lo largo de la estructura.

En este capítulo hemos descrito algunos elementos básicos de C++, y hemos presentado algunos ejemplos elementales de programación. Los siguientes capítulos desarrollarán estos temas con profundidad.

Ejercicios

1. Recomendamos al lector la lectura del capítulo «Introducción al entorno de Microsoft Visual C++» para poder realizar los ejemplos y ejercicios descritos en los siguientes capítulos.

2. Rescribir el ejemplo 5 de forma que el programa solicite y lea los nombres y apellidos del alumno.

3. Escribir un programa que lea el año de nacimiento del alumno y lo presente en pantalla.

4. Modificar el ejemplo 6 para que pueda leer no sólo el nombre de la persona, sino también sus apellidos. (Nota: no utilizar getline).

5. Escribir los programas de conversión de grados Celsius a Fahrenheit del capítulo 1 del curso de C en C++.

Respuestas a los ejercicios

2. Rescribir el ejemplo 5 de forma que el programa solicite y lea los nombres y apellidos del alumno.

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
  string nombre;
  cout << "Introduce tu nombre y apellidos: ";
  getline(cin, nombre);
  cout << "Bienvenido " << nombre << '\n';
}
```

3. Escribir un programa que lea el año de nacimiento del alumno y lo presente en pantalla.

```
#include <iostream>
using namespace std;

int main()
{
  int agno;

  cout << "Escribe tu año de nacimiento: ";
  cin >> agno;
  cout << "Naciste en " << agno << '\n';
}
```

4. Modificar el ejemplo 6 para que pueda leer no sólo el nombre de la persona, sino también sus apellidos. (Nota: no utilizar getline).

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

struct Entrada {
  string nombre;
  string apellido;
  int tel;
};

int main()
{
  int i;
  vector<Entrada> guiaTel(100);
  for (i = 0; i <= 5; i++)
  {
```

```
    cout << "Nombre, apellidos y telefono:\n";
    cin >> guiaTel[i].nombre >> guiaTel[i].apellido
        >> guiaTel[i].tel;
  }
  for (i = 0; i <= 5; i++)
  {
    cout << guiaTel[i].nombre << " " << guiaTel[i].apellido << " -> "
         << guiaTel[i].tel << '\n';
  }
}
```

5. Escribir los programas de conversión de grados Celsius a Fahrenheit del capítulo 1 del curso de C en C++.

```
#include <iostream>
using namespace std;

int main()
{
  int LOWER = 0, UPPER = 300, STEP = 20;
  int fahr; /* Declaración de variables */
  for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
  {
    cout << fahr << "\t"
         << (5.0 / 9.0) * (fahr - 32.0) << "\n";
  }
  return 0;
}
```

8

Elementos fundamentales del lenguaje

Tipos y declaraciones

En C++ cada nombre (o identificador) tiene que tener un tipo de dato asociado. No se pueden definir variables sin su tipo asociado. Para cada tipo, existen unas operaciones aritméticas y lógicas asociadas con un comportamiento definido. Ejemplos típicos los vemos en el Código fuente 81.

```
bool a = true;
float b, c = 2.0, d = 1f;
char a = 'f';
```

Código fuente 81

El alumno debe ser consciente de que en C++, el tamaño (número de bits que el sistema asigna a cada tipo) de los tipos intrínsecos del lenguaje depende de la implantación e incluso el compilador que se esté utilizando (rizando el rizo, se dan casos de dependencia con las distintas versiones de un mismo compilador). Por ello, para asegurar una portabilidad adecuada del código, se recomienda estudiar en detalle la implantación de estos tipos en el compilador usado. Esta dependencia contrasta con la constancia en el tamaño de los tipos impuesta por otros lenguajes, como Java. El hecho de que C++ no imponga tamaños prefijados se debe únicamente a mantener la compatibilidad con código C más antiguo.

Tipos

Existen varios tipos fundamentales en C++, que además son una herencia directa de C.

- Tipos intrínsecos enteros
  - booleano (bool): es un tipo lógico que puede cualquiera de los dos valores, true (verdadero) o false (falso). Las variables booleanas se pueden utilizar en expresiones aritméticas, convirtiéndose en enteros. El valor true equivale a 1, y el valor false a 0. El proceso inverso es también cierto, tomándose cualquier entero no nulo en true, y false en caso contrario. Estos hechos se comprueban en el Código fuente 82.

```
#include <iostream> // ejemplo 1
using namespace std;

int main()
{
   bool a = true, b = false;
   cout << "a (true) = " << a << " - b (false) " << b << endl;
   int i = 2;
   cout << "(i+a) = " << i + a << " - (i+b) " << i + b << endl;
}
```

Código fuente 82

El alumno puede observar el uso de una nueva entidad equivalente al salto de línea: endl.

  - caracteres (char): tipo que define un único carácter. La utilización de los caracteres en C es limitada. La implantación más genérica de char es de 8 bits (1 byte). Esto indica en general que sólo se pueden representar 256 caracteres, definidos a partir del estándar ISO-646. En la mayoría de las implantaciones se utiliza el conjunto de caracteres ASCII, que no incluyen caracteres típicos del idioma español, como las vocales acentuadas o la letra ñ. Por otro lado, en la representación de cualquier binario, podemos considerar el primer bit como signo. Para ello, existen los prefijos signed o unsigned char, que definen directamente el carácter. Para la representación de conjuntos de caracteres más genéricos como UNICODE (que, por ejemplo, se incorpora por defecto en Java), existe el tipo wchar_t. Cada carácter tiene asociado un entero, como se comprueba en el Código fuente 83.

```
#include <iostream> //ejemplo 2
using namespace std;

int main()
{
   char c = 't';
   cout << "caracter c: " << c << " " << int(c) << endl;
}
```

Código fuente 83

  - enteros (int): tipo que representa un número entero. Al igual que ocurría en el caso anterior, existe la posibilidad de que tengan signo o no. En ese caso, se utilizan también los prefijos signed o unsigned. Los números enteros, presentan además otra peculiaridad,

conservada con compatibilidad con C. Pueden ser de tres tipos: short int (short), int, o long int (o long). Las tres variantes pueden ser iguales, o tener un número distinto de bits. Un entero de tipo long se puede representar con la cadena L al final del número.

```
long i = 4L;
```

Además existe la posibilidad de representar enteros en base octal o hexadecimal. Los primeros son enteros que comienzan con '0', y los segundos son enteros que terminan en 'x' o en 'X'. Por ejemplo, los números:

```
int a = 045, b = 43x, c = 4eX;
```

son perfectamente válidos. En la representación hexadecimal se utilizan los caracteres a, b, c, d, e, f (o sus equivalentes en mayúsculas) para denotar 10, 11, 12, 13, 14 y 15. En cualquiera de los literales (L, 0, X), nunca deben existir espacios en blanco entre el número y el propio literal.

- Tipos intrínsecos aritméticos

Estos tipos corresponden a los números de coma flotante (números reales), y se reducen a dos tipos, float y double. Una variable float se distingue de una variable double, en el tamaño de palabra de memoria asignada, mayor para este último. Para la representación de potencias de 10 (muy comunes en los números reales), se utiliza el literal 'e' (o 'E'). Por ejemplo:

```
float f = 4.2e4;
double d = 1.03E-3;
```

equivalen a $4.2x10^4$, y $1.03x10^{-3}$, respectivamente. También existe el literal 'f' o 'F' como sufijo de float.

Para cualquiera de los tipos intrínsecos, es conveniente determinar los detalles de implantación del compilador. Para ello, C++ incluye el operador sizeof. El Código fuente 84 descubre los tamaños de los enteros para nuestra implantación.

```
#include <iostream> // ejemplo 3
using namespace std;

int main()
{
   short s = 0;
   int c = 4;
   long a = 33300300L;
   cout << "El tamaño de un short es: " << sizeof s << endl;
   cout << "El tamaño de un entero es: " << sizeof c << endl;
   cout << "El tamaño de un long es: " << sizeof a << endl;
}
```

Código fuente 84

- Tipos void

Este tipo nunca se puede asignar a una variable. No existe una variable «vacía». Únicamente se puede asignar a una función que no devuelve ningún valor, por ejemplo:

```
void ff (int a, int b) { ... }
```

o en un puntero que recogerá el valor de una variable cuyo tipo no se conoce en ese momento,

```
void *p;
```

y p podrá apuntar más adelante a un int, a un float, etc. El comportamiento es análogo al ANSI C.

- Tipos definidos por el usuario

  Estos tipos son básicamente dos: enum y class. Del segundo tipo nos ocuparemos en detalle en otro capítulo. Un elemento enum es un conjunto de elementos definidos con un nombre:

  ```
  enum coche { PUERTA, RUEDA, VOLANTE };
  ```

  En este caso, PUERTA, RUEDA y VOLANTE son constantes enteras de valor 0, 1 y 2, respectivamente, y que se pueden utilizar en operaciones lógicas y aritméticas como cualquier otro entero. Se les puede asignar cualquier otro valor por defecto, mediante asignación directa:

  ```
  enum saludo { HOLA = 9, ADIOS };
  ```

  que asigna a HOLA el valor 9, y a ADIOS el valor 10. Recomendamos al alumno la lectura detallada de la ayuda relacionada con el tipo **enum** en el compilador.

Declaraciones

La forma genérica de las declaraciones en C++ es:

```
[especificador] tipo [declarador] nombre_variable [= valor_inicial];
[especificador] tipo [declarador] nombre_función(variables)
   { cuerpo de función }
```

donde los corchetes implican elementos opcionales. El especificador es una palabra clave que modifica el comportamiento de la variable. Puede ser const, virtual o extern. const implica que la variable es una constante que no se puede modificar durante la ejecución del programa. Los otros dos especificadores se estudiarán más adelante. tipo puede ser cualquiera de los tipos analizados en la sección anterior. Los declaradores son los siguientes:

- * puntero (prefijo)
- *const puntero constante (prefijo)
- & referencia a dirección de memoria (prefijo)
- [] array (sufijo)
- () función (sufijo)

En el Código fuente 85 presentamos algunos ejemplos.

```
extern a = 4; // no válido, falta el tipo
int b = 4, *a;
int f(int a) { return 2 * a };
```

```
   int h(double z, float b); // válido, el cuerpo de h se define en otro archivo
   long z; // válido, equivalente a long int z;
```

Código fuente 85

Declaración múltiple de variables

En C++ se pueden declarar varias variables del mismo tipo en la misma sentencia, separando los nombres con comas.

```
   int* p, y; // Equivale a: int *p; int y;
   int p, *y, z; // Equivale a: int p; int *y; int z;
   int v[10], *p, y = 4; // Equivale a: int v[10]; int *p; int y = 4;
```

Código fuente 86

Observe que el modificador * en la primera línea afecta sólo a p, y no a y.

Nombres de variable en C++

Los nombres en C++ están compuestos por cualquier secuencia de caracteres y dígitos, siendo la primera letra siempre un carácter alfabético o el carácter subrayado «_». Mayúsculas y minúsculas son significativas en C++. El número de caracteres máximo es dependiente del sistema. Se desaconseja la utilización del carácter «$» en nombres de variables, aunque aceptado por algunos compiladores, porque no conduce a programas portables. Nunca se puede usar una palabra clave como nombre de variable. Así, son aceptables nombres como:

```
   mes0a   variaBLE   a_d_i_o_s   nombre_de_variable_largo   CLASS
 _2Funcion
```

No son aceptables nombres como:

```
   1x   class   a-b_c   (a   ñ
```

Alcance de las variables

En C++, las variables tienen distintos alcances: global o local. El alcance viene determinado por su declaración, y por la pareja de delimitadores { . . . } donde se ha declarado. Las variables globales se definen fuera previamente a main.

```
#include <iostream> // ejemplo 4
using namespace std;

double a = 4.2; // a global

int main()
{
   cout << "a global = " << a << endl;
   {
      double a; // a local
```

```
      a = 2.0;
      cout << "a local = " << a << endl;
   }
   cout << "a global, de nuevo = " << a << endl;
}
```

Código fuente 87

Existe el operador de resolución de alcance, ::, que permite utilizar variables globales que entran en conflicto con variables locales.

```
#include <iostream> // ejemplo 5
using namespace std;

double a = 4.2; // a global

int main()
{
   cout << "a global = " << a << endl;
   {
      double a; // a local
      a = 2.0;
      cout << "a local = " << a << endl;
      a = a + ::a;
      cout << "a local = " << a << endl;
   }
}
```

Código fuente 88

También se pueden utilizar namespaces para definir variables.

```
#include <iostream> // ejemplo 6
using namespace std;

namespace a1 { double x = 3.0; }
namespace a2 { double x = -2.0; }

int main()
{
   cout << "a1::x global = " << a1::x << endl;
   cout << "a2::x global = " << a2::x << endl;
}
```

Código fuente 89

Otro aspecto importante es la iniciación por defecto de las variables. Variables globales, definidas en namespaces, u objetos locales estáticos se inician con el valor 0 del correspondiente tipo. Variables locales (objetos dinámicos) no tienen ningún valor por defecto.

```
#include <iostream> // ejemplo 7
using namespace std;

namespace a1 { double x; }
```

```
void f()
{
   int j;
   cout << "j (absurdo) = " << j << endl; // valor absurdo
}

int main()
{
   int i;
   cout << "i = " << i << endl; // cero
   cout << "x global = " << a1::x << endl; // cero
   f();
}
```

Código fuente 90

Typedef

El modificador typedef se utiliza para declarar nuevos tipos de variable a partir de otro. Ese nuevo tipo se puede usar en posteriores declaraciones de variables. Por ejemplo, para definir una nueva variable que sea un puntero a carácter, usamos:

```
typedef char* PuntChar;
PuntChar pc1, pc2;
```

Punteros, Arrays y Estructuras

Los punteros, arrays y estructuras son elementos comunes del lenguaje C y de C++. Son elementos que representan agrupaciones (arrays y estructuras), y cómo navegar a través de ellas. Estos elementos han sido utilizados con profusión en programas, cuando no existían otros elementos de función equivalente. También se ha aprovechado la potencia de estos elementos para realizar código muy complejo, de difícil comprensión, y aún peor mantenimiento. Nosotros recomendamos utilizar elementos equivalentes de la biblioteca estándar (<vector>) para los arrays, y clases (class) para las estructuras. Los compiladores actuales implantan estos elementos con tanta eficiencia como lo puedan hacer con un array, pero con herramientas auxiliares potentes.

Punteros¡Error! Marcador no definido.

Un puntero es un tipo de variable que apunta a una dirección de memoria. Se utiliza para apuntar a otros objetos. La Figura 2 se aplica al Código fuente 91.

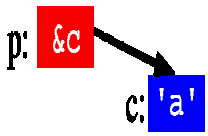


Figura 2

```
char c = 'a';
char *p = &c;
```

Código fuente 91

En este caso, definimos una variable de tipo carácter, c, al que asignamos el carácter 'a'. En la segunda línea, mediante el operador de dereferencia, asignamos al puntero a carácter p, la dirección de memoria de c. Tenemos pues, los siguientes elementos:

- T* puntero a tipo T
- &var dirección de memoria de variable var (operador de dereferencia)

```
#include <iostream> // ejemplo 8
using namespace std;

int main()
{
  char c = 'a';
  char *p = &c;

  cout << "Caracter c: " << c << endl;
  cout << "Puntero p (hexadecimal): " << hex << long(p) << endl;
  cout << "Contenido de p: " << *p << endl;
}
```

Código fuente 92

En la impresión por pantalla observamos que el valor de p se corresponde con una dirección de memoria, que conviene convertir en un número long por cuestiones de implantación de las direcciones de memoria (el prefijo hex indica número hexadecimal). Para obtener el contenido de p, basta con usar el operador *. Incluimos otros ejemplos en el Código fuente 93.

```
  char** ppc; // puntero a puntero de chars
  int* ap[15]; // array de 15 punteros a int
  int (*fp)(char *); // puntero a función que devuelve un entero
  int *fp(char *); // función que devuelve un puntero a int
```

Código fuente 93

En C es tradicional que el puntero nulo se represente con la macro (constante) NULL. En C++ no es necesaria esta equivalencia, ya que la conversión automática de tipos facilita que el puntero nulo se puede representar con 0 (cero).

Arrays

Un array es una colección de elementos del mismo tipo.

```
  tipo nombre[enteroConstante];
  tipo nombre[] = { conjunto de valores separados por comas };
  tipo nombre[enteroConstante] = { conjunto de valores separados por comas };
```

Código fuente 94

Cada elemento es accesible por su índice, desde 0 hasta enteroConstante-1, con la notación nombre[índice]. Ejemplos típicos aparecen en el Código fuente 95.

```
   int x[3];
   char a[] = { 'a', 'b', 'c' }; // a es un array de 3 elementos,
                                 // a[0] = 'a', etc.
   float x[2] = { 1.2, 2.1, 3.2 }; // error, no se pueden iniciar más de dos
elementos
   int y[3] = { 34, -10 }; // correcto, y[0] = 34, y[1] = -10, y[2] = 0 (valor por
defecto)
```

Código fuente 95

Unos arrays especiales son las cadenas de caracteres, que terminan con el carácter '\0'. Así:

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

significa que s es una cadena de 5 caracteres: "hola" + '\0'.

Punteros y arrays: aritmética de punteros

Los punteros y arrays están estrechamente ligados en C++ (al igual que se vio en C). Veamos su relación con el ejemplo del Código fuente 96.

```
#include <iostream> // ejemplo 9
using namespace std;

int main()
{
  char c[] = "veronica"; // 9 caracteres
  char *p = c; // p asociado al comienzo de c
  cout << "caracter c[0]: " << c
       << " - direccion c[0]: " << hex << long(&c[0]) << endl;
  cout << "puntero p: " << hex << long(p)
       << " - caracter en *p: " << *p << endl;
  ++p;
  cout << "puntero p+1: " << hex << long(p)
       << " - caracter en *(p+1): " << *p << endl;
}
```

Código fuente 96

Se observa en el ejemplo como se puede asignar el puntero p al principio de la cadena c, porque al declarar un array de cualquier tipo c[N], c es simplemente el puntero a la dirección de memoria del primer elemento c[0]. El ejemplo también ha introducido la conocida y "peligrosa" aritmética de punteros. Las siguientes operaciones están definidas:

- Diferencia entre punteros del mismo array
- Suma con un entero
- Diferencia con un entero

El peligro inherente en las operaciones con punteros, es que en el ejemplo 9, podríamos haber introducido tras su definición de p la sentencia: p = p-1; que no daría ningún error de compilación, pero que daría lugar a un puntero a una dirección de memoria existente, pero posiblemente ocupada por otra sección del programa. La manipulación incorrecta de punteros es uno de los errores más frecuentes en los programas escritos en C y C++.

Constantes

Es frecuente encontrarnos con variables cuyos valores no deseemos que sean modificados durante la ejecución de un programa. Para ello, C++ introduce el especificador const. La sintaxis es la siguiente:

```
const tipo variable;
```

Referencias

Las referencias proporcionan nombres alternativos para objetos. Debemos destacar que las variables tienen que ser inicializadas con la referencia de la variable a la que apuntan.

```
#include <iostream> // ejemplo 10
using namespace std;

int main()
{
  double f = 1.2;
  double &r = f; // r y f son equivalentes
  r = 2.0; // modificamos r y f

  cout << "r = " << r << " f = " << f << endl;
}
```

Código fuente 97

Se observa que una vez definida la referencia, los nombres apuntan al mismo espacio de memoria, y cualquier modificación en una afecta a la otra.

Otro uso que veremos más adelante es utilizar referencias para el paso de variables a funciones por referencia, y no por valor.

Estructuras

Las estructuras son agrupaciones de elementos de distinto tipo. Son una herencia directa de C, en el que representaban la agrupación de datos más sofisticada. Sin embargo, en C++ existen elementos más sofisticados como son las clases, que veremos en otro capítulo, y cuyo uso recomendamos sobre estos elementos. La declaración típica es la que muestra el Código fuente 98.

```
  struct registro {
     char *nombre;
     char *direccion;
     long telefono;
```

```
      int cod_postal;
      char *email;
   };
```

Código fuente 98

Observe que la definición de una estructura es uno de los pocos casos en C++ donde un grupo de llaves { ... } debe ser terminado en punto y coma ';'. Olvidar este punto y coma es una fuente frecuente de errores. Una vez definida la estructura se pueden declarar variables como un tipo más del lenguaje. Los elementos de la estructura son accedidos mediante el operador '.':

```
   registro agenda[10];
   agenda[0].nombre = "Carlos";
   agenda[0].telefono = 900001001;
```

Finalmente, se pueden definir punteros a estructuras. En este caso, el operador de dereferencia para acceder a los elementos de la estructura es: ->

```
#include <iostream> // ejemplo 11
using namespace std;

struct saludo {
  char *bienvenida;
  char *despedida;
};

int main()
{
  saludo misaludo;
  misaludo.bienvenida = "hola";
  misaludo.despedida = "adios";

  cout << "saludo.bienvenida: " << misaludo.bienvenida << endl
       << "saludo.despedida: " << misaludo.despedida << endl;
}
```

Código fuente 99

El alumno puede observar que para usar estructuras definidas en el programa principal no es necesario el uso de la palabra clave struct como en C.

Expresiones y sentencias

Operadores

La Tabla 7 muestra la mayor parte de los operadores típicos en C++. El comportamiento es totalmente análogo a los operadores en C, por lo que no nos detendremos en detalles. En particular no incidiremos sobre los operadores relacionados con la manipulación de bits, para los que recomendamos la consulta de la ayuda de Microsoft Visual C++. Observe que sizeof es un operador y no una función como en C.

Operadores aritméticos y lógicos	
Selección de miembro (estructuras, clases)	objeto.miembro
Selección de miembro (estructuras, clases)	objeto->miembro
Post-incremento	variable++
Post-decremento	variable--
Tamaño de objeto	sizeof objeto
Tamaño de tipo	sizeof tipo
Pre-incremento	++variable
Pre-decremento	--variable
Complemento a uno	~expresión
No lógico	!expresión
Menos unario	-expresión
Positivo unario	+expresión
Dirección de	&variable
Dereferencia	*expresión
Crear (asignar memoria)	new tipo
Crear (asignar memoria e inicializar)	new tipo { expresiones; }
Destruir	delete puntero
Destruir array	delete[] puntero
Conversión de tipo (cast)	(tipo) expresión
Selección de miembro	objeto.*puntero_a_miembro
Selección de miembro	objeto->*puntero_a_miembro
Multiplicar	expresión * expresión
Dividir	expresión / expresión
Resto en división	expresión % expresión
Sumar	expresión + expresión

Restar	expresión - expresión
Menor que	expresión < expresión
Menor o igual que	expresión <= expresión
Mayor que	expresión > expresión
Mayor o igual que	expresión >= expresión
Igualdad	expresión == expresión
Desigualdad	expresión != expresión
Y lógico	expresión && expresión
O lógico	expresión || expresión
Asignación simple	variable = expresión
Multiplicar y asignar	variable *= expresión
Dividir y asignar	variable /= expresión
Resto y asignar	variable %= expresión
Sumar y asignar	variable += expresión
Restar y asignar	variable -= expresión
Lanzar excepción	throw expresión
Coma (secuencia)	expresión1, expresión2

Tabla 7

Los resultados de las operaciones obedecen las reglas aritméticas habituales. En particular, no existe un orden predeterminado para la evaluación de expresiones de igual prioridad. Por ejemplo, la sentencia:

```
v[i] = i++;
```

se ejecutará en unos compiladores como: v[i] = i; i++; y en otros como: i++; v[i] = i; . Desaconsejamos totalmente la escritura de código que dependa de un posible orden en la ejecución.

La precedencia de operadores sigue el orden de asociatividad lógico de todos los lenguajes de programación. Así expresiones como:

```
if(f <= i || g >= j) { ... }
```

serán equivalentes a if((f < i) || (g >= j)) { ... }. En cualquier caso, siempre es recomendable la utilización de paréntesis para evitar efectos no-deseados.

Los operadores incremento o decremento (++, --), en sus versiones prefijo (es decir, incrementar o restar antes de ejecutar la sentencia), o sufijo (es decir, incrementar o restar después de ejecutar la sentencia) permiten escribir código de forma muy compacta. Así, por ejemplo, para copiar dos cadenas de caracteres de N elementos, bastaría con escribir:

```
while ((*p++ = *q++) != 0);
```

si recordamos que toda cadena termina en el carácter nulo.

Los operadores new y delete sirven para asignar memoria. Debemos recordar que se debe usar delete (o delete[] para arrays) para recuperar la memoria asignada a cualquier objeto mediante new:

```
char *s = new char[30];
```

Es importante reseñar que en C++ no existe un "recogedor de basura" (garbage collector) equivalente al de Java. Toda la manipulación de objetos es manual, a no ser que se implante un modelo de asignación de memoria con recogedor de basura. La función del Código fuente 100 agotaría la memoria del ordenador en unos cuantos bucles.

```
void f()
{
   try { for (;;) new char[10000]; } // bucle infinito, agota la memoria
   catch (bad_alloc) { cerr << "Se agoto la memoria\n"; } // error en pantalla
}
```

Código fuente 100

El Código fuente 101 sustituye cómo manipula C++ el caso de error en la asignación de memoria mediante new con una función diseñada por el usuario mediante la función set_new_handler. Recomendamos ejecutar este programa con prudencia, ya que puede dar lugar en algunos casos a que el ordenador no responda.

```
#include <iostream> // ejemplo 12
#include <new>
using namespace std;

void sin_memoria()
{
  cerr << "Error con operador new, sin memoria\n";
  throw bad_alloc();
}

int main()
{
  set_new_handler(sin_memoria);
  for (;;) new char[10000];
  cerr << "fin\n";
}
```

Código fuente 101

Constructores

Para cada tipo T, existe el constructor T(otro_tipo x), que crea una variable del tipo T a partir de una variable de otro tipo. Veamos, por ejemplo, el Código fuente 102.

```
int i = 0;
double x = double(i); // x = 0.0
```

Código fuente 102

Este tipo de conversiones se debe usar con prudencia, ya que a veces la conversión no está definida. Una declaración del tipo

```
int j = int();
```

indica que la variable se inicia al valor por defecto del tipo correspondiente, generalmente 0. Este tipo de construcciones serán frecuentes para las clases.

Declaraciones o Sentencias

Queremos destacar que en C++, al contrario de lo que ocurría en C, podemos incluir declaraciones dentro del propio código, es decir, no existe diferenciación entre declaraciones o sentencias. Este hecho es denostado por algunos programadores acostumbrados a semánticas más rígidas, pero por otro lado, permite optimizar el código, al introducir variables allá donde son necesarias, sin tener que asignar espacio de memoria a variables en otros puntos del código. En este punto, debemos advertir al usuario que Microsoft Visual C++ no se adhiere al estándar en este caso de forma rígida en contadores declarados dentro de un bucle for.

Sentencias de selección

En C++ disponemos de las siguientes sentencias de selección: if ... else, y switch.

If

La definición de una sentencia if es la siguiente:

```
if(condición1) { sentencias; }
else if(condición2) { sentencias; }
...
else { sentencias; }
```

El programa comprueba la condición-1. Si es cierta (true), ejecuta el conjunto de sentencias asociado a ella, y sale del grupo. Si no es cierta (false), se comprueba la condición-2. Si es cierta, se ejecutan sus sentencias asociadas, y si es falsa, se comprueban los "else if" subsiguientes hasta encontrar la sentencia "else", que corresponde a la ejecución por defecto. Tanto else, como los else if son opcionales.

Switch

La selección switch se puede expresar como:

```
switch(valor) {
case constante1:
   sentencias;
   break;
case constante2:
    sentencias;
    break;
...
default:
   sentencias;
   break;
}
```

La gramática es muy similar. Se compara valor con cada una de las constantes. Si coincide con alguna de ellas, se ejecuta el conjunto de sentencias asociadas. Si al final de estas aparece la sentencia opcional break; el programa sale de esta sentencia. Si no existe break, se ejecutará todo el código subsiguiente de switch hasta el final.

Sentencias de iteración

En este caso existen tres tipos: while, do ... while, y for.

While

La sintaxis de un bucle while es:

```
while(condición) { sentencias; }
```

La forma de proceder de un bucle while es la siguiente. Se evalúa la condición una vez. Si es cierta, se ejecuta el conjunto de sentencias asociadas; si es falsa, se pasa a la siguiente sentencia del código, ignorando las sentencias asociadas. En el primer caso, al terminar de ejecutar las sentencias asociadas, se re-evalúa la condición de nuevo. Si es cierta, se ejecutan las sentencias asociadas de nuevo. El conjunto de sentencias asociadas se ejecutará hasta que la condición sea falsa.

Do ... while

Su sintaxis es:

```
do { sentencias; } while(condición);
```

Este caso es muy similar al anterior. Sin embargo, primero se ejecuta el conjunto de sentencias, y después se comprueba la condición. Cuando es falsa, se continúa con el resto del código, y cuando es cierta, se ejecutan sus sentencias de nuevo.

For

El bucle for tiene por sintaxis:

```
   for(condición_inicial; expresión_condición_final;
expresión_iteración)
      { sentencias; }
```

En este caso, el conjunto de sentencias asociadas se ejecuta a partir de una condición inicial, y mientras no se cumpla la condición final, en cuyo caso, se continúa con el resto del código. El valor de iteración corresponde a cómo se modifica la variable de iteración en cada bucle, a partir de su condición inicial.

Goto

En C++, también existe el criticado goto. Su presencia, además de justificarse por criterios de compatibilidad con C, es debida a su posible necesidad en fragmentos de código para su ejecución en tiempo real. No se recomienda su uso salvo casos raramente justificables. Su sintaxis es:

```
goto etiqueta;
...
etiqueta: sentencias;
```

Ejercicios

1. Crear un programa en C++ que imprima todos los códigos ASCII del alfabeto (a-z, A-Z), y los números 0-9.

2. Crear un programa en C++ que asigne distintos enteros en notación octal y hexadecimal. Imprimir esos números con cout. ¿Qué representación obtiene por pantalla, decimal, octal o hexadecimal? Repetir el ejercicio precediendo la impresión con << hex. (Por ejemplo, cout << hex << int;)

3. ¿Qué declaraciones son incorrectas para estos números enteros?

   ```
   a = 4x;
   b = 44gX;
   int j = i + 33 X;
   long k, j = 22 + 3fx;
   ```

4. ¿Qué declaraciones son incorrectas para estos números de coma flotante?

   ```
   f = 4.0 F;
   double x = 1.0330E-8;
   float y = aa + 0.44e+14;
   double x, y, z = 0, a = 1.4e-.2;
   ```

5. ¿Cuáles de los siguientes nombres de variable son legales en C++?

   ```
      aDIOs   443   kilo11   ___NEW___   2dobles   $a44   a444e444
   444444   c______
   ```

6. Averigüe el número máximo de caracteres admitidos como nombre de variable en Microsoft Visual C++

7. Realizar un programa en C++ que imprima por pantalla un array de 10 elementos enteros con un bucle for.

8. Determinar mediante sizeof los tamaños en bytes de los tipos típicos en C++.

9. ¿Cuáles de estas declaraciones son incorrectas?

```
int *p, a, j;
double &y;
char p, *c;
float x; float &y = x;
char const j = 4;
```

10. Mediante la estructura registro definida en el texto, crear una agenda de 10 entradas que pide los datos al usuario, y los imprime a continuación por pantalla (Nota: sustituye char* por string).

11. Comprobar si en C++ existe la equivalencia de tipos. Por ejemplo, sean las estructuras S1 y S2 definidas como:

```
   struct S1 { int i; };
   struct S2 { int j; };
```

¿es correcto el siguiente código?

```
   S1 x;
   S2 y = x;
```

12. ¿Cuál es el valor de i y j tras ejecutar este código?

```
   i = 3;
   j = 2 * i++;
i = 4; j = 8;
B. i = 3; j = 8;
C. i = 4; j = 6;
D. i = 4; j = 9;
```

Respuestas a los ejercicios

1. Crear un programa en C++ que imprima todos los códigos ASCII del alfabeto (a-z, A-Z), y los números 0-9.

```
#include <iostream>
using namespace std;

int main()
{
  int i;

  for (i = 'A'; i <= 'Z'; i++)
```

```
  {
    cout << char(i) << ' ' << i << endl;
  }
  for (i = 'a'; i <= 'z'; i++)
  {
    cout << char(i) << ' ' << i << endl;
  }
  for (i = '0'; i <= '9'; i++)
  {
    cout << char(i) << ' ' << i << endl;
  }
}
```

2. Crear un programa en C++ que asigne distintos enteros en notación octal y hexadecimal. Imprimir esos números con cout. ¿Qué representación obtiene por pantalla, decimal, octal o hexadecimal? Repetir el ejercicio precediendo la impresión con << hex. (Por ejemplo, cout << hex << int;)

```
#include <iostream>
using namespace std;

int main()
{
  int i = 123, j = 0x3a;

  cout << "entero decimal: " << i << endl
       << "entero octal: " << oct << i << endl
       << "entero hexadecimal: " << hex << i << dec << endl;
  cout << "entero decimal: " << j << endl
       << "entero octal: " << oct << j << endl
       << "entero hexadecimal: " << hex << j << dec << endl;
}
```

3. ¿Qué declaraciones son incorrectas para estos números enteros?

```
a = 4x;
b = 44gX;
int j = i + 33 X;
long k, j = 22 + 3fx;
```

Respuesta:

```
b = 44gX;
int j = i + 33 X;
```

4. ¿Qué declaraciones son incorrectas para estos números de coma flotante?

```
f = 4.0 F;
double x = 1.0330E-8;
float y = aa + 0.44e+14;
double x, y, z = 0, a = 1.4e-.2;
```

Respuesta:

```
f = 4.0 F;
double x, y, z = 0, a = 1.4e-.2;
```

5. ¿Cuáles de los siguientes nombres de variable son legales en C++?

```
aDIOs   443   kilo11   ___NEW___   2dobles   $a44   a444e444444
444   c_______
```

Respuesta:

```
443
2dobles
$a44
```

6. Averigüe el número máximo de caracteres admitidos como nombre de variable en Microsoft Visual C++

   - El número máximo de caracteres para una variable o identificador interno o externo en Visual C++ es 247 (se puede buscar en la ayuda introduciendo en el campo de búsqueda naming identifiers).

7. Realizar un programa en C++ que imprima por pantalla un array de 10 elementos enteros con un bucle for.

```
#include <iostream>
using namespace std;

int main()
{
  int i[] = { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

  for (int j = 0; j < 10; j++)
    cout << "i[" << j << "] = " << i[j] << endl;
}
```

8. Determinar mediante sizeof los tamaños en bytes de los tipos típicos en C++.

```
#include <iostream>
using namespace std;

int main()
{
  bool b;
  char c;
  short int i;
  int j;
  long int k;
  float x;
  double y;
  long double z;
  cout << "bool: " << sizeof b << endl;
  cout << "char: " << sizeof c << endl;
  cout << "short int: " << sizeof i << endl;
  cout << "int: " << sizeof j << endl;
  cout << "long int: " << sizeof k << endl;
  cout << "float: " << sizeof x << endl;
  cout << "double: " << sizeof y << endl;
  cout << "long double: " << sizeof z << endl;
}
```

9. ¿Cuáles de estas declaraciones son incorrectas?

```
int *p, a, j;
double &y;
char p, *c;
float x; float &y = x;
char const j = 4;
```

Respuesta:

```
double &y;
char const j = 4;
```

10. Mediante la estructura registro definida en el texto, crear una agenda de 10 entradas que pide los datos al usuario, y los imprime a continuación por pantalla (Nota: sustituye char* por string).

```
#include <iostream>
#include <string>
struct registro {
  string nombre;
  string direccion;
  long telefono;
  int cod_postal;
  string email;
};
int main()
{
  const int REGS = 10;
  int i;
  registro agenda[REGS];
  for (i = 0; i < REGS; i++)
  {
    cout << "Introduce nombre, direccion, telefono, codigo"
         << " postal y correo electronico:" << endl;
    cin >> agenda[i].nombre >> agenda[i].direccion
        >> agenda[i].telefono >> agenda[i].cod_postal
        >> agenda[i].email;
  }
  for (i = 0; i < REGS; i++)
  {
    cout << "Entrada " << i << ": "
         << agenda[i].nombre << ' ' << agenda[i].direccion << ' '
         << agenda[i].telefono << ' ' << agenda[i].cod_postal << ' '
         << agenda[i].email << endl;
  }
  return 0;
}
```

11. Comprobar si en C++ existe la equivalencia de tipos. Por ejemplo, sean las estructuras S1 y S2 definidas como:

```
struct S1 { int i; };
struct S2 { int j; };
```

¿es correcto el siguiente código?

```
   S1 x;
   S2 y = x;
```

Respuesta: NO

12. ¿Cuál es el valor de i y j tras ejecutar este código?

```
   i = 3;
   j = 2 * i++;
i = 4; j = 8;
i = 3; j = 8;
i = 4; j = 6;
i = 4; j = 9;
```

Respuesta:

```
A. i = 4; j = 8;
B. i = 3; j = 8;
D. i = 4; j = 9;
```

9

Organización de código en C++

En este capítulo vamos a discutir diversos elementos del lenguaje que sirven para ayudar al programador a organizar el código de forma modular. Hablaremos de funciones, de encabezados, de namespaces, de archivos `include` y de bibliotecas. Al finalizar estos capítulos, el alumno debe ser capaz de escribir programas en C++ modulares, y de organizar el código de forma eficiente.

Funciones

En C, las funciones representaban el escalón más alto que se podía alcanzar en cuanto a la modularidad de programas. Obviamente, para la programación orientada a objetos, existen elementos de mayor complejidad como las clases que permiten la implantación de este paradigma de programación. En este capítulo nos centraremos primero en las características de las funciones en C++.

Declaración y definición de funciones

En C, y en C++ las funciones deben ser siempre declaradas en cualquier programa. Sin embargo pueden estar definidas en otro archivo, y ser enlazadas (linkadas) tras su compilación. La sintaxis de una **declaración** es:

```
tipo_retorno nombre_función(tipo var1, tipo var2, ...);
```

y la sintaxis de una definición, que se puede encontrar en otro archivo:

```
tipo_retorno nombre_función(tipo var1, tipo var2, ...)
{
   sentencias;
   return expresión;
}
```

En el Código fuente 103, definimos en un archivo SQUARE.CPP la función square.

```
int square(int a)
{
  int z;
  z = a * a;
  return z;
}
```

Código fuente 103

Y podemos utilizarla en el programa que muestra el Código fuente 104.

```
#include <iostream> // ejemplo 1
using namespace std;
int square(int);
int main()
{
  int i = 2;
  cout << square(i) << endl;
}
```

Código fuente 104

Que imprime 4. Se observa que toda función tiene un tipo de retorno, en nuestro caso un int. El valor devuelto se envía mediante el comando return. Si no existe un valor de retorno, la función es de tipo void. Veamos, por ejemplo, el Código fuente 105.

```
#include <iostream>
using namespace std;
void square (int a)
{
  int z = a * a;
  cout << z << endl;
}
```

Código fuente 105

Normalmente, para cada función, el compilador genera su tabla de variables, memoria, asignaciones, etc. Sin embargo, existen casos en que las funciones se pueden definir **inline**.

```
inline int z2 (int x) { return x*x; }
```

Código fuente 106

En este caso, la intención del programador es que el compilador sustituya la llamada a la función por sus sentencias equivalentes. Este caso conduce a un código más eficiente, si la función pertinente es muy simple. En cualquier caso, es potestad del compilador realizar la sustitución, para optimizar el código.

Variables estáticas

En el capítulo anterior hemos discutido el alcance de las variables en detalle. Dentro de las funciones existe un tipo especial de variable denominado estática. Estas variables poseen un valor que se conserva a través de las distintas llamadas. Veamos, por ejemplo, el Código fuente 107.

```
#include <iostream> // ejemplo 2
using namespace std;

void f()
{
  static int i = 1;
  cout << "f invocada " << i++ << " veces\n";
}

int main()
{
  for (int i = 0; i < 4; i++) f();
}
```

Código fuente 107

La variable i dentro de f se inicia una vez con el valor 1. Con el operador post-incremento, se incrementa una unidad, después de imprimir su valor por pantalla. En la siguiente invocación i no se inicia de nuevo, sino que conserva su valor anterior.

Paso de argumentos

Existen diversos modos de pasar argumentos a funciones, lo que dota a C++ de una gran flexibilidad. El primer tipo es el **paso de argumentos por valor**, ya visto en C.

```
int g (int a) { ... }
```

En este caso, al invocar g, ésta crea su propia copia local de a, que podrá ser modificada y manipulada dentro de g, sin afectar al valor de la variable global a dentro del programa principal. Puede ocurrir que deseemos que la función modifique el programa principal. En este caso, pasamos **argumentos por referencia**. En este caso, definimos la función como

```
int g (int& a) { ... }
```

Ambos casos se ilustran en el Código fuente 108.

```
#include <iostream> // ejemplo 3
using namespace std;
```

```
void g(int& a, double f)
{
   a = 2 * a;
   f = 2.0 * f;
}

int main()
{
   int a = 1; // a se modifica
   double f = 1.0; // f no se modifica
   g(a, f);
   cout << "a: " << a << endl << "f: " << f << endl;
}
```

Código fuente 108

El pasar valores por referencia puede mejorar la eficiencia del programa en el caso de tipos creados por el usuario, clases, estructuras, etc. porque evita los procesos de reasignación de memoria dentro de la función. Pasar argumentos por referencia es equivalente también a la técnica de C utilizando punteros, es decir:

```
int g (int *a) { ... }
```

En este caso, g se debe invocar como g(&a);

Queremos reseñar que en C++ podemos dejar argumentos de la lista de argumentos sin utilizar. Este hecho no generará ningún error del compilador, y se puede utilizar para posibles ampliaciones de las capacidades de la función.

El modificador const en las variables de la lista de argumentos se usa para evitar que la función modifique cualquiera de los parámetros accidentalmente. Por otro lado es de utilidad cuando deseamos que literales, constantes y argumentos que requieren conversión de tipo se pasen a la función como argumentos válidos. Por ejemplo, definamos la función:

```
double sqrt (double& x) { ... }
```

e imaginemos que deseamos invocarla de la forma x = sqrt(3.9); En este caso, obtendremos un error de compilación. Sin embargo, si definimos la función como:

```
double sqrt (const double& x) { ... }
```

la invocación anterior sería correcta.

La equivalencia mencionada en el capítulo anterior entre arrays y punteros hace que no sea posible pasar arrays por valor. Conviene recordar, sin embargo, que no es posible conocer el tamaño del array en la función destino, salvo que sea un array de chars, en cuyo caso se puede buscar la terminación por el carácter 0. El tamaño debería ser una variable más en la lista de argumentos.

Sobrecarga de funciones

Un aspecto importante de C++, y de gran uso en la POO, es la sobrecarga de funciones. El concepto fundamental subyacente es muy simple. Si queremos realizar el mismo tipo de manipulación sobre objetos de distinto tipo, ¿porqué no llamar a la función del mismo modo? Así, en C++ podemos definir dos funciones, como vemos en el Código fuente 109.

```
void print (int x) { ... }
void print (double x) { ... }
```

Código fuente 109

El compilador distingue una de otra por los **argumentos**. No podemos definir dos funciones con el mismo nombre, el mismo tipo y número de argumentos pero con distinto tipo de retorno porque el compilador no podría decidir a cual de las dos formas de la función llamar. La sobrecarga de funciones es un arma potente, pero recomendamos su uso con prudencia. La forma de selección de los argumentos se realiza mediante el siguiente orden:

1. Coincidencia exacta del tipo.
2. Promociones entre tipos: bool, char o short para int. int para long. float para double. double para long double.
3. Conversiones estándar entre tipos: int para double, float a int, etc.
4. Conversiones definidas por el usuario (se verán más adelante).

El Código fuente 110 muestra un caso de ambigüedad.

```
void print(double);
void print(long);
void f()
{
   print (1L); // correcto, versión "long"
   print (1.0); // correcto, versión "double"
   print (1); // error ¿qué versión?
}
```

Código fuente 110

La propiedad de sobrecarga de funciones no es aplicable a funciones con diferente alcance.

Argumentos con valores por defecto

En C++ existe la posibilidad de definir argumentos de funciones con valores por defecto. Para ello basta con asignarles un valor:

```
int ff (double a, double x, int i = 0) { ... }
```

En la invocación de ff, si no se desea modificar el valor por defecto de i, se puede omitir: j = ff(a, x); Se pueden definir tantas variables por defecto como se deseen. Si en una función de N argumentos, el argumento N-i tiene un valor por defecto, es necesario que lo tengan también los argumentos N-i+1, ..., N. Es decir, todos los argumentos a su derecha.

En C++ existe la posibilidad, al igual que en C, de definir funciones con un número variable de argumentos. Referimos al alumno al capítulo correspondiente en C (capítulo 6) para su estudio, y a la ayuda de Visual C++.

Punteros a funciones

En C++ existe la posibilidad de definir punteros a funciones. Esta es una herramienta de gran potencia, que permite que una función sea argumento de otra función. La sintaxis es totalmente análoga a los punteros con variables. Por ejemplo, vemos el Código fuente 111.

```
void error (string s) { ... }
void (*perr)(string);
void f ()
{
   perr = &error; // asigna a perr la dirección de error
   perr("Ha cometido un error"); // correcto
}
```

Código fuente 111

Macros

Las macros son elementos de los que debe huir el programador en C++ con dos excepciones. La primera de ellas corresponde a la inclusión de archivos de biblioteca con la directiva #include. La segunda la veremos posteriormente. Una macro consiste fundamentalmente en sustituir un trozo de código por un nombre equivalente, con la directiva #define

```
#define nombreMacro Resto De La Línea De Código Equivalente
```

En las macros, se pueden definir variables, que se recomienda vayan entre paréntesis. Como vemos en el Código fuente 112.

```
#define printHex(a) cout << hex << a
```

Código fuente 112

En las macros no existe sobrecarga, ni puede haber recursión.

```
#define factorial(n) (n>1) ? n*factorial(n-1) : 1
```

Código fuente 113

Compilación condicional

Esta es la segunda excepción a la regla de la ausencia de macros. Es muy frecuente que el programador decida compilar su código con unas características u otras dependiendo de determinadas condiciones, como por ejemplo distintas plataformas, etcétera. Esos argumentos se pasan al compilador en tiempo de compilación, y éste decide qué código incluir. La sintaxis es:

```
#ifdef VARIABLE
   sentencias;
#else
```

```
   sentenciasAlternativas;
#endif
```

Espacios de nombres (*Namespaces*)

La utilización de namespaces o espacios de nombres es una técnica de C++ que permite crear distintos módulos con distintas utilidades, que a su vez permiten crear un interfaz para el usuario del módulo. De este modo se puede separar el interfaz del módulo de su implantación. Los namespaces expresan un agrupamiento lógico entre variables, funciones, clases, etc. La sintaxis es:

```
namespace NombreGenerico {
   double ax, bx;
   int z;
   class Vector { ... };
   double f(bool);
   int expr(double);
}
```

Las funciones se pueden definir dentro del mismo namespace, o en otro archivo. Para referirse a la función f del namespace NombreGenerico, se usa el operador de alcance.

```
double NombreGenerico::f(bool get) { ... }
```

Código fuente 114

La referencia a una variable o función de un namespace es siempre a través del operador de alcance. Sin embargo, como hemos visto con la biblioteca estándar, se puede usar en un programa el comando using.

```
using namespace NombreGenerico;
f (a); // usa NombreGenerico::f
void g()
{
   bool b;
   f (b); // usa NombreGenerico::f
}
```

Código fuente 115

Se pueden utilizar namespaces dentro de otros namespaces.

```
namespace NombreGenerico {
   int z;
   double f (bool);
   int expr (double);
   using namespace Mio;
}
```

Código fuente 116

También para ciertos casos genéricos, existen namespaces sin nombre. La utilización de una función u otra depende del contexto del programa.

Alias de namespaces

La utilización de nombres de espacios muy cortos (por ejemplo, namespace A{ ... }) puede conducir tarde o temprano a la colisión con otro namespace de igual nombre. Por ello, conviene usar nombres de namespaces significativos, como muestra el Código fuente 117.

```
namespace American_Telegraph_and_Telephone { ... }
```

Código fuente 117

A la hora de trabajar con estos nombres, puede resultar engorroso. Por ello, C++ permite la definición de nombres alternativos (alias) para los namespaces, como vemos en el Código fuente 118.

```
namespace ATT = American_Telegraph_and_Telephone { ... }
```

Código fuente 118

Composición de namespaces

Existe en C++ la posibilidad de componer namespaces a partir de otros. El objetivo es construir módulos complejos basados en otros más sencillos. Por ejemplo, supongamos definidos los namespaces Su_string y Su_vector (Código fuente 119)

```
namespace Su_string {
   class String { ... };
   String operator+(const String&, const String&);
   String operator+(const String&, const char *);
}
namespace Su_vector {
   template<class T> class Vector { ... };
}
```

Código fuente 119

Podemos definir un módulo adicional basado en estos dos, como aparece en el Código fuente 120.

```
namespace Mi_lib {
   using namespace Su_cadena;
   using namespace Su_vector;
   void myFct(String&); // usa String de Su_cadena
}
```

Código fuente 120

De igual modo, se puede hacer una selección limitada. Por ejemplo, si deseamos usar sólo la clase String de Su_cadena, debemos rescribir Mi_lib como muestra el Código fuente 121.

```
namespace Mi_lib {
   using Su_cadena::String;
   ...
}
```

Código fuente 121

Por último, debemos recordar que los namespaces son abiertos. Por ello, podemos añadirles elementos dentro del código. Por ejemplo, definiendo lo que vemos en el Código fuente 122.

```
namespace A { int f(); }
```

Código fuente 122

Podemos escribir más tarde lo que muestra el Código fuente 123.

```
namespace A { int g(); }
```

Código fuente 123

que añade la función g a A.

Excepciones

C++ proporciona una gestión de errores más sofisticada que C, a través de las excepciones. Cuando un programa está compuesto de distintos módulos, y especialmente, cuando estos módulos pertenecen a bibliotecas desarrolladas por separado, la gestión de los errores debe dividirse en dos partes:

1. Informar sobre las condiciones de error que no pueden resolverse localmente.
2. Manipular o gestionar los errores detectados por el código.

Throw ... try ... catch

Las excepciones están especialmente diseñadas para casos en los que no es posible gestionar el error de forma local. El Código fuente 124 muestra un caso típico para la gestión de errores.

```
#include <iostream> // ejemplo 4
using namespace std;

struct Range_error
{
```

```
   int i;
   Range_error(int ii) { i = ii; } // constructor
};

char to_char(int i)
{
   char c = i & static_cast<unsigned char>(-1);
   if (i != c) throw Range_error(i);
   return c;
}

int main()
{
   int x = 1100;
   try {
      char c = to_char(x);
      cout << "Caracter de " << x << " = " << c << endl;
   }
   catch(Range_error err) {
      cerr << "Error terrible de rango: " << err.i << endl;
   }
}
```

Código fuente 124

Conviene en primer lugar crear distintos tipos para la gestión de errores. Estos suelen ser estructuras o clases definidos ad-hoc para la gestión de errores. Para programas diseñados por el alumno recomendamos primero la creación de un namespace donde situar los tipos de error, y crear un tipo para cada categoría de error. En nuestro caso hemos creado el tipo Range_error para gestionar errores de rango con char. El siguiente paso es que la función que detecta el error (to_char) lance mediante el operador throw el error a través del constructor definido. El error ocurre cuando el entero está fuera de rango para ser un carácter. Finalmente, el código contiene el grupo try { /* sentencias */ } catch(error) { /* sentencias */ }. Las sentencias de try se ejecutan hasta que encuentren algún error. Si no encuentran ninguno, prosigue la ejecución después de catch. Si alguna sentencia lanza alguna excepción, el programa compara con el argumento de catch. Si coincide, ejecuta el conjunto de sentencias que el programador ha creado para gestionar el error. En este caso, imprimir un mensaje de error. Pueden existir multitud de sentencias catch para un único try, gestionando distintos tipos de error.

También dentro de grupos catch, se pueden anidar otros try ... catch, alcanzando distintos niveles de complejidad.

Programas y archivos fuente: organización del código

Aunque este tema se ha discutido en la parte dedicada a C, conviene aclarar algunos conceptos sobre la organización del código de C++. Obviamente, es imposible tener todo el código de un programa en un único archivo, especialmente cuando estamos trabajando con librerías que no hemos diseñado nosotros. Incluso si todo el material ha sido diseñado por nosotros, resulta poco práctico tratar de gestionar todo el programa en un único archivo.

Compilación por separado

La solución obvia es dividir el código en partes. Visual C++ y otros entornos de desarrollo integrado (IDE) facilitan la tarea del programador para gestionar programas de cierto tamaño mediante la creación de proyectos. Pero la creación de estos proyectos no es algo automático. El desarrollador

debe organizar el código en distintos archivos. Esto es lo que se conoce como **estructura física del programa**. En la mayoría de los sistemas operativos, el archivo es la unidad tradicional de almacenamiento y, por lo tanto, de compilación. La organización lógica consiste en separar las declaraciones de las definiciones. En el proceso de enlace (linkado), se agrupan todos los elementos.

Las herramientas para dividir el programa se basan en la utilización de archivos de encabezado, y en la declaración de variables externas (extern). Por ejemplo, podemos tener dos archivos con la estructura que aparece en el Código fuente 125.

```
file1.cpp

   int x = 1; // definición y declaración
   int f() { ... } // definición

file2.cpp

   extern int x; // declaración
   int f(); // declaración
   int g() { ... x = f(); } // definición y declaración
```

Código fuente 125

La necesidad para evitar repetir en cada archivo todas las declaraciones de variables y funciones utilizadas, introduce la inclusión de archivos de encabezado (*header files*), reconocidos históricamente por su extensión *.h. Los archivos de encabezado se invocan con la directiva que aparece en el Código fuente 126,

```
   #include <nombreArchivo> // archivo en el PATH estándar de las bibliotecas de
C++
   #include "nombreArchivo" // archivo en el directorio actual o en el directorio
del proyecto
```

Código fuente 126

y pueden contener

- Espacios de nombres (namespaces) y namespaces sin nombre
- Definiciones de tipos
- Declaraciones de plantillas (templates)
- Definiciones de plantillas (templates)
- Definiciones de funciones inline
- Definiciones de funciones ordinarias
- Declaraciones de datos
- Definiciones de datos

- Definiciones de constantes
- Enumeraciones
- Declaraciones de nombres
- Definiciones de macros
- Otros encabezados mediante directivas #include
- Directivas de compilación condicional
- Definiciones de plantillas exportadas
- Definiciones de arrays
- Comentarios

En general, hay que aplicar la **Regla de la Definición Única**: clases, plantillas, enumeraciones y funciones deben estar **definidas** una única vez en un programa.

Otro de los problemas típicos en la organización de código es la inclusión (debido a llamadas repetitivas de #include) múltiples veces de archivos de encabezado. El problema se soluciona organizando cada archivo de encabezado de la como muestra el Código fuente 127.

```
#ifndef MI_ARCHIVO_H
#define MI_ARCHIVO_H // la siguiente inclusión no lee el resto del archivo
   sentencias;
#endif
```

Código fuente 127

Terminación de programas

Un programa en C++ puede terminar por las siguientes causas:

- Sentencia return desde main()
- Llamada a la función exit() desde cualquier función
- Llamada a la función abort() desde cualquier función
- Por lanzar una excepción no capturada

Por último, queremos destacar la función que aparece en el Código fuente 128.

```
int atexit(&fnct); // necesita #include <cstdio>
```

Código fuente 128

que ejecuta el código de fnct al salir de la correspondiente función. Debemos destacar que el número de llamadas a atexit está limitado en cada programa. Cuando se alcanza ese límite, se devuelve el valor 0.

Ejercicios

1. Definamos la función g como: int g(double x, float* r, int& a) { ... } ¿cuál de las siguientes invocaciones es correcta?

   ```
   g(x, 2.0, a);   g(&x, r, 1);   g(x, &r, a);   g(1.0, &r);
   ```

2. ¿Es correcto el siguiente código? ¿Porqué?

   ```
   #include <cstdio>
   #include <iostream>
   using namespace std;

   int g (char *a)
   {
      return strlen(a);
   }

   int main()
   {
      char *z = "Visual C++";
      int i = g(z);
      cout << "i: " << i << endl;
      return 0;
   }
   ```

3. ¿Es correcto el siguiente código?

   ```
      void erase (double);
      void erase (float);
      void hh() { erase(1.0); }
   ```

4. ¿Es correcto el siguiente código?

   ```
      void f (int x, double y = 0.0);
      void f (int x);
      void g (int x) { int h = f(x); }
   ```

5. ¿Es correcto el siguiente código?

   ```
      void f (int x, double y = 0.0, int z);
   ```

6. ¿Es correcto el siguiente código?

```
void f (int z, double y = 0.0, int x = 1);
```

7. Realizar un programa que calcule el factorial de un número entero a partir de una función recursiva. El factorial de un número N se define por el producto de sí mismo por todos los enteros positivos menores que él hasta 1, es decir: N*(N-1)*...*2*1.

8. Realizar una versión en C++ del programa cal descrito en el ejercicio 10, capítulo 5 (parte C). (Nota: la manipulación de argumentos en la línea de comandos es igual en C y en C++; el ancho del campo de impresión en cout se expresa llamando a la función cout.width(ancho); antes de usarlo).

9. Crear una función para dividir dos números de doble precisión y que lance una excepción antes de dividir por cero.

10. Utilizar esta función para un programa calculadora que divide dos números introducidos por el usuario. Capture las excepciones lanzadas.

Respuestas a los ejercicios

1. Definamos la función g como: int g(double x, float* r, int& a) { ... } ¿cuál de las siguientes invocaciones es correcta?

   ```
   g(x, 2.0, a);   g(&x, r, 1);   g(x, &r, a);   g(1.0, &r);
   ```

   Respuesta:

   ```
   g(x, 2.0, a);
   g(&x, r, 1);
   g(1.0, &r);
   ```

2. ¿Es correcto el siguiente código? ¿Porqué?

```
#include <cstdio>
#include <iostream>
using namespace std;

int g (char *a)
{
   return strlen(a);
}

int main()
{
   char *z = "Visual C++";
   int i = g(z);
   cout << "i: " << i << endl;
   return 0;
}
```

   Respuesta: Sí. Porque a es una cadena de caracteres.

3. ¿Es correcto el siguiente código?

```
void erase (double);
void erase (float);
void hh() { erase(1.0); }
```

Respuesta: No. No es posible determinar que función llamar.

4. ¿Es correcto el siguiente código?

```
void f (int x, double y = 0.0);
void f (int x);
void g (int x) { int h = f(x); }
```

Respuesta: No. No es posible determinar que función sobrecargar.

5. ¿Es correcto el siguiente código?

```
void f (int x, double y = 0.0, int z);
```

Respuesta: No. El argumento y debería ser el último de la lista.

6. ¿Es correcto el siguiente código?

```
void f (int z, double y = 0.0, int x = 1);
```

Respuesta: Sí.

7. Realizar un programa que calcule el factorial de un número entero a partir de una función recursiva. El factorial de un número N se define por el producto de sí mismo por todos los enteros positivos menores que él hasta 1, es decir: N*(N-1)*...*2*1.

```
#include <iostream>
using namespace std;

long fact (int i)
{
  return (i > 1) ? i * fact (i-1) : 1;
}

int main()
{
  int x;
  cout << "Introduce un numero entero: ";
  cin >> x;
  cout << "Factorial de " << x << " = " << fact (x) << endl;
}
```

8. Realizar una versión en C++ del programa cal descrito en el ejercicio 10, capítulo 5 (parte C). (Nota: la manipulación de argumentos en la línea de comandos es igual en C y en C++; el

ancho del campo de impresión en cout se expresa llamando a la función cout.width(ancho); antes de usarlo).

```
#include <iostream>
#include <cstdlib>
using namespace std;

int bisiesto (int year)
{
  if ( ( year % 4 == 0 && year % 100 != 0 ) || (year % 400 == 0) )
    return 1;
  return 0;
}

int main (int argc, char* argv[])
{
  int i, dia_Ene1, mes, agno, fecha;
  int dias_mes[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  char* nombre_mes[] = { "enero", "febrero", "marzo", "abril",
                 "mayo", "junio", "julio", "agosto",
                 "septiembre", "octubre", "noviembre",
                 "diciembre" };
  char _string[60];

  if (argc != 3)
  {
    cout << "Uso: ex8 mes año\n";
    exit (1);
  }
  mes = atoi (argv[1]);
  agno = atoi (argv[2]);
  if (mes < 1 || mes > 12)
  {
    cout << "Error en la lectura del mes.\n";
    exit (1);
  }
  if (agno < 1900)
  {
    cout << "Error en la lectura del año.\n";
    exit (1);
  }
  // Averiguando el dia de la semana para el año
  for (fecha = 1900, dia_Ene1 = 1; fecha < agno; fecha++)
  {
    if (bisiesto (fecha)) dia_Ene1 += 2;
    else dia_Ene1++;
    if (dia_Ene1 >= 7) dia_Ene1 -= 7;
  }
  // 366 dias para un año bisiesto
  if (bisiesto (agno)) ++dias_mes[1];
  for (i = 1; i < mes; i++)
    dia_Ene1 = (dia_Ene1 + dias_mes[i-1]) % 7;
  cout.width (15); cout << nombre_mes[mes-1] << ' ';
  cout.width (5); cout << agno << endl;
  cout << "Dom Lun Mar Mie Jue Vie Sab\n"
       << "--- --- --- --- --- --- ---\n";
  for (fecha = 1; fecha <= dia_Ene1 * 4; fecha++)
    cout << ' ';
  for (fecha = 1; fecha <= dias_mes[mes-1]; fecha++)
  {
    cout.width (3); cout << fecha;
    if ((fecha + dia_Ene1) % 7 > 0) cout << ' ';
    else cout << endl;
  }
```

```
  cout << endl;
  return 0;
}
```

9. Crear una función para dividir dos números de doble precisión y que lance una excepción antes de dividir por cero.

```
struct DividirCero { };

double dividir (double x, double y)
{
  if (y == 0.0) throw DividirCero;
  return y/x;
}
```

10. Utilizar esta función para un programa calculadora que divide dos números introducidos por el usuario. Capture las excepciones lanzadas.

```
#include <iostream>

struct DividirCero { };

double dividir (double x, double y)
{
  DividirCero tmp;
  if (y == 0.0) throw tmp;
  return x/y;
}

int main()
{
  double x, y;
  cout << "Introduce 2 números reales para dividirlos: ";
  cin >> x >> y;
  try {
    cout << x << " dividido por " << y << " = "
         << dividir (x, y) << endl;
  }
  catch (DividirCero) {
    cout << "Error grave: intento de division por cero.\n";
  }
  return 0;
}
```

10

Abstracción de datos en C++ - clases

Los elementos que hemos analizado hasta ahora de C++, aunque potentes, no dotan al lenguaje de la versatilidad que uno esperaría de un lenguaje orientado a objetos. En este capítulo vamos a analizar en detalle las clases, que hacen que C++ posea toda la riqueza de la Programación Orientada a Objetos.

Clases

Las clases (`class`) son un concepto de C++ (y de otros lenguajes como Java o SmallTalk) que concede al programador una herramienta para crear tipos propios en sus programas tan convenientes o más que los tipos intrínsecos del lenguaje. Una clase (o cualquier otro tipo de C++) no es más que la representación de un concepto. Al igual que existe el tipo float para representar el concepto matemático de los números reales, podemos crear en un programa de juegos la clase Villano para representar los movimientos de algunos elementos.

Funciones miembro

Las clases están compuestas por variables y funciones. Estos elementos se conocen como **funciones miembro**. La estructura típica de la declaración de una clase aparece en el Código fuente 129.

```
class NombreClase {
   int d, m, a;
```

```
   void iniciar (int dd, int mm, int aa);
};
```

Código fuente 129

El lector debe observar el punto y coma al final de la declaración. Es un caso análogo a las estructuras. Las definiciones de las funciones siguen la misma convención que los namespace, usando el operador de alcance. Veamos, por ejemplo, el Código fuente 130.

```
NombreClase::iniciar(int dd, int mm, int aa)
{
   sentencias;
}
```

Código fuente 130

El acceso a las funciones miembro se realiza con el operador ".". Por ejemplo, dada una clase NombreClase X, accedemos a iniciar con X.iniciar(dd, mm, aa);

Control de acceso

Una de las características de la POO es la **encapsulación de datos**. Con ello, al desarrollar distintos tipos de datos podemos separar el interfaz que presenta la clase al usuario de esa clase, de su implantación real. Por ejemplo, supongamos que en un sistema de ventanas cualquiera, se desarrolla la clase Menu para representar un Menú. Si los desarrolladores de esa clase, deciden portar el código a otro sistema operativo, el código cambiará, pero si mantienen el mismo interfaz para la clase, cualquier código desarrollado sobre la base de esa clase **no deberá ser modificado**. La definición que hemos presentado hasta ahora de las clases no las diferencia nada de las estructuras. Sin embargo, las clases disponen de los elementos public y private para distinguir que partes de la clase son públicas o privadas. Por defecto, el elemento private no es necesario, y todo lo que aparezca antes de public será considerado como privado de la clase.

```
class Fecha {
private: // opcional, private: por defecto
   int d, m, a;

public:
   void iniciar (int dd, int mm, int aa);
   void suma_dia (int dd);
};
```

Código fuente 131

En general, una estructura es una clase con todos sus elementos públicos. Cualquier intento por utilizar los elementos privados de una clase por un programa conduce a error.

```
void f (Fecha& fec)
{
```

```
   fec.a = 200; // error
}
```

Código fuente 132

Constructores

La función iniciar presentada para la clase Fecha permite definir los valores iniciales de cualquier elemento del tipo Fecha. Sin embargo, el hecho de que cada desarrollador de clases tenga libertad para elegir los nombres de sus funciones de inicialización conduciría ciertamente a confusión. La aproximación elegida por C++, es más intuitiva. Las funciones de inicialización (constructores) poseen el mismo nombre que la clase. Por ejemplo, para la clase Fecha tendríamos el Código fuente 133.

```
class Fecha {
   int d, m, a;
public:
   Fecha (int dd, int mm, int aa); // día, mes y año
   Fecha (int dd, int mm); // día y mes; año actual por defecto
   Fecha (int dd); // día; mes y año actual por defecto
   Fecha(); // día, mes y año actual por defecto
   Fecha (const char *); // iniciar con cadena de caracteres
};
```

Código fuente 133

Observe que se ha hecho uso de la sobrecarga de funciones para definir distintos tipos de constructores. Por ejemplo, serían declaraciones de clase Fecha válidas, las que aparecen en el Código fuente 134.

```
Fecha d (12, 10, 2000);
Fecha ff (10);
Fecha g;
Fecha s ("14 de abril de 1999");
```

Código fuente 134

Miembros estáticos

En el caso descrito anteriormente, puede ser conveniente que el programador disponga de la libertad de establecer una fecha por defecto para todas las instancias de la clase. Para ello existen los elementos estáticos (static). La aproximación más «sensible» en nuestro caso es declarar una variable privada estática de tipo Fecha, y una función estática pública para modificar esa fecha por defecto.

```
class Fecha {
   int d, m, a;
   static Fecha fec_defecto;
public:
   Fecha (int dd = 0, int mm = 0, int aa = 0);
   Fecha (const char *);
```

```
      static void est_fec_defecto (int dd, int mm, int aa);
   };
```

Código fuente 135

Para invocar una función estática no hace falta que se cree una instancia de la clase. Se puede invocar con la clase general.

```
   void f() { Fecha::est_fec_defecto (1,1,1900); } // fec_defecto = "1 de enero de
1900"
```

Código fuente 136

En el ejemplo de la clase Fecha, hemos reducido la declaración de múltiples constructores a una única declaración haciendo uso de los argumentos por defecto en C++, y de que no existen días, meses o años cero en el calendario europeo. Así, la definición del constructor se muestra en el Código fuente 137.

```
   Fecha::Fecha (int dd, int mm, int aa)
   {
      d = dd ? dd : fec_defecto.d;
      m = mm ? mm : fec_defecto.m;
      a = aa ? aa : fec_defecto.a;
   }
```

Código fuente 137

Copiando objetos de clases

En capítulos anteriores hemos observado que existe la posibilidad para los tipos básicos de definir una variable, asignándole el valor de otra. Para las clases, existe el mismo mecanismo, que por defecto asigna a cada variable miembro el mismo valor que las variables miembro del objeto asignado. Así, es legal definir el Código fuente 138.

```
   Fecha navidad (24, 12, 1999);
   Fecha d = navidad;
```

Código fuente 138

Si se desea modificar el comportamiento de la declaración anterior, basta con definir en la clase X correspondiente un constructor copia de la forma X::X(const X&);

Funciones miembro constante

Hasta ahora hemos creado funciones que permiten modificar los valores de las variables miembro de la clase. Pero obviamente, en los programas nos vamos a encontrar con la necesidad de conocer el

estado de una clase, sin modificarlo. Por ejemplo, podemos desear conocer el día, el mes o el año de cualquier Fecha. Para ello, deberíamos declarar en el interfaz público de Fecha las funciones, que aparecen en el Código fuente 139.

```
int dia() const { return d; .
int mes() const { return m; }
int ano() const { return a; }
```

Código fuente 139

El operador const antes de la definición de la función indica que no se va a poder modificar el estado de la clase.

Auto-referencia

Imaginemos que deseamos modificar el estado de una instancia de una clase, y que se devuelva como valor la propia clase. Así, se podrían concatenar llamadas a distintas funciones miembro.

```
Fecha cumple (6, 1, 1964); cumple.sumaAno(1).sumaMes(4).sumaDia(2);
```

Código fuente 140

Por ejemplo, podríamos declarar en la clase Fecha la función que muestra el Código fuente 141

```
Fecha& sumaAno (int n);
```

Código fuente 141

y definirla como aparece en el Código fuente 142

```
Fecha& Fecha::sumaAno (int n)
{
   if(d == 29 && m == 2 && !bisiesto(y + n)) // año bisiesto
   {
      d = 1; m = 3;
   }
   a += n;
   return *this;
}
```

Código fuente 142

La clave de la auto-referencia es el operador this. this devuelve la dirección de memoria de la instancia de la clase. Así, en la definición anterior, se podría haber sustituido las llamadas a las variables miembro por this->d, this->m o this->a respectivamente.

Constancia física y lógica

Sea una instancia de la clase Fecha que transformamos frecuentemente en una cadena de caracteres. El coste computacional de realizar frecuentemente esta "traducción" puede llegar a ser alto. Sería pues conveniente disponer en la propia clase de un lugar donde almacenar esta representación, una especie de cache de memoria, y de un "semáforo lógico" que indique al sistema si la representación está actualizada. Para ello, existe la palabra clave mutable, que indica que la variable miembro es constante, pero puede ser modificada por otras funciones miembro.

Así en la clase Fecha, podríamos incluir las modificaciones que se indican en el Código fuente 143.

```
class Fecha {
   int d, m, a;
   mutable bool cache_valido;
   mutable string cache;
   void computar_cache_valido() const; // calcula cache
   ...
public:
   ...
   string fecha_cadena() const;
};
```

Código fuente 143

Donde fecha_cadena() se define como muestra el Código fuente 144.

```
string Fecha::fecha_cadena() const
{
   if(!cache_valido)
   {
      computar_cache_valido();
      cache_valido = true;
   }
   return cache;
}
```

Código fuente 144

Definición de funciones dentro de clases

Hasta ahora hemos separado la declaración de las funciones miembro de sus definiciones. Esto es lo recomendable, porque siempre es posible declarar una clase en un archivo de encabezado para definir posteriormente la clase en otro archivo para crear la correspondiente biblioteca de clases. Sin embargo, para funciones simples, de definición sencilla, se puede incluir ésta en la propia clase. En ese caso, se considera la función como inline.

Destructores

Es evidente que los objetos necesitan constructores. Pero al mismo tiempo, si los objetos que vamos generando ocupan un espacio en memoria, y no vamos liberando ese espacio, llegará un momento en que saturaremos la memoria del sistema. C++ proporciona un mecanismo de destrucción de objetos

por defecto para clases con variables miembro sencillas, cuando ya no son necesarias. Sin embargo, para objetos complejos, ese mecanismo no funciona, o bien deseamos usar un algoritmo más eficaz, y al no existir un recolector de basura (*garbage collector*) como en Java, debemos definir el destructor nosotros mismos.

La declaración de destructores es sencilla. Sean dos clases las que aparecen en el Código fuente 145.

```
class Nombre {
   const char *s;
   ...
};
class Tabla {
   Nombre *p;
   size_t sz; // tipo definido por el sistema para arrays (normalmente int)
public:
   Tabla(size_t s = 10) { p = new Nombre[sz = s]; }
   ~Tabla() { delete[] p; }
};
```

Código fuente 145

En este caso, al tener Tabla como componente un array de Nombres, definido con new, debemos obligatoriamente incluir en el destructor el operador delete[]. En clases utilizadas como variables locales, los destructores se invocan automáticamente al salir de la función o del bloque bajo consideración.

Copiando objetos

Hemos visto anteriormente la copia de objetos por asignación, pero conviene también definir en una clase un constructor copia para declaraciones de variable del estilo de Tabla c(); ... Tabla t(c);. Estos constructores se definen como se muestra en el Código fuente 146.

```
class Tabla {
   Nombre *p;
   size_t sz; // tipo definido por el sistema para arrays (normalmente int)
public:
   Tabla (size_t s = 10) { p = new Name[sz = s]; }
   Tabla (const Tabla&); // constructor copia
   Tabla& operator= (const Tabla); // asignando copia
   ~Tabla() { delete[] p; }
};
Tabla::Tabla (const Tabla& t)
{
   p = new Nombre[sz = t.sz];
   for (int i = 0; i < sz; i++) p[i] = t.p[i];
}
Tabla& Tabla::operator= (const Tabla& t)
{
   if (this != t) // cuidado con t=t
   {
      delete[] p;
      p = new Nombre[sz = t.sz];
      for (int i = 0; i < sz; i++) p[i] = t.p[i];
   }
```

```
   return *this;
}
```

Código fuente 146

Almacenamiento libre

Las clases se pueden declarar con new. En este caso, no es posible la desasignación automática de memoria invocando el destructor. En esos casos es absolutamente imprescindible utilizar delete antes de salir del bloque para evitar "fugas de memoria". Veamos el Código fuente 147.

```
int main()
{
   Tabla *p = new Tabla;
   Tabla *q = new Tabla(20);
   ...
   delete p;
   delete[] q;
}
```

Código fuente 147

Inicio obligatorio de variables miembro

Existen clases en las que puede ser absolutamente necesario definir algunas de las variables miembro para evitar que se declare un estado del sistema con carencia de significado. Por ejemplo, definamos una clase Club, en la que cada instancia (cada Club nuevo que creemos) debe tener obligatoriamente un nombre y una fecha de fundación.

```
class Club {
   string nombre;
   Tabla miembros, oficiales;
   Date fundado;
   ...
public:
   Club (const string& n, Date fd);
   ...
};
```

Código fuente 148

Donde el constructor se debe declarar como muestra el Código fuente 149.

```
Club::Club (const string& n, Date fd) : nombre(n), fundado(fd)
{
   ...
}
```

Código fuente 149

De este modo queda claro que tanto n como fd son variables que deben estar siempre definidas.

Arrays

Si una clase se puede construir sin inicializadores explícitos, se pueden definir arrays de esa clase como:

```
Tabla tbl[20];
```

Debemos señalar que siempre hay que asignar una dimensión, **no** siendo posible inicializadores dinámicos como los descritos para los tipos intrínsecos de C++: Tabla tbl[] = { ... conjunto_de_valores ... };

Para arrays creados sin new, se invoca automáticamente el destructor de la clase al salir del bloque correspondiente. Para arrays creados con new, es necesario invocar delete, como se ha visto anteriormente. En variables definidas fuera de cualquier función (variables globales, en namespace, o variables estáticas de clases), el orden de creación es el siguiente: se invocan los constructores, se ejecuta el programa main(), y antes de devolver el control al sistema, se invocan los destructores correspondientes.

Sobrecarga de operadores

Cada campo técnico tiene definidas una serie de operaciones entre elementos típicos, que se pueden expresar de forma concisa. Así, las matemáticas tienen distintos operadores que permiten realizar operaciones entre los distintos números enteros, reales, complejos, etc. Pero existen otro tipo de operadores en otros campos de la técnica: conexiones entre líneas telefónicas, carreteras, etc. Para cada uno de ellos, C++ facilita la creación de un tipo propio (clase), y como es lógico, facilitará la representación de operaciones entre ellos. Debemos remarcar aquí que otros lenguajes orientados a objetos como Java no disponen de esta flexibilidad. Ya hemos visto los operadores entre los tipos intrínsecos de C++. En esta sección analizaremos como definir el comportamiento deseado para nuestra clase con distintos operadores.

Un ejemplo típico es el siguiente. Definimos una clase propia Complex, para representar los números complejos, que como sabemos equivalen a un par de números reales (parte real y parte imaginaria). Entre dos números complejos, deseamos crear el operador suma, que se comporta como nuestra intuición predice: sumando partes real e imaginaria por separado para crear otro número complejo.

```
#include <iostream> // ejemplo 1
using namespace std;

class Complex {
  double re, im;
public:
  Complex (double r, double i) : re(r), im(i) { }
  Complex operator+ (Complex);
  double Real() const { return this->re; }
  double Imag() const { return this->im; }
};

int main()
{
  Complex a(1.1, 0.1), b(0.2, -0.2);
  Complex c = b;
```

```
  c = a + b;
  cout << "c = (" << c.Real() << ", " << c.Imag() << ")" << endl;
}

Complex Complex::operator+ (Complex z)
{
  Complex tmp(z.re + this->re, z.im + this->im);
  return tmp;
}
```

C++ permite sobrecargar los operadores que aparecen en la Tabla 8.

+	-	*	/	%	^	&
|	~	!	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	<<=	>>=	==	!=	<=
>=	&&	||	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

Tabla 8

El hecho de que no se puedan declarar operadores además de los citados puede parecer poco flexible, pero el permitir mayor flexibilidad podría dar lugar a ambigüedades en la resolución de algunas sobrecargas.

Operadores binarios

En C++ los operadores binarios se pueden declarar de dos formas en C++:

- como función miembro no estática con un único argumento (ver el ejemplo 1), o
- como función no-miembro con dos argumentos. Por ejemplo para la clase X, declaramos el operador @, como: X operator@ (X,X);

Por ello, en C++, dado un operador @, la operación aa @ bb equivale a:

```
aa.operator @(bb);    o    operator @(aa,bb);
```

Significados predefinidos

En C++ existen operadores con significados predefinidos: operator=, operator[], operator-> . Estos operadores tienen que ser siempre funciones miembro no-estáticas. Los operadores asignación (=), dirección de (&) y secuencia (,) tienen significados preasignados. Si no se desea que las distintas instancias de una clase tengan acceso a estos operadores, conviene declararlos privados.

```
class X {
private:
   void operator= (const X&);
   void operator& ();
   void operator, (const X&);
};
```

Código fuente 150

Los operadores binarios cuyo primer operando es un tipo básico, y el segundo operando es una clase, no pueden ser nunca funciones miembro. Esto es así porque los tipos básicos no son clases. Este tipo de operador se define fuera de la propia clase

Funciones y clases amigas (*friends*)

Hasta ahora hemos visto que las funciones miembro de una clase:

1. tienen acceso a los elementos privados de la misma
2. la función pertenece al alcance de la clase
3. la función debe ser invocada sobre un objeto de esa misma clase (puntero *this)

Ocurre frecuentemente en el diseño de clases, la necesidad de que una función no-miembro posea la propiedad número 1, es decir, tenga acceso a los elementos privados de la misma. Para ello, basta con declarar la función como friend (amiga). Un ejemplo típico es el caso de vectores y matrices en el Álgebra lineal. Estos elementos tienen operaciones algebraicas comunes como la multiplicación. En este caso, resulta conveniente definir el operador producto como friend. La función friend debe ser declarada en ambas clases, si se desea que tenga acceso a las partes privadas de cada una de ellas. También se puede declarar una clase amiga de otra. Veamos el Código fuente 151.

```
class Matrix;
class Vector {
   ...
   friend Vector operator* (const Matrix&, const Vector&);
};
class Matrix {
   ...
   friend Vector operator* (const Matrix&, const Vector&);
};
```

Código fuente 151

Clases derivadas

En la vida real no existen objetos aislados. Cada objeto existe dentro de un contexto, y tiene una serie de propiedades, pero sobre todo, existe en relación con otros objetos de los que puede derivar. Por ejemplo, pensemos en un programa para la manipulación de coches. Rápidamente pensaríamos en una clase coches con todas sus características: ruedas, volantes, etc. Pero puede ocurrir, que tengamos que manipular posteriormente camiones. ¿Debemos re-estructurar nuestro programa para manipular coches y camiones? ¿Y si más tarde necesitamos trabajar con autobuses? Evidentemente, todo alumno familiarizado con la POO habrá deducido que la respuesta es no. La aproximación común es diseñar una clase genérica vehículos, y a partir de ahí obtener todo tipo de vehículos. Podemos introducir otro

ejemplo con un simple programa de diseño gráfico. Imaginemos que el usuario puede diseñar círculos, cuadrados, pentágonos, etc. La aproximación apropiada es diseñar una clase forma con las propiedades comunes, y a partir de ahí, todas las figuras geométricas. Estos ejemplos permiten introducir el concepto de **herencia** (fundamental en la POO). De igual modo, en esta sección hablaremos también de **polimorfismo**.

Vamos a ilustrar estos conceptos con un simple ejemplo para manejar registros de empleados de la base de datos de una empresa. En principio, cabe pensar que todos los registros de los empleados son iguales, y que no es necesario "heredar" clases. Sin embargo, el departamento de personal nos sacará pronto del error, y nos solicitará para cada manager (director de departamento) un registro de los empleados que tiene bajo su tutela. Evidentemente, no es apropiado generar dos clases distintas cuando tienen tantas propiedades en común. Podemos generar una primera aproximación al problema con las estructuras que muestra el Código fuente 152.

```
struct Empleado {
   string nombre, apellido;
   short dept;
   // ...
};
struct Manager : public Empleado {
   set<Empleado*> grupo; // conjunto de empleados; ver STL
   short nivel;
};
```

Código fuente 152

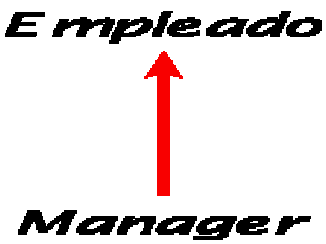


Figura 3

La sentencia : public Empleado indica al compilador que una estructura de tipo Manager posee las mismas variables que la estructura Empleado, con dos elementos adicionales: un conjunto de subordinados y un nivel en la escala de mandos. Esta dependencia se suele representar gráficamente como en la Figura 3: la estructura Manager depende de la estructura Empleado.

En cualquier función donde se utilice la estructura Empleado, se puede usar la estructura Manager. Por ejemplo, definamos una función que acepta dos argumentos, un Empleado y un Manager, que los añade a una lista. El código se podría escribir como vemos en el Código fuente 153.

```
void f (Empleado e, Manager m)
{
   list<Empleado*> elist;
   elist.push_front(m);
   elist.push_front(e);
   // ...
}
```

Código fuente 153

Funciones miembro

Como hemos visto, la verdadera potencia de C++ en la POO radica en la utilización de clases. Rescribamos nuestro ejemplo con las clases que aparecen en el Código fuente 154.

```
class Empleado {
   string nombre, apellido;
   short dept;
   // ...
public:
   void print() const;
   void nombre_completo() const;
   // ...
};
class Manager : public Empleado {
   set<Empleado*> grupo; // conjunto de empleados; ver STL
   short nivel;
public:
   void print() const;
   // ...
};
```

Código fuente 154

Observemos que Manager posee las variables y funciones, tanto privadas como públicas de Empleado. El ejemplo anterior redefine (**sobrecarga**) la función print(), porque Manager debe imprimir información adicional a la de Empleado, pero no sobrecarga nombre_completo() porque coincide con la función de Empleado. Esta técnica se conoce, como ya hemos visto como polimorfismo. El código de print() aparece en el Código fuente 155.

```
void Manager::print() const
{
   Empleado::print();
   cout << nivel << endl;
}
```

Código fuente 155

Debemos hacer notar que las clases derivadas no pueden acceder a los miembros privados de la clase base. En caso contrario, el concepto de privacidad (encapsulación) no tendría sentido. Bastaría con crear una clase derivada de otra clase para acceder a sus elementos privados. Sin embargo, existe un tercer tipo de acceso: protegido (protected:). Los elementos con acceso protegido pueden ser accedidos por las clases derivadas, no así otro tipo de funciones. Debemos hacer notar que una clase derivada puede ser clase base de otra clase derivada. Con ello, podemos construir una **jerarquía de clases**.

Constructores y destructores

Un punto importante de las clases derivadas es que si la clase base tiene un constructor, éste debe ser llamado, y si éste necesita argumentos, deben ser invocados. Por ejemplo, imaginemos el constructor para Empleado (Código fuente 156).

```
Empleado::Empleado (const string& n, int d) : apellido(n), dept(d)
{ ... }
```

Código fuente 156

El constructor de la clase Manager será el que se observa en el Código fuente 157.

```
Manager::Manager (const string& n, int d, int lvl) : Empleado(n,d), nivel(lvl)
{ ... }
```

Código fuente 157

Copiando elementos

En estos momentos parece lógico preguntarse si es posible la asignación y la copia entre clase y su derivada. En efecto, los constructores de copia y asignación se pueden usar, dando lugar a lo que se llama *slicing* (partición) de una clase. Veamos el Código fuente 158.

```
Manager m;
...
Empleado e = m; // e es la "parte" Empleado de m
```

Código fuente 158

Funciones virtuales

En las secciones anteriores hemos definido funciones en la clase base que pueden ser redefinidas (sobrecargadas) en la clase base. Este polimorfismo puede originar problemas de ejecución cuando en una llamada a una función se llama a la función de la clase base, que no manipula la clase derivada. Para ello, se definen las funciones virtuales, que son definidas obligatoriamente en la clase derivada, y que permitirá al compilador seleccionar la función adecuada según el tipo de objeto. La sintaxis es sencilla:

```
class Empleado {
   // ...
public:
   virtual void print() const;
   // ...
};
```

La principal utilidad de estas funciones es en la manipulación de objetos a través de punteros y referencias.

Clases abstractas

En el diseño de elementos a través de la POO nos encontraremos siempre con la posibilidad de diseñar clases que representan conceptos abstractos para los que no existen objetos. Para estos elementos,

estas clases actúan como un interfaz con sus clases derivadas (nombre que se utiliza en otros lenguajes de programación para las clases abstractas). La definición de una **clase abstracta** es aquella que se declara con **funciones virtuales puras**. Una clase abstracta se declara como indica el Código fuente 159.

```
class Forma {
public:
   virtual void rotar (int) = 0; // funciones virtuales puras
   virtual void dibujar() = 0;
   virtual bool es_cerrada() = 0;
};
```

Código fuente 159

De esta clase Forma se pueden derivar otras clases.

```
class Punto { ...};
class Circulo : public Forma {
public:
   void rotar(int) { }
   void dibujar();
   bool es_cerrada() { return true; }
   Circulo (Punto p, double r);
private:
   double radio;
   Punto centro;
};
```

Código fuente 160

Conviene recordar que de una clase abstracta no se pueden crear objetos o instancias. Debemos hacer notar al alumno que una función virtual pura no definida en la clase derivada permanece como tal, y por lo tanto, la clase derivada continúa siendo abstracta. Por ejemplo, podemos derivar una clase abstracta Poligono de Forma.

```
class Poligono : public Forma {
public:
   bool es_cerrada() { return true; }
};
```

Código fuente 161

Plantillas (*templates*)

Existe un elemento importante dentro de algunos lenguajes de programación como es el soporte a la **programación genérica**, es decir, a la programación usando tipos como parámetros. Como no podía ser menos, C++ soporta este tipo de programación de una forma elegante mediante el uso de plantillas o patrones (*templates*). Estas proporcionan una forma de representar un amplio espectro de conceptos generales, y formas simples de combinarlos. Otros lenguajes orientados a objetos como Java, no poseen esta riqueza de programación implantada de forma tan flexible y potente.

Además de introducir al alumno en los conceptos relacionados con las plantillas, esta sección será de gran utilidad a la hora de comenzar a usar los tipos y algoritmos de la Biblioteca Estándar de C++.

Declaraciones y definiciones de clases con plantillas

La forma de declarar clases con plantillas no introduce ninguna complejidad a la declaración de una clase. Vemos en el Código fuente 162, una clase genérica String que manipula cadenas de distintos tipos de caracteres.

```
template<class T> class String {
   struct Srep;
   Srep *rep;
public:
   String();
   String (const T*);
   String (const String&);
   T read (int i) const;
   // ...
};
```

Código fuente 162

La clase (o tipo; las plantillas no reducen el ámbito de aplicación) T permite definir distintos tipos de String. Los objetos se instancian como indica el Código fuente 163.

```
String<char> cs;
String<wchar_t> ws;
```

Código fuente 163

Dentro de la Biblioteca Estándar existe una clase basic_string, que se puede considerar equivalente a la que estamos tratando de definir. La clase string se obtiene a partir de ésta con una simple redefinición.

```
typedef basic_string<char> string;
```

Código fuente 164

Por otro lado, las definiciones de las funciones miembro son como se podría esperar.

```
template<class T> struct String<T>::Srep {
   T* s; // puntero a los elementos
   int sz; // numero de elementos
   int n; // numero de referencias
};
template<class T> T String<T>::read (int i) const
{
   return rep->s[i];
}
```

```
template<class T> String<T>::String<T>()
{
   p = new Srep(0, T());
}
```

Código fuente 165

La declaración del parámetro en el segundo término del constructor es opcional. Es decir, podemos declararlo como se muestra en el Código fuente 166.

```
template<class T> String<T>::String() { ... }
```

Código fuente 166

Debemos remarcar que no es posible sobrecargar el nombre de una clase con plantillas con otra clase.

Plantillas con parámetros

Es posible definir plantillas con parámetros. Estos parámetros son **constantes** útiles para definir tamaños por defecto, valores máximos, etc. Se declaran como vemos en el Código fuente 167.

```
template<class T, int valor_defecto> class Cont { /* ... */ };
```

Código fuente 167

En el Código fuente 168 definimos un buffer con tamaño.

```
template<class T, int i> class Buffer {
   T v[i];
   int sz;
public:
   Buffer() : sz(i) { };
   // ...
};
```

Código fuente 168

Las llamadas a la clase serán las mostradas en el Código fuente 169.

```
Buffer<char, 100> cbuf;
Buffer<Rectangulo, 4> rbuf;
Buffer<double, i> dbuf; // error
```

Código fuente 169

Funciones con plantilla

La flexibilidad de C++ no se limita a definir clases con plantillas, sino que además posee funciones con plantillas. Imaginemos una función de ordenación de vectores de cualquier tipo de dato.

```
template<class T> void sort (vector<T>&);
```

Código fuente 170

Entonces, cualquier función puede llamar a esta última, seleccionando el compilador el tipo adecuado.

```
void f (vector<int>& vi, vector<string>& vs)
{
   sort(vi); // llama a sort (vector<int>&)
   sort(vs); // llama a sort (vector<string>&)
}
```

Código fuente 171

Como comentario adicional, observemos que la utilización de plantillas permite definir algoritmos genéricos. Una implantación plausible de sort podría ser la mostrada en el Código fuente 172.

```
template<class T> void sort(vector<T>& v)
{ // ordenación shell
   const size_t n = v.size(); // función de STL para el tamaño de un vector
   for (int gap = n/2; gap > 0; gap /= 2)
      for (int i = gap; i < n; i++)
         for (int j = i - gap; gap >= 0; j -= gap)
            if (v[j+gap] < v[j])
            {
               T temp = v[j];
               v[j] = v[j+gap];
               v[j+gap] = temp;
            }
}
```

Código fuente 172

La utilización de la función de la Biblioteca Estándar swap simplifica el Código fuente 172.

```
template<class T> void sort (vector<T>& v)
{ // ordenación shell
   const size_t n = v.size(); // función de STL para el tamaño de un vector
   for (int gap = n/2; gap > 0; gap /= 2)
      for (int i = gap; i < n; i++)
         for (int j = i - gap; gap >= 0; j -= gap)
            if (v[j+gap] < v[j]) swap (v[j], v[j+gap]);
}
```

Código fuente 173

Funciones con plantillas y argumentos

Existen casos en los que la llamada a una determinada clase con plantillas, necesita de sus argumentos constantes. En tal caso, es necesario declarar esos argumentos en la plantilla de la función. Por ejemplo, consideremos una función de búsqueda que necesita llamar a la clase Buffer definida anteriormente. El código correspondiente se define como indica el Código fuente 174.

```
template<class T, int i> T buscando (Buffer<T,i>& bf, const char *p);
```

Código fuente 174

Sobrecarga de funciones con plantilla

Como era de esperar, las funciones con plantilla también se sobrecargan. Por ejemplo, podemos definir funciones raíz cuadrada como muestra el Código fuente 175.

```
template<class T> T sqrt (T);
template<class T> complex<T> sqrt (complex<T>&);
double sqrt (double);
```

Código fuente 175

La resolución para elegir la versión adecuada depende del compilador. En caso de ambigüedad conviene especificar la versión a la que nos referimos.

```
template<class T> T max (T, T);
void k()
{
   max ('a', 1); // error: ambiguo
   max<int> ('a', 1); // correcto: max<int> (int('a'), 1);
}
```

Código fuente 176

Debemos recordar que el uso de inline es recomendable en funciones simples para optimizar los tiempos de ejecución de los programas.

Usando plantillas para especificar políticas

Existen casos, en los que se pueden diseñar algoritmos que no sólo son independientes de los tipos, sino que pueden ser lo suficientemente flexibles como para permitir implantar distintas políticas o caminos de ejecución. Un ejemplo típico es la ordenación y la comparación de cadenas de caracteres. En español y muchos otros idiomas existen criterios de ordenación alfabética distintos del código ASCII, así, desde el punto de vista de ordenación alfabética, los caracteres 'a' y 'á' son iguales, etc. De igual modo, se pueden hacer comparaciones de cadenas independientes de mayúsculas y minúsculas. Para ello, se definen clases adicionales con funciones estáticas que incorporan estos criterios.

```
class Alfabeto { // igualdad entre mayúsculas y minúsculas
public:
   static int igual (char a, char b);
   static int menor (char a, char b);
};
```

Código fuente 177

Si definimos la función de comparación como muestra el Código fuente 178, se puede aplicar como indica el Código fuente 179.

```
template<class T, class C>
   int comparar (const String<char>& s1, const String<char>& s2)
{
   for (int i = 0; i < s1.length() && i < s2.length(); i++)
      if (!C::igual (s1[i], s2[i])) return C::menor(s1[i], s2[i]);
   return s2.length() - s1.length();
}
```

Código fuente 178

```
void f (const String<char>& s1, const String<char>& s2)
{
   comparar<char, Alfabeto> (s1, s2);
}
```

Código fuente 179

También se pueden suministrar argumentos por defecto en las plantillas. Pensando en nuestra función comparar sería deseable que por defecto, la función eligiese la comparación típica.

```
template<class T> class Cmp { // comparación normal
public:
   static int igual (char a, char b) { return a == b; }
   static int menor (char a, char b) { return a < b; }
};
```

Código fuente 180

Con lo que comparar se rescribe como indica el Código fuente 181.

```
template<class T, class C = Cmp<T> >
   int comparar (const String<char>& s1, const String<char>& s2)
{
   for (int i = 0; i < s1.length() && i < s2.length(); i++)
      if (!C::igual (s1[i], s2[i])) return C::menor(s1[i], s2[i]);
   return s2.length() - s1.length();
}
```

Código fuente 181

Nota importante: A pesar de que el estándar admite la declaración de argumentos por defecto en las plantillas, ni Microsoft Visual C++, ni otros compiladores importantes como GNU g++ admiten esta característica del lenguaje.

Manipulación de excepciones

Se puede establecer en general que el autor de una biblioteca o librería de subrutinas es capaz de definir y detectar los errores que puede generara su uso. Sin embargo, no puede anticiparse a los deseos del usuario de la biblioteca para saber como procesar estos errores. C++ facilita una serie de herramientas que permiten la manipulación de errores, a través de excepciones. En el desarrollo de programas existen cuatro tipos de acciones que se suelen tomar para procesar errores:

1. Terminar el programa.
2. Devolver un valor desde la función que represente un error.
3. Devolver un valor legal desde la función, pero dejando al programa en una situación "ilegal" (por ejemplo, con una variable externa que representa un error).
4. Llamar a una función que procese el error.

El mecanismo estándar de C++ está preparado para detectar **excepciones síncronas**: errores de entrada/salida, rangos ilegales en arrays, etc. No contempla eventos asíncronos: interrupciones de teclado, movimientos de ratón, errores aritméticos, etc. que pueden no manipularse necesariamente con este mecanismo. Debemos recordar que el C++ estándar no posee la noción de hilo o proceso, que depende fundamentalmente de la implantación de C++ usada.

Excepciones. Agrupamiento

Una excepción es un objeto de alguna clase que representa la ocurrencia de un error. El código que detecta el error lanza (throw) el objeto que es capturado (catch) por la función que llama a la que detecta el error. Es muy conveniente agrupar excepciones por elementos comunes, y utilizar los mecanismos de herencia. Por ejemplo, podemos crear una clase Matherr para errores matemáticos, de la que derivamos diversas clases.

```
class Matherr { /* ... */ };
class Overflow : public Matherr { /* ... */ }; // error de desbordamiento
class Underflow : public Matherr { /* ... */ }; // número muy pequeño
class Zerodivide : public Matherr { /* ... */ }; // dividir por cero
```

Código fuente 182

Ahora podemos crear una función para la que es importante detectar errores de desbordamiento, pero para la que otro tipo de errores matemáticos caen en una categoría común.

```
void f()
{
   try {
      // ...
```

```
      }
      catch (Overflow) {
         // ...
      }
      catch (Matherr) {
         // ...
      }
   }
```

Código fuente 183

Por supuesto, dentro de las clases usadas como excepciones podemos definir funciones miembro y componentes que nos ayudan a depurar los resultados.

```
#include <iostream> // ejemplo 2
using namespace std;

const int AMAX = 1024, AMIN = -1024;

class Matherr {
public:
  virtual void debug_print() const { cerr << "Error matematico\n"; }
};

class Int_overflow : public Matherr {
  const char* op;
  int a1, a2;
public:
  Int_overflow (const char* p, int a, int b)
  {
    op = p; a1 = a; a2 = b;
  }
  virtual void debug_print() const
  {
    cerr << "Error matematico: " << op << '(' << a1
         << ", " << a2 << ")\n";
  }
};

int add (int x, int y)
{
  if (x + y > AMAX || x + y < AMIN)
    throw Int_overflow ("+", x, y);
  return x+y;
}

int main()
{
  try {
    int i = add(1000, 300);
    cout << "i = " << i << endl;
  }
  catch (Matherr& m) {
    m.debug_print();
  }
}
```

Código fuente 184

Las excepciones capturadas pueden ser lanzadas de nuevo con throw. La instrucción catch (...) captura todas las excepciones no capturadas en catch anteriores. Conviene recordar que la captura de excepciones es secuencial. La estructura try ... catch es completamente análoga a la de switch ... case.

```
void m()
{
   try {
      // ...
   }
   catch (...) {
      // ...
   }
}
```

Código fuente 185

Gestión de recursos

La gestión de recursos del sistema (entrada/salida de archivos, control de acceso, bloqueos (lock), etc.) es un tema que requiere la máxima atención por parte del programador. Es obvio que cuando una función adquiere un recurso, éste debe ser liberado. Veamos un ejemplo en el Código fuente 186.

```
void uso_archivo (const char* fn)
{
   FILE* f = fopen (fn, "w");
   // ...
   fclose (f);
}
```

Código fuente 186

Sin embargo, ¿qué ocurre si se detecta un error y se lanza una excepción antes de llamar a fclose? Obviamente, el recurso no ha sido liberado, y la acumulación de errores de este tipo ocasionará problemas al programa, y probablemente al sistema. Una opción muy utilizada es utilizar "clases envolventes" para los recursos, de forma que el destructor de la clase libere el recurso allá donde sea necesario. Vemos un ejemplo en el Código fuente 187.

```
class File_ptr {
   FILE* p:
public:
   File_ptr (const char* n, const char* a) { p = fopen (n, a); }
   File_ptr (FILE* pp) { p = pp; }
   ~File_ptr() { fclose (p); }
   operator FILE*() { return p; }
};
```

Código fuente 187

También conviene lanzar excepciones en los constructores de las clases para casos de fallo en la inicialización de los elementos. Por ejemplo, podemos definir una clase Vector que tenga como máximo 32000 elementos, como indica el Código fuente 188.

```
class Vector {
   // ...
public:
   class Size { };
   enum { max = 32000 };
   Vector (int sz)
   {
      if (sz < 0 || sz > max) throw Size();
   }
};
```

Código fuente 188

Especificación de excepciones

En general, conviene definir en la declaración de la función, por claridad, que excepciones lanza ésta. De este modo, quedan claras en la definición las condiciones de error que gestiona la función. Así el Código fuente 189, equivale al Código fuente 190.

```
void f() throw (x2, x3)
{
   // ...
}
```

Código fuente 189

```
void f()
try {
   // ...
}
catch (x2) { throw; }
catch (x3) { throw; }
catch (...) { std::unexpected(); // sale del programa, no devuelve el control
}
```

Código fuente 190

Debe quedar claro que las excepciones no capturadas no devuelven el control al programa principal, ejecutando la función de la biblioteca estándar std::terminate(), quien a su vez, llama a abort(), causando la terminación del programa.

Ejercicios

1. Rescribir el ejemplo 1 para la clase Complex, considerando el operador suma como una función no-miembro.

2. Definir interfaces para Guerrero, Monstruo y Objeto (es decir, cosas que puedes recoger o tirar en cada pantalla) para un juego típico de aventuras (sin acciones de ataque o defensa).

3. Consideremos las clases Empleado y Manager definidas en la sección sobre clases derivadas. ¿Es correcta la siguiente definición del constructor para Manager? ¿Porqué?

```
Manager::Manager(const string& n, int d, int lvl) :
   apellido(n), dept(d), nivel(lvl)
{ ... }
```

4. ¿Son correctas las siguientes definiciones a partir de las clases Forma y Circulo definidas en la sección de clases derivadas?

```
Forma f; //
Circulo s, a(p, 2.0); //
```

5. Definir una clase String basada en punteros a un tipo T, y con un indicador de tamaño n. Definir los operadores () y [] de forma que aplicados a String<T> s; s(i), o s[i], devuelvan el elemento i de la cadena.

6. Añadir a String<T> una comprobación en tiempo de ejecución para que cualquier invocación a s(i), o a s[i], compruebe que el rango es correcto.

7. Añadir a String<T> una función para sobrecargar el operador ostream& operator<<, para poder escribir a la salida estándar.

8. Escribir las funciones miembro de la clase Alfabeto de forma que permitan una comparación no-sensible a mayúsculas, minúsculas y acentos, para elementos String<T>. Definir la comparación del siguiente modo: =1, si a > b; =0, si a == b; =-1, si a < b.

9. Crear una clase Char que se corresponda con el tipo char, y que lance una excepción de rango cuando se cree con un entero mayor o menor que el rango permitido (1 byte).

Respuestas a los ejercicios

1. Rescribir el ejemplo 1 para la clase Complex, considerando el operador suma como una función no-miembro.

```
#include <iostream>
using namespace std;

class Complex {
  double re, im;
public:
  Complex (double r, double i) : re(r), im(i) { }
  double Real() const { return this->re; }
  double Imag() const { return this->im; }
};

Complex operator+ (Complex, Complex);

int main()
{
  Complex a(1.1, 0.1), b(0.2, -0.2);
  Complex c = b;
  c = a + b;
```

```
  cout << "c = (" << c.Real() << ", " << c.Imag() << ")" << endl;
}

Complex operator+ (Complex a, Complex b)
{
  Complex tmp(a.Real() + b.Real(), a.Imag() + b.Imag());
  return tmp;
}
```

2. Definir interfaces para Guerrero, Monstruo y Objeto (es decir, cosas que puedes recoger o tirar en cada pantalla) para un juego típico de aventuras (sin acciones de ataque o defensa).

```
#include <vector>
#include <algorithm>

class Coordenadas { };

class ElementoJuego { // clase abstracta
protected:
  Coordenadas r;
public:
  virtual bool es_animado() = 0;
  virtual bool es_transportable() = 0;
  virtual void se_mueve_a (Coordenadas r1) = 0;
};

class Objeto : public ElementoJuego {
public:
  Objeto (Coordenadas r1) { r = r1; }
  Objeto (const Objeto &e) { r = e.r; }
  bool es_animado() { return false; }
  bool es_transportable() { return true; }
  void se_mueve_a (Coordenadas r1) { r = r1; }
};

class Guerrero : public ElementoJuego {
  vector<Objeto> lista;
public:
  bool es_animado() { return true; }
  bool es_transportable() { return false; }
  void se_mueve_a (Coordenadas r1) { r = r1; }
  void coge_a (Objeto e)
  {
    lista.push_back (e);
  }
  void deja_a (Objeto e, Coordenadas r1)
  {
    // funcion para borrar e de lista
    e.se_mueve_a (r1);
  }
};

class Monstruo : public ElementoJuego {
  vector<Objeto> lista;
public:
  bool es_animado() { return true; }
  bool es_transportable() { return false; }
  void se_mueve_a (Coordenadas r1) { r = r1; }
  void coge_a (Objeto e)
  {
    lista.push_back (e);
  }
  void deja_a (Objeto e, Coordenadas r1)
```

```
  {
    // funcion para borrar e de lista
    e.se_mueve_a (r1);
  }
};
```

Código fuente 191

3. Consideremos las clases Empleado y Manager definidas en la sección sobre clases derivadas. ¿Es correcta la siguiente definición del constructor para Manager? ¿Porqué?

```
Manager::Manager(const string& n, int d, int lvl) :
   apellido(n), dept(d), nivel(lvl)
{ ... }
```

Respuesta: No. Porque Manager no tiene acceso a las variables privadas de Empleado. Debe invocarse el constructor de Empleado.

4. ¿Son correctas las siguientes definiciones a partir de las clases Forma y Circulo definidas en la sección de clases derivadas?

```
Forma f; //
```

Respuesta: No. Forma es abstracta

```
Circulo s, a(p, 2.0); //
```

Respuesta: Sí

5. Definir una clase String basada en punteros a un tipo T, y con un indicador de tamaño n. Definir los operadores () y [] de forma que aplicados a String<T> s; s(i), o s[i], devuelvan el elemento i de la cadena.

```
#include <iostream>

template<class T> class String {
  T *s;
  int n;
public:
  String (const T*);
  String (const String&);
  T operator() (int i) const;
  T operator[] (int i) const;
};

template<class T> String<T>::String (const T* c)
{
  int i = 0;
  while (*(c + i) != '\0') { i++; }
  s = (T*)(c);
  n = i;
```

```
}

template<class T> String<T>::String (const String& c)
{
  s = c.s;
  n = c.n;
}

template<class T> T String<T>::operator() (int i) const
{
  return s[i];
}

template<class T> T String<T>::operator[] (int i) const
{
  return s[i];
}

int main()
{
  int i = 2;
  String<char> c("Hola y adios");
  String<char> d(c);

  cout << "c(" << i << ") = " << c(i) << endl;
  cout << "d(" << i+1 << ") = " << d[i+1] << endl;
}
```

6. Añadir a String<T> una comprobación en tiempo de ejecución para que cualquier invocación a s(i), o a s[i], compruebe que el rango es correcto.

```
#include <iostream>

class ErrorRango { };

template<class T> class String {
  T *s;
  int n;
public:
  String (const T*);
  String (const String&);
  T operator() (int i) const;
  T operator[] (int i) const;
  int length () const { return n; }
};

template<class T> String<T>::String (const T* c)
{
  int i = 0;
  while (*(c + i) != '\0') { i++; }
  s = (T*)(c);
  n = i;
}

template<class T> String<T>::String (const String& c)
{
  s = c.s;
  n = c.n;
}

template<class T> T String<T>::operator() (int i) const
{
  if (i >= n) throw ErrorRango();
```

```
  return s[i];
}

template<class T> T String<T>::operator[] (int i) const
{
  if (i >= n) throw ErrorRango();
  return s[i];
}

int main()
{
  int i;
  String<char> c("Hola y adios");

  i = c.length();
  cout << "Tamaño de c: " << i << endl;
  try {
    cout << "c(" << i << ") = " << c(i) << endl;
  }
  catch (ErrorRango) {
    cerr << "Error de rango accediendo c.\n";
  }
}
```

7. Añadir a String<T> una función para sobrecargar el operador ostream& operator<<, para poder escribir a la salida estándar.

```
#include <iostream>

class ErrorRango { };

template<class T> class String {
  T *s;
  int n;
public:
  String (const T*);
  String (const String&);
  T operator() (int i) const;
  T operator[] (int i) const;
  int length () const { return n; }
};

template<class T> String<T>::String (const T* c)
{
  int i = 0;
  while (*(c + i) != '\0') { i++; }
  s = (T*)(c);
  n = i;
}

template<class T> String<T>::String (const String& c)
{
  s = c.s;
  n = c.n;
}

template<class T> T String<T>::operator() (int i) const
{
  if (i >= n) throw ErrorRango();
  return s[i];
}

template<class T> T String<T>::operator[] (int i) const
```

```
{
  if (i >= n) throw ErrorRango();
  return s[i];
}

template<class T> ostream& operator << (ostream& o, const String<T>& s)
{
  for (int i = 0; i < s.length(); i++)
    o << s[i] << ", ";
  o << endl;
  return o;
}

int main()
{
  String<char> c("Hola y adios");

  cout << "String c:\n" << c;
}
```

8. Escribir las funciones miembro de la clase Alfabeto de forma que permitan una comparación no-sensible a mayúsculas, minúsculas y acentos, para elementos String<T>. Definir la comparación del siguiente modo: =1, si a > b; =0, si a == b; =-1, si a < b.

```
#include <iostream>
#include <string>

class ErrorRango { };

template<class T> class String {
  T *s;
  int n;
public:
  String (const T*);
  String (const String&);
  T operator() (int i) const;
  T operator[] (int i) const;
  int length () const { return n; }
};

template<class T> String<T>::String (const T* c)
{
  int i = 0;
  while (*(c + i) != '\0') { i++; }
  s = (T*)(c);
  n = i;
}

template<class T> String<T>::String (const String& c)
{
  s = c.s;
  n = c.n;
}

template<class T> T String<T>::operator() (int i) const
{
  if (i >= n) throw ErrorRango();
  return s[i];
}

template<class T> T String<T>::operator[] (int i) const
{
  if (i >= n) throw ErrorRango();
```

```
  return s[i];
}

template<class T> ostream& operator << (ostream& o, const String<T>& s)
{
  for (int i = 0; i < s.length(); i++)
    o << s[i] << ", ";
  o << endl;
  return o;
}

template<class T> class Cmp { // comparación normal
public:
  static int igual (T a, T b) { return a == b; }
  static int menor (T a, T b) { return a < b; }
};

template<class T> class Alfabeto {
  static T convertir (T x)
  {
    switch (x)
    {
      case 'á': case 'à': case 'ä': case 'â':
      case 'A': case 'Á': case 'À' : case 'Ä': case 'Â':
        x = 'a'; break;
      case 'é': case 'è': case 'ë': case 'ê':
      case 'E': case 'É': case 'È' : case 'Ë': case 'Ê':
        x = 'e'; break;
      case 'í': case 'ì': case 'ï': case 'î':
      case 'I': case 'Í': case 'Ì' : case 'Ï': case 'Î':
        x = 'i'; break;
      case 'ó': case 'ò': case 'ö': case 'ô':
      case 'O': case 'Ó': case 'Ò' : case 'Ö': case 'Ô':
        x = 'o'; break;
      case 'ú': case 'ù': case 'ü': case 'û':
      case 'U': case 'Ú': case 'Ù' : case 'Ü': case 'Û':
        x = 'u'; break;
      default: x = tolower(x);
    }
    return x;
  }
public:
  static int igual (T a, T b)
  {
    return Cmp<T>::igual (convertir (a), convertir (b));
  }
  static int menor (T a, T b)
  {
    return Cmp<T>::menor (convertir (a), convertir (b));
  }
};

template <class T, class C>
  int comparar (const String<T>& s1, const String<T>& s2)
{
  int i1 = s1.length(), i2 = s2.length();
  for (int i = 0; i < i1 && i < i2; i++)
    if (!C::igual (s1[i], s2[i]))
    {
      if (C::menor(s1[i], s2[i])) return -1;
      else return 1;
    }
  if (i2 == i1) return 0;
  else if (i1 < i2) return -1;
  return 1;
}
```

```
int main()
{
  int i;
  const String<char> s1("Camion"), s2("camión");

  cout << "String s1:\n" << s1;
  cout << "String s2:\n" << s2;

  i = comparar<char, Cmp<char> > (s1, s2);
  cout << "Comparación ASCII: " << i << endl;
  i = comparar<char, Alfabeto<char> > (s1, s2);
  cout << "Comparación alfabética: " << i << endl;

  return 0;
}
```

9. Crear una clase Char que se corresponda con el tipo char, y que lance una excepción de rango cuando se cree con un entero mayor o menor que el rango permitido (1 byte).

```
#include <iostream>

class RangeError { };

class Char {
  char c;
public:
  Char() { c = 0; }
  Char (unsigned int i)
  {
    if (i > 0xff || i < 0) throw RangeError();
    c = i;
  }
  char read() const { return c; }
};

ostream& operator << (ostream& o, const Char& s)
{
  o << s.read();
  return o;
}

int main()
{
  try {
    int x;
    cout << "Introduce un entero para imprimir el carácter correspondiente:
";
    cin >> x;
    Char c(x);
    cout << "El carácter es: " << c.read() << endl;
  }
  catch (RangeError) {
    cerr << "Error en la creación del Char\n";
  }

  return 0;
}
```

11

La biblioteca estándar de C++

Introducción

Hasta la aparición de la **Biblioteca Estándar de Plantillas** (*Standard Template Library, STL*) a mediados de los 90, la programación utilizando C++ requería en gran medida la utilización de bibliotecas comerciales, o bien gran pericia para desarrollar una biblioteca propia de utilidades. La gran diversidad de plataformas existentes hacía además difícil optimizar el código para cada una de ellas. La STL posee las siguientes características:

- Proporciona soporte para utilidades como gestión de memoria, e información en tiempo de ejecución.
- Proporciona información sobre la implantación del lenguaje utilizada (límites numéricos, etc.).
- Proporciona versiones optimizadas de funciones como sqrt(), y otras.
- Proporciona versiones portables de listas, mapas, algoritmos de ordenación y facilidades de entrada/salida.
- Proporciona una estructura para extender las facilidades proporcionadas.
- Proporciona generadores de números aleatorios y otros elementos matemáticos.

Organización de la biblioteca estándar

Las facilidades de la biblioteca estándar se agrupan en el *namespace* std, y se invocan a través de una serie de encabezados.

Se agrupan según su función (de la Tabla 9 a la Tabla 18).

Contenedores	
<vector>	array unidimensional de tipo T
<list>	lista doblemente enlazada de tipo T
<deque>	cola de doble fin de tipo T
<queue>	cola de tipo T
<stack>	pila de tipo T
<map>	array asociativo de tipo T
<set>	conjunto de tipo T
<bitset>	conjunto de boléanos

Tabla 9

Utilidades generales	
<utility>	operadores y pares
<functional>	objetos funciones
<memory>	gestores de espacio para contenedores
<ctime>	hora y fecha estilo C

Tabla 10

Iteradores	
<iterator>	Iteradores

Tabla 11

Algoritmos	
<algorithm>	algoritmos generales

<cstdlib>	bsearch(), qsort()

Tabla 12

Diagnósticos	
<stdexcept>	excepciones estándar
<cassert>	macro assert
<cerrno>	manipulación de errores estilo-C

Tabla 13

Cadenas	
<string>	cadena de tipo T
<cctype>	clasificación de caracteres
<cwtype>	caracteres "anchos" (ISO-xxxx)
<cstring>	cadenas tipo-C
<cwchar>	caracteres "anchos" tipo-C
<cstdlib>	cadenas tipo-C

Tabla 14

Entrada/Salida	
<iosfwd>	declaradores de E/S
<iostream>	operaciones estándar E/S
<ios>	base de <iostream>
<streambuf>	stream con búfer
<istream>	operaciones estándar Entrada
<ostream>	operaciones estándar Salida
<iomanip>	manipuladores
<sstream>	streams de cadenas
<fstream>	streams para archivos

<cstdio>	soporte para la familia printf()
<cwchar>	soporte para la familia printf() con caracteres anchos

Tabla 15

Localización	
<locale>	representación de distintos idiomas
<clocale>	representación de distintos idiomas estilo-C

Tabla 16

Soporte al Lenguaje	
<limits>	límites numéricos
<climits>	límites numéricos estilo-C
<cfloat>	límites numéricos estilo-C para números reales
<new>	gestión dinámica de memoria
<typeinfo>	información en tiempo de ejecución sobre tipos
<exception>	manipulación de excepciones
<cstddef>	soporte al lenguaje C
<cstdarg>	listas variables de argumentos
<csetjmp>	pilas tipo-C
<cstdlib>	terminación de programas
<ctime>	reloj del sistema
<csignal>	manipulación de señales tipo-C

Tabla 17

Cálculo Numérico	
<complex>	números complejos
<valarray>	vectores numéricos
<numerics>	operaciones numéricas generales

<cmath>	funciones matemáticas estándar
<cstdlib>	números aleatorios tipo-C

Tabla 18

Por razones históricas, abs(), fabs() y div() se encuentran en <cstdlib>.

Vectores

Los vectores son un tipo estándar de contenedor para objetos de cualquier tipo (se puede pensar en un array de elementos con facilidades adicionales). Para navegar sobre un vector, además de la referencia al índice, disponemos de los llamados iteradores, iterator, que se comportan como punteros a los elementos del vector. Para obtener los elementos iniciales y finales, debemos usar las funciones vector.begin() y vector.end(). Así, para navegar por un vector v, el procedimiento es el que muestra el Código fuente 192.

```
   vector<int> v[10];
   vector<int>::iterator it;
   for (it = v.begin(); it != v.end(); it++) cout << *it;
```

Código fuente 192

El Código fuente 193 ofrece otros ejemplos típicos.

```
#include <iostream> // ejemplo 1
#include <vector>

int main ()
{
  vector<int> v1; // vector vacio de enteros
  cout << "vacio? = " << v1.empty () << endl;
  cout << "tamaño = " << v1.size () << endl;
  cout << "tamaño maximo = " << v1.max_size () << endl;
  v1.push_back (42); // añade un elemento
  cout << "tamaño = " << v1.size () << endl;
  cout << "v1[0] = " << v1[0] << endl;
  return 0;
}
```

Código fuente 193

El Código fuente 194 ilustra algunas funciones adicionales:

```
#include <iostream> // ejemplo 2
#include <vector>

void print (vector<double>& v)
{
  for (int i = 0; i < v.size (); i++)
```

```
    cout << v[i] << " ";
  cout << endl;
}

int main ()
{
  vector<double> v1; // vector vacio de dobles
  v1.push_back (32.1);
  v1.push_back (40.5);
  vector<double> v2; // otro vector vacio
  v2.push_back (3.56);
  cout << "v1 = ";
  print (v1);
  cout << "v2 = ";
  print (v2);
  v1.swap (v2); // intercambia v1 y v2
  cout << "v1 = ";
  print (v1);
  cout << "v2 = ";
  print (v2);
  v2 = v1; // asigna un vector a otro
  cout << "v2 = ";
  print (v2);
  return 0;
}
```

Código fuente 194

Observe cómo se utilizan en funciones estos elementos, y su asignación. El cambio de tamaño en tiempo de ejecución les dota de gran flexibilidad. Las funciones de borrado se utilizan en el Código fuente 195.

```
#include <iostream> // ejemplo 3
#include <vector>

int array [] = { 1, 4, 9, 16, 25, 36 };

int main ()
{
  int i;
  vector<int> v (array, array + 6);
  for (i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  v.erase (v.begin ()); // borra el primer elemento
  for (i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  v.erase (v.end () - 1); // borra el ultimo elemento
  for (i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  v.erase (v.begin () + 1, v.end () - 1); // borra todo menos primero y ultimo
  for (i = 0; i < v.size (); i++)
    cout << "v[" << i << "] = " << v[i] << endl;
  cout << endl;
  v.erase (v.begin (), v.end ()); // borra todo
  return 0;
}
```

Código fuente 195

Además de los vectores descritos, la STL posee otros muchos tipos de contenedores: listas (list), mapas (map), pilas (stack), colas (queue), etc. Recomendamos al alumno la lectura de la ayuda de Microsoft Visual C++ para cada uno de estos tipos. Es difícil encontrar el diseño de nuevos contenedores que no estén contemplados en la biblioteca estándar.

Algoritmos y funciones

Un conjunto de contenedores aislados no tiene en sí mucho interés si no está complementado con un conjunto básico de funciones y algoritmos para copiar, buscar, y ordenar elementos. STL posee estos elementos, y la flexibilidad suficiente para personalizar muchos de ellos. Los algoritmos se expresan como funciones con plantillas. Los algoritmos que devuelven un iterador, utilizan el final de la secuencia para indicar fallo. Vemos un ejemplo en el Código fuente 196.

```
void f (list<string>& ls)
{
   list<string>::const_iterator p = find (ls.begin(), ls.end(), "Fred");
   if (p == ls.end())
   { // no se encuentra "Fred"
   }
   else
   { // p apunta a "Fred"
   }
}
```

Código fuente 196

Los algoritmos se encuentran en el encabezado <algorithm>, y los objetos de funciones en <functional>. Los algoritmos no-modificadores, que se aplican a la secuencia indicada aparecen en la Tabla 19 (donde predicado equivale a una expresión lógica). Los algoritmos modificadores se muestran en la Tabla 20

Algoritmos no-modificadores	
for_each()	realiza la operación en cada elemento
find()	encuentra la primera ocurrencia de un valor
find_if()	encuentra la primera ocurrencia de un predicado
find_first_of()	encuentra el valor de una secuencia en otra
adjacent_find()	encuentra un par de valores adyacentes
count()	cuenta las ocurrencias de un valor
count_if()	cuenta las ocurrencias de un predicado
mismatch()	encuentra los primeros elementos para los que difiere una secuencia

equal()	verdadero si los elementos de dos secuencia son iguales
search()	encuentra la primera ocurrencia de una secuencia como subsecuencia
find_end()	encuentra la última ocurrencia de una secuencia como subsecuencia
search_n()	encuentra la enésima ocurrencia de un valor

Tabla 19

Algoritmos modificadores	
transform()	aplica una operación a toda la secuencia
copy()	copia una secuencia
copy_backward()	copia una secuencia en orden inverso
swap()	intercambia dos elementos
iter_swap()	intercambia dos elementos apuntados con iteradores
swap_ranges()	intercambia elementos de dos secuencias
replace()	reemplaza elementos con un valor dado
replace_if()	reemplaza elementos con un valor dado si coinciden con un predicado
replace_copy()	copia una secuencia, reemplazando elementos con un valor dado
replace_copy_if()	copia una secuencia, reemplazando elementos con un valor dado si coinciden con un predicado
fill()	reemplaza cada elemento con un valor dado
fill_n()	reemplaza n elementos con un valor dado
generate()	reemplaza cada elemento con el resultado de una operación
generate_n()	reemplaza n elementos con el resultado de una operación
remove()	borra elementos con un valor dado
remove_if()	borra elementos que coinciden con un predicado
remove_copy()	copia una secuencia eliminando elementos con un valor dado
remove_copy_if()	copia una secuencia eliminando elementos que coinciden con un predicado

unique()	elimina elementos adyacentes iguales
unique_copy()	copia una secuencia eliminando elementos adyacentes iguales
reverse()	invierte el orden de los elementos
reverse_copy()	copia una secuencia en orden inverso
rotate()	rotar elementos
rotate_copy()	copia una secuencia en una secuencia rotada
random_shuffle()	mueve los elementos uniformemente

Tabla 20

Además se dispone de algoritmos de ordenación (Tabla 21).

Algoritmos de ordenación	
sort()	ordenación con eficiencia media
stable_sort()	ordenación manteniendo el orden de elementos iguales
partial_sort()	ordena la primera parte de la secuencia
partial_sort_copy()	copia obteniendo la primera parte de la salida en orden
nth_element()	sitúa el enésimo elemento en su orden adecuado
lower_bound()	encuentra la primera ocurrencia de un valor
upper_bound()	encuentra la última ocurrencia de un valor
equal_range()	encuentra una subsecuencia con un valor dado
binary_search()	encuentra si un valor está dentro de una secuencia
merge()	funde dos secuencias ordenadas
inplace_merge()	funde dos subsecuencias consecutivas ordenadas
partition()	sitúa elementos que coinciden con un predicado primero
stable_partition()	sitúa elementos que coinciden con un predicado primero, manteniendo el orden relativo

Tabla 21

También encontramos algoritmos de máximo y mínimo, los vemos en la Tabla 22.

Algoritmos de máximo y mínimo	
min()	el menor de dos valores
max()	el mayor de dos valores
min_element()	el menor valor de una secuencia
max_element()	el mayor valor de una secuencia
lexicographical_compare()	comparación lexicográfica

Tabla 22

Además, existen algoritmos para manipular conjuntos, pilas, etc. La forma de invocar estos elementos aparece en el Código fuente 197.

```
   list<string>::const_iterator LI;
   list<string> frutas;
   ...
   LI p1 = find (frutas.begin(), frutas.end(), "pera");
```

Código fuente 197

El Código fuente 198 muestra el uso de count.

```
#include <algorithm> // ejemplo 4
#include <iostream>
#include <vector>

int main ()
{
  vector<int> i;
  i.push_back (1);
  i.push_back (4);
  i.push_back (2);
  i.push_back (8);
  i.push_back (2);
  i.push_back (2);
  int n = 0; // n debe tener un valor inicial nulo
  count (i.begin(), i.end(), 2, n);
  cout << "2 aparece " << n << " veces." << endl;
  return 0;
}
```

Código fuente 198

Predicados

Como era lógico suponer, STL proporciona una lista de predicados estándar en <functional>. Estos son los que aparecen en la Tabla 23.

Predicados	
equal_to	arg1==arg2
not_equal_to	arg1!=arg2
Greater	arg1>arg2
Less	arg1<arg2
greater_equal	arg1>=arg2
less_equal	arg1<=arg2
logical_and	arg1&&arg2
logical_or	arg1||arg2
logical_not	!arg

Tabla 23

Por ejemplo, la función transform, aplicada a dos arrays de enteros se puede expresar como muestra el Código fuente 199.

```
#include <algorithm> // ejemplo 5
#include <functional>
#include <iostream>

int input1 [4] = { 1, 7, 2, 2 };
int input2 [4] = { 1, 6, 2, 3 };

int main ()
{
  int output [4];
  transform (input1, input1 + 4, input2, output, equal_to<int> ());
  for (int i = 0; i < 4; i++)
    cout << output[i] << endl;
  return 0;
}
```

Código fuente 199

Cadenas (*strings*)

Un string es una cadena de caracteres. Debido a su uso extensivo, STL proporciona los elementos para manipularlas, tales como subíndices, asignación, comparación, adición, concatenación y búsqueda. Todas las cadenas están basadas en la clase basic_string. string es un basic_string de chars. Con las cadenas se pueden utilizar los algoritmos descritos en la sección anterior. También se puede acceder a sus elementos a través de índices. Veamos el Código fuente 200.

```
string s = "hola";
char t = s[0]; // t = h
```

Código fuente 200

Las cadenas se pueden inicializar con una cadena tipo-C, con otra cadena, o con una secuencia de caracteres únicamente. Por supuesto, existe la posibilidad de asignar cadenas, y de concatenar cadenas con el operador '+'.

```
string s1 = "hola", s2 = "mundo";
string s = s1 + ' ' + s2; // s = "hola mundo"
```

Código fuente 201

Archivos (*streams*)

Para cerrar este repaso rápido a la biblioteca estándar, vamos a estudiar la entrada y salida en C++ a través de los *streams*. A lo largo del curso de C++ hemos usado con profusión la entrada y salida estándar a través de cout y cin.

Como ya ocurriera en la parte de C vamos a estudiar formas más flexibles de introducir y almacenar datos. Al igual que C, en C++ disponemos de los tres tipos básicos de E/S:

- cin: entrada estándar (teclado)
- cout: salida estándar con búfer (pantalla)
- cerr: salida de error estándar sin búfer (pantalla)
- clog: salida de error estándar sin búfer (pantalla)

Las cuatro pueden ser redireccionadas dependiendo del sistema operativo. También existen sus versiones para caracteres "anchos": wcin, wcout, wcerr, wclog.

Salida

Se podría pensar que la solución más sencilla para implantar mecanismos de entrada/salida era sobrecargar las distintas funciones de E/S de C: printf, scanf, putc, getc, etc. Sin embargo, aunque las funciones anteriores están disponibles por cuestión de compatibilidad, C++ ofrece una solución más elegante a las funciones de E/S a través de los *streams* y sus operadores de entrada >> y salida <<.

La estructura de esas clases es bastante compleja, y no la vamos a discutir aquí. Todas las clases derivan de los tipos básicos basic_istream y basic_ostream, declarados en <ios>.

Todos los tipos intrínsecos de C++ tienen estos elementos sobrecargados, imprimiéndose cada elemento de forma adecuada. Los elementos booleanos se imprimen por defecto como 0 ó 1, a no ser que usemos el modificador boolalpha.

```
#include <iostream> // ejemplo 6
#include <iomanip>

int main ()
{
  bool i = false, j = true;
  cout << boolalpha << "i: " << i << " - j: " << j << endl;
  return 0;
}
```

Código fuente 202

Los punteros se imprimen por defecto como números hexadecimales. Para definir nuestras propias funciones de impresión, basta con sobrecargar cout. Por ejemplo, para números complejos podríamos definirlo como indica el Código fuente 203.

```
ostream& operator<< (ostream& s, complex z)
{
   return s << '(' << z.real() << ',' << z.imag() << ')';
}
```

Código fuente 203

Debemos destacar que es conveniente declarar en las clases abstractas funciones virtuales de salida, para obligar a implantar las operaciones de salida en las clases derivadas.

Entrada

Al igual que ocurría con la salida de datos, todos los tipos intrínsecos de C++ tienen definidos su función de entrada con cin. Para la lectura de un único carácter, se lee el primer carácter no-blanco introducido por el teclado. Para leer cualquier carácter usar get. Tanto en operaciones de entrada como de salida, conviene comprobar el estado del stream. Esto es especialmente crítico en operaciones de entrada, ya que la no comprobación puede conducir a la lectura de datos erróneos. Estas funciones son:

- good(): buen estado
- eof(): fin de entrada
- fail(): la siguiente operación fallará
- bad(): stream en malas condiciones
- rdstate(): obtiene el estado del stream
- clear(): establece las condiciones del stream
- setstate(iostate f): añade f a las condiciones del stream

Los bits que definen el estado son: badbit, eofbit, failbit, y goodbit. Un stream se puede usar como condición.

```
T buf;
while (is >> buf) os << buf << '\n';
```

Código fuente 204

Para entrada de caracteres conviene usar las funciones: get(char*, int tamaño, char terminador); o getline con la misma sintaxis. Para leer un único carácter, usar get(char&);. tamaño y el carácter terminador son opcionales, pero conviene especificarlos.

Formateo

El formateo de la entrada y salida se obtiene a través de bits de estado denominados fmtflags de la clase ios_base (o ios). Estos se establecen mediante las funciones flags() y setf(). Por ejemplo, para establecer salida decimal, octal, o hexadecimal, tenemos el Código fuente 205.

```
cout << 1234 << ' ';
cout.setf(ios::oct, ios::basefield); cout.setf(ios::showbase);
cout << 1234 << ' ';
cout.setf(ios::hex, ios::basefield); cout.setf(ios::showbase);
cout << 1234 << ' ' << endl;
```

Código fuente 205

La instrucción cout.setf(ios::showbase); muestra la base en la que se escribe el número decimal. Para números reales, es necesario definir el formato y la precisión. Veamos el Código fuente 206.

```
cout << 1234.8899 << ' ';
cout.setf(ios::scientific, ios::floatfield); // formato científico
cout << 1234.8899 << ' ';
cout.setf(ios::fixed, ios::floatfield); // formato punto fijo
cout << 1234.8899 << ' ';
cout.setf(0, ios::floatfield);
cout << 1234.8899 << ' ' << endl;
```

Código fuente 206

La precisión se define con la función precision(int). Por ejemplo veamos el Código fuente 207. El ancho del campo se define con la función width(int), como vemos en el Código fuente 208.

```
cout.precision(8); // define un campo con 8 decimales
```

Código fuente 207

```
cout.width(3); // define un campo de 3 elementos
```

Código fuente 208

Para definir el carácter de relleno se usa la función fill(char), como se observa en el Código fuente 209.

```
cout.width(4); cout.fill('#');
cout << '(' << "ab" << ')'; // produce (##ab)
```

Código fuente 209

La alineación se expresa como muestra el Código fuente 210.

```
cout.setf(ios::left, ios::adjustfield); // izquierda
cout.setf(ios::left, ios::adjustfield); // derecha
cout.setf(ios::left, ios::adjustfield); // interna
```

Código fuente 210

Existen una serie de manipuladores que se pueden insertar cuando se está enviando la salida al correspondiente stream en lugar de tener que usar setf. Estos son:

- internal, left o right
- showpoint o noshowpoint
- showbase o noshowbase
- oct, hex o dec
- endl
- ends
- flush
- setbase
- setfill
- setprecision
- setw

El Código fuente 211 muestra su uso.

```
#include <iostream> // ejemplo 7
#include <iomanip>

int main ()
{
```

```
  cout << 12 << ' ' << setw(4) << setfill('#') << 12 << endl;
  cout << 1234.9222 << ' ' << setw(12)
       << setprecision(2) << 1234.9222 << endl;
  return 0;
}
```

Código fuente 211

Archivos

Vamos a ver ahora la definición de archivos. Para ello debemos considerar la biblioteca estándar <fstream>. Consideremos el Código fuente 212, que copia el primer argumento de la línea de comandos en el segundo.

```
#include <fstream> // ejemplo 8

int main (int argc, char* argv[])
{
  if (argc != 3)
  {
    cerr << "Número erróneo de argumentos\n";
    std::exit(1);
  }
  std::ifstream from(argv[1]);
  if (!from) cerr << "No puedo abrir " << argv[1] << endl;
  std::ofstream to(argv[2]);
  if (!to) cerr << "No puedo abrir " << argv[2] << endl;
  char c;
  while (from.get(c)) to.put(c);
  if (!from.eof() || !to) cerr << "Error!" << endl;
  return 0;
}
```

Código fuente 212

Vemos que existen archivos de entrada (ifstream) y de salida (ofstream). También existe un tipo genérico fstream, que acepta un segundo argumento miembro de ios_base y de ios:

- app: abrir para añadir
- ate: abrir y buscar el fin de archivo
- binary: abrir en modo binario
- in: archivo de entrada
- out: archivo de salida
- trunc: truncar el archivo a 0 bytes

Un ejemplo lo veríamos en el Código fuente 213.

```
fstream diccionario("mi_diccionario.bin", ios::app | ios::bin);
```

Código fuente 213

No olvidemos que conviene cerrar los archivos usados, para evitar que los recursos se coman la memoria:

```
diccionario.close();
```

C++ también dispone de elementos de entrada/salida con búfer, que no vamos a considerar en este curso.

Hemos descrito en este capítulo los elementos principales de la biblioteca estándar. El cubrir cada elemento en detalle podría llevar fácilmente todo un curso completo. La intención de este capítulo es facilitar al alumno las herramientas básicas, así como una indicación de cómo buscar información sobre herramientas adicionales.

Ejercicios

4. Crear un programa que lea el nombre de un usuario en un string, lo convierta en un vector, y elimine las vocales.

5. Crear una lista (list) con nombres de fruta (pera, manzana, plátano, melocotón). Encontrar los elementos que comiencen con la letra 'p'.

6. Utilizar el algoritmo find para buscar pera.

7. Crear un vector de años aleatorios y desordenados. Ordenarlo usando sort.

8. Modificar el ejemplo 5 para que output refleje aquellos valores de input1 mayores que input2.

9. Usar for_each para imprimir todos los elementos de un vector.

10. Escribir en C++ el ejercicio 10 del capítulo 6 de la parte de C para contar el número de caracteres y de líneas de todos los archivos introducidos por la línea de comandos.

Respuestas a los ejercicios

1. Crear un programa que lea el nombre de un usuario en un string, lo convierta en un vector, y elimine las vocales.

```
#include <iostream>
#include <string>
#include <vector>

int main ()
{
  int i;
  string s;
  vector<char> v, w;
```

```
  vector<char>::iterator it;

  cout << "Introduce tu nombre: ";
  cin >> s;
  for (i = 0; i < s.length(); i++) v.push_back (s[i]);
  for (it = v.begin(); it != v.end(); it++)
    switch (*it)
      {
      case 'a': case 'e': case 'i': case 'o': case 'u':
      case 'A': case 'E': case 'I': case 'O': case 'U':
        break;
      default:
        w.push_back (*it);
      }
  cout << "Hola, ";
  for (it = w.begin(); it != w.end(); it++) cout << *it;
  cout << endl;
  return 0;
}
```

2. Crear una lista (list) con nombres de fruta (pera, manzana, plátano, melocotón). Encontrar los elementos que comiencen con la letra 'p'.

```
#include <iostream>
#include <string>
#include <list>

int main ()
{
  int i;
  string s[] = { "pera", "manzana", "platano", "melocoton" };
  list<string> l;
  list<string>::iterator it;

  for (i = 0; i < 4; i++) l.push_back (s[i]);
  cout << "Lista completa:\n";
  for (it = l.begin(); it != l.end(); it++)
    cout << *it << " ";
  cout << endl;
  cout << "Empiezan por p:\n";
  for (it = l.begin(); it != l.end(); it++)
    if ((*it)[0] == 'p')
      cout << *it << " ";
  cout << endl;
  return 0;
}
```

3. Utilizar el algoritmo find para buscar pera.

```
#include <algorithm>
#include <iostream>
#include <string>
#include <list>

int main ()
{
  int i;
  string s[] = { "pera", "manzana", "platano", "melocoton" };
  list<string> l;
  list<string>::iterator it;
```

```
  for (i = 0; i < 4; i++) l.push_back (s[i]);
  cout << "Lista completa:\n";
  for (it = l.begin(); it != l.end(); it++)
    cout << *it << " ";
  cout << endl;
  it = find (l.begin(), l.end(), "pera");
  cout << "Elemento con peras: " << *it << endl;
  return 0;
}
```

4. Crear un vector de años aleatorios y desordenados. Ordenarlo usando sort.

```
#include <algorithm>
#include <iostream>
#include <vector>

int main ()
{
  vector<long> yrs;
  yrs.push_back (1962);
  yrs.push_back (1992);
  yrs.push_back (2001);
  yrs.push_back (1999);
  sort (yrs.begin(), yrs.end());
  vector<long>::iterator i;
  for (i = yrs.begin (); i != yrs.end (); i++)
    cout << *i << endl;
  return 0;
}
```

5. Modificar el ejemplo 5 para que output refleje aquellos valores de input1 mayores que input2.

```
#include <algorithm>
#include <functional>
#include <iostream>
int input1 [4] = { 1, 7, 2, 2 };
int input2 [4] = { 1, 6, 2, 3 };

int main ()
{
  int output [4];
  transform (input1, input1 + 4, input2, output, greater<int> ());
  for (int i = 0; i < 4; i++)
    cout << output[i] << endl;
  return 0;
}
```

6. Usar for_each para imprimir todos los elementos de un vector.

```
#include <algorithm>
#include <iostream>
#include <vector>

void print (int a)
{
  cout << a << ' ';
}
```

```
int main ()
{
  vector<int> v(10);
  fill (v.begin(), v.end(), 1);
  for_each (v.begin(), v.end(), print);
  cout << endl;
  return 0;
}
```

7. Escribir en C++ el ejercicio 10 del capítulo 6 de la parte de C para contar el número de caracteres y de líneas de todos los archivos introducidos por la línea de comandos.

```
#include <fstream>

void filecount (char *, char *);

int main (int argc, char *argv[])
{
  char *program_name = argv[0];

  if (argc == 1) /* sin argumentos: mensaje de error */
  {
    cerr << program_name << ": Llamada sin argumentos\n"
         << "Uso: \'f_count lista_ficheros\'\n";
    exit (1);
  }
  else
    while (--argc > 0)
      filecount (program_name, *++argv);

  return 0;
}

/* Función void filecount: cuenta los caracteres y
   las líneas del fichero introducido */
void filecount (char *program_name, char *file_name)
{
  std::ifstream in(file_name);
  int nc = 0, nl = 1;
  char c;
  if (!in)
  {
    cerr << program_name << ": error abriendo " << file_name << endl;
    exit (1);
  }
  while (in.get(c))
  {
    ++nc;
    if (c == '\n') ++nl;
  }
  in.close();
  cout << file_name << ':' << endl << '\t' << nc << " caracteres\n"
       << '\t' << nl << " lineas\n";
  if (!cout)
  {
    cerr << "Error escribiendo en cout\n";
    exit (2);
  }
}
```

12

Visual C++. Un entorno de desarrollo integrado

Introducción

Los capítulos que siguen presentan al alumno una serie de temas relacionados con Microsoft Visual C++. Visual C++ es lo que se conoce en el mundo del desarrollo de software como un Entorno de Desarrollo Integrado (*Integrated Development Environment*, IDE). Los IDE deben su nombre al hecho de permitir desarrollos de software en un entorno que permite el control total del proceso: desde la compilación, al enlace con bibliotecas, pasando por el desarrollo de interfaces gráficos de usuario.

En los capítulos anteriores hemos desarrollado los aspectos relacionados con la parte estandarizada de C y C++. En esa parte ya hemos mencionado que no existe una biblioteca estándar para entornos gráficos. Aquí es donde interviene Visual C++, proporcionando a través de las *Microsoft Foundation Classes* (MFC) todas las herramientas necesarias para el desarrollo de Interfaces de Usuario Gráficos (*Graphical User Interfaces*, GUIs) en el entorno Windows 95/98/NT. Pero la potencia de Visual C++, en combinación con las MFC, no se detiene ahí. Además de la mejora en la versión 6.0 de las clases GUI como Extended Combo Box, Slider, Toolbar, Tab, Status Bar, Image List, Hot Key, Progress Bar y Tool Tip, proporciona la posibilidad de creación de controles Active X, la inclusión de documentos con HTML dinámico, multitud de clases para el control de Internet (direcciones IP, etc.), acceso a contenedores OLE (intercambio de documentos entre aplicaciones), y acceso a bases de datos (Microsoft Universal Data Access). Por otro lado, Visual C++ incorpora un conjunto de clases nuevas en la *Active Template Library* (ATL, Biblioteca de Plantillas Activa) para desarrollar controles Active X y otras aplicaciones que consuman pocos recursos.

Como el alumno podrá entender, el desarrollo en detalle de todos las posibilidades de Visual C++ llevaría fácilmente un curso de 400 horas, por lo que en esta introducción nos centraremos en los conceptos básicos que le permitirán profundizar en otras utilidades de Visual C++ a través de la ayuda en pantalla y tutoriales existentes, así como de los Wizards (Magos para ayuda).

Instalación

El primer paso antes de utilizar Microsoft Visual C++ es instalar el software. La instalación de Visual C++ no reviste mayor complejidad que cualquier otra aplicación de Microsoft. Como el lector sabrá, Visual C++ puede ser adquirido por separado, o como parte de la suite Visual Studio. Los pasos a seguir son:

- Introducir el CD-Rom de Instalación (el propio CD-Rom arrancará el programa de instalación). Seleccionar el producto a instalar, en el caso de haber adquirido la suite completa. Observe que el interfaz de Visual C++ no se encuentra traducido al español.

- Introducir los datos del usuario y la licencia.

- Elegir el directorio de instalación (en el caso de que se quiera modificar la selección por defecto), y comprobar que existe suficiente espacio en el disco duro.

- Seleccionar los componentes a instalar (para el caso de usuarios noveles no modificar las opciones por defecto).

- Ejecutar la instalación y reiniciar el equipo.

Es importante reseñar que es posible modificar (añadir y eliminar) componentes a posteriori con el programa de instalación. El alumno debe ser consciente de que existen una serie elementos comunes a toda la suite Visual Studio, cuya ubicación en el disco duro es detectada normalmente por el programa de instalación. En caso de problemas, contacte con su distribuidor de software o con Microsoft.

Visual C++ incorpora en sus últimas versiones una versión de su *Knowledge Base* (Base de Datos de Conocimientos), de *Microsoft Developer Network* (MSDN), de la que se puede instalar una parte en el disco duro. Estos CD-Roms, además de la Ayuda On-line, incorporan tutoriales, libros, programas de ejemplo, etc.

Nuestro primer programa

Después de instalar el programa, éste se arranca fácilmente desde Inicio + Programas + Microsoft Visual C++ 6.0 + Microsoft Visual C++ 6.0. Una vez arrancado el programa, aparecerá el Consejo del Día (*Tip of the Day*), que contiene notas rápidas escritas por los desarrolladores de C++ con consejos para sacar mayor provecho del entorno. Si se considera molesto, ésta característica puede ser desactivada con la casilla de selección *Show Tips at startup*.

Una vez en el entorno, nos encontramos con una pantalla como la que muestra la Figura 4. En ella vemos los distintos elementos del entorno: la barra de título, los menús, una o varias filas con distintas herramientas (barra de herramientas), y el área de trabajo dividida en dos zonas. La parte izquierda contiene información sobre el proyecto: archivos, clases, elementos gráficos, etc., y la parte derecha se utiliza para editar el código correspondiente.

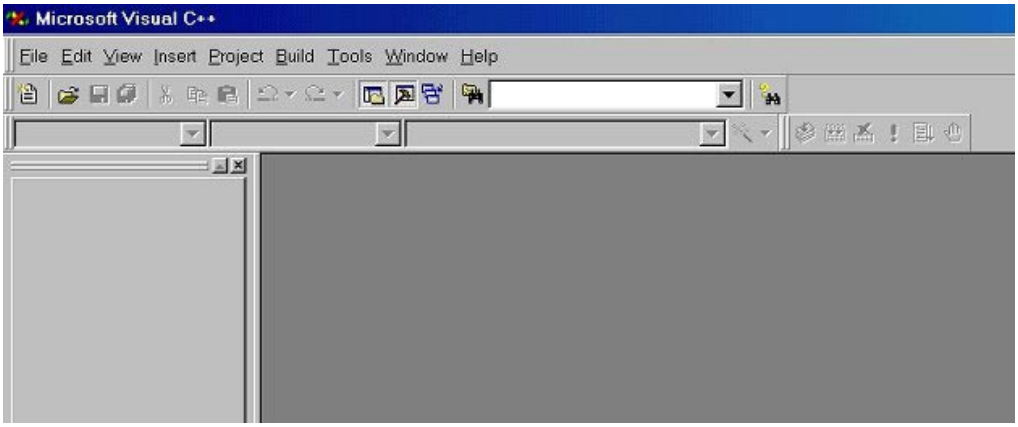


Figura 4

El lector debe recordar que la unidad fundamental de trabajo con Visual C++ es el proyecto. En los capítulos anteriores hemos trabajado con programas y funciones que se podían agrupar en bibliotecas. En los IDE, el proyecto contiene información sobre todos los elementos del programa: archivo fuente, archivos "include", bibliotecas para enlazado, etc.

Nuestro primer proyecto va a ser un "Hola Mundo 1". Para ello elegimos en el menú File + New. Esta selección nos lleva hasta el AppWizard (Programa de ayuda para generar el esqueleto para distintas aplicaciones). Este "mago" ofrece distintos tipos de aplicaciones. Seleccionamos MFC App Wizard (exe) para crear un programa ejecutable MFC, y le asignamos un nombre al proyecto (Project name). No olvide seleccionar en Location el subdirectorio donde situar los archivos del proyecto en su disco duro. No cambie las opciones por defecto en el resto de las pantallas que le aparecerán, con la excepción de seleccionar Single Document Interface (Interfaz de un sólo documento). Pulsando Next sucesivamente hasta que pueda pulsar Finish.

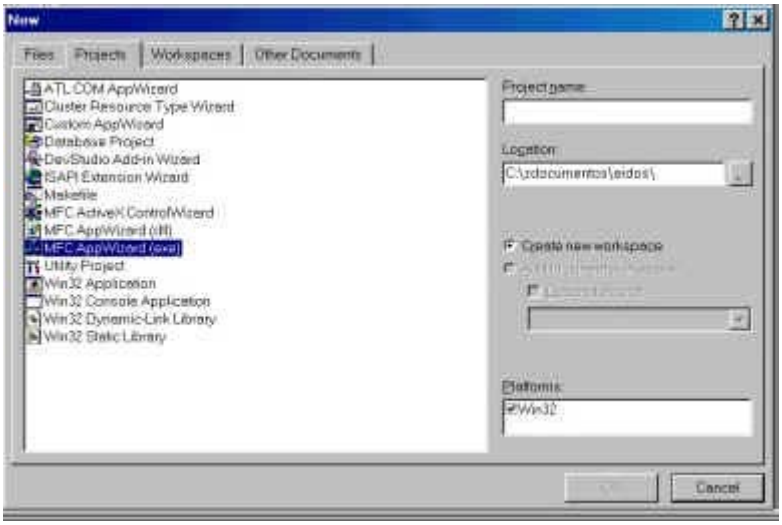


Figura 5

Con ello, ha creado su primer proyecto. Ahora tendrá en pantalla el interfaz que aparece en la Figura 6. El lector puede observar que ahora la barra de título muestra el nombre del proyecto "HolaMundo1", y que la zona de trabajo muestra en su parte izquierda distintas plantillas: ClassView (Vista de Clases), ResourceView (Vista de Recursos) y FileView (Vista de Archivos). Cada uno de ellos da acceso a distintas partes del programa.

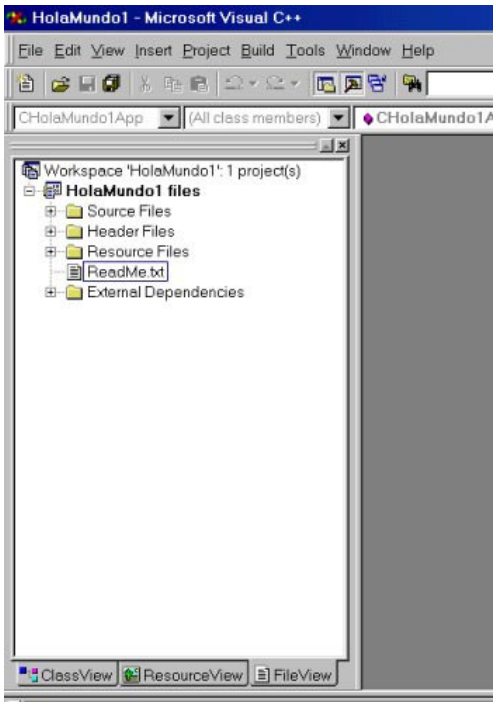


Figura 6

Para terminar, hay que modificar un único archivo de los creados por el MFC Wizard: seleccionamos la plantilla FileView, y dentro de Source Files, elegimos el archivo HolaMundo1View.cpp, que contiene los métodos de la clase HolaMundo1View.

El lector puede observar que la extensión por defecto para los archivos de C++ en Visual C++ es .CPP. En ese archivo modificamos el método OnDraw, añadiendo el Código fuente 214.

```
...
pDC->TextOut (25, 30, "Hola Mundo!");
...
```

Código fuente 214

El alumno debe observar que al introducir el texto pDC-> aparece la utilidad IntelliSense de Microsoft que ofrece para cada clase MFC los distintos métodos disponibles, con sus respectivas opciones, reduciendo el número de consultas. Una vez guardados los cambios con File + Save, el archivo ejecutable se construye con Build + Build HolaMundo1.exe (tecla rápida F7). Para ejecutar el programa, se puede hacer directamente desde el propio entorno de Visual C++ en la opción de menú Build + Execute HolaMundo1.exe (tecla rápida Ctrl+F5), obteniéndose nuestra primera aplicación de Windows.

Observe que la aplicación generada por defecto contiene todos los elementos típicos de una aplicación Windows (tiene su "*Look & Feel*"): Barra de Titulo, Menú, Barra de Herramientas, y zona de trabajo. Para iniciar un nuevo proyecto, ejecutar File + Close Workspace, que cierra el proyecto actual (previamente puede ser necesario guardar el resultado con File + Save Workspace).

Figura 7

Aplicaciones de consola

A lo largo de este curso hemos realizado multitud de ejercicios para estudiar las peculiaridades de C y C++. En los ejemplos y ejercicios, se han manipulado ejemplos que se ejecutan en un terminal (aplicaciones de consola). Visual C++ permite también la creación de aplicaciones de consola. Para crear una aplicación de consola, utilizamos de nuevo el AppWizard (File + New), seleccionando Win32 Console Application, y dándole el nombre HolaMundo2 al proyecto. Como dato anecdótico, en la segunda pantalla aparece la creación de un Hello World. Seleccionar esa opción. El código creado por defecto es el que muestra el Código fuente 215.

```
// HolaMundo2.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int main(int argc, char* argv[])
{
        printf("Hello World!\n");
        return 0;
}
```

Código fuente 215

Este método permite crear rápidamente aplicaciones de consola en C y C++.

El entorno de desarrollo integrado

A lo largo de los pequeños ejercicios anteriores hemos aprendido como utilizar algunos aspectos del entorno de desarrollo de Visual C++. En esta sección comentaremos otros aspectos que se pueden personalizar. En primer lugar, destacamos el uso de una estructura de árbol en la presentación de los proyectos (Workspaces). Así, tanto en la vista de archivos (FileView), clases (ClassView) o recursos (ResourceView), estos se agrupan por elementos comunes y jerarquías.

Otro elemento importante es que la creación de proyectos origina dos tipos de archivos objeto y ejecutable. Existe la versión de Depuración (Debug) activada por defecto, y la versión de Publicación (Release), que no contiene información para depuración. La selección de un tipo u otro se realiza en el menú Build + Set Active Configuration.

En el entorno de desarrollo de Visual C++ existen atajos de teclado (Keyboard shortcuts) que permiten acceder a los distintos elementos. Para ver la lista de estos atajos ir a Help + Keyboard Map. La personalización de estos elementos se puede realizar en el menú Tools (Herramientas) + Customize (Personalizar).

Ejercicios

11. Modificar el proyecto HolaMundo1 para incluir distintos mensajes nuevos por pantalla. Compilar y ejecutar el resultado.

12. Comprobar que el programa ejecutable HolaMundo1.exe, que se encuentra en el subdirectorio Debug del directorio del proyecto, es ejecutable directamente desde el Explorador de Windows.

13. Crear proyectos de consola para los ejemplos y ejercicios de los capítulos anteriores.

14. ¿Cómo obtener en Visual C++ el código del programa HolaMundo2?

Respuestas a los ejercicios

Al tratarse de un código especial, las respuestas se encuentran en:

HolaMundo1

HolaMundo2

13

Fundamentos de la programación en Windows

Introducción

El desarrollo de aplicaciones para entornos de ventanas en distintos sistemas operativos (desde las distintas versiones de UNIX con X-Window, hasta los Mac de Apple, y las primeras versiones de Windows) fue durante la década de los 80 y principios de los 90 casi un arte arcano. Bibliotecas escritas en C, gestión compleja de hilos de ejecución y distribución de elementos gráficos eran horas de desarrollo.

Sin embargo, desde principios de los 90, el desarrollo en los entornos Windows se fue simplificando gracias al desarrollo de lenguajes orientados a objetos como C++ y a dos bibliotecas desarrolladas casi en paralelo. Por un lado, Borland (ahora Inprise) desarrolló la **OWL** (Object Window Library), y Microsoft las **MFC** (Microsoft Foundation Classes). El hecho de que los sistemas operativos MS Windows (9x/NT/2000) incorporen por defecto éstas últimas ha hecho prevalecer en el mercado el conjunto de las MFC.

Este es un capítulo un poco especial en el que el alumno no va a encontrar ejemplos, ejercicios o aplicaciones. Sin embargo, va a ser de los más útiles porque le va a permitir una consulta rápida de las principales clases de MFC.

¿Qué son las MFC?

Las MFC son un conjunto jerárquico de clases escritas en C++ para el desarrollo de aplicaciones en el entorno MS Windows. Las MFC poseen un modelo de desarrollo denominado **doc/view**. El concepto fundamental de este modelo es la separación de los datos de la aplicación (documento, doc) del interfaz de usuario (view). Este modelo se encuentra organizado en una estructura jerárquica de clases. Por ejemplo, la clase CWnd es la clase de ventanas de más alto nivel, de la que derivan todas las demás, y pueden manipular textos, gráficos, eventos de ratón, etc.

¿Cuáles son las ventajas de las MFC?

Las ventajas de utilizar las MFC se pueden resumir en los siguientes puntos:

- Presentan la arquitectura documento/vista ya descrita
- Presentan la posibilidad de diseñar aplicaciones con interfaz de documento múltiple (Multiple Document Interface, MDI)
- Soportan impresión y vista preliminar de documentos
- Soportan el uso y creación de controles Active X
- Soportan bases de datos con ODBC y Microsoft Access
- Soportan la programación a través de Internet (TCP/IP)
- Soportan controles comunes de Win9x/NT/2000
- Soportan la programación multihilo (multithread)

Organización de las clases

Como es obvio suponer, la organización de las MFC obedece una jerarquía estricta. De otro modo, la programación con estas clases sería caótica. Todas las clases de las MFC derivan de la clase CObject.

A partir de esta clase, se derivan los tipos principales

- Clases de mantenimiento y gestión de archivos
- Gestión de ventanas: CWnd
- Gráficos: CDC
- Bases de Datos

De la Tabla 24 a la Tabla 30 se presentan las clases principales de MFC, clasificadas según su aplicación.

Clases de Mantenimiento de Archivos	
Nombre de clase	**Descripción**
Cfile	Proporciona servicios de acceso binario sin buffer a disco.
CrecentFileList	Proporciona control sobre toda la lista de archivos usados más recientemente.
CMemFile	Incluye acceso a archivos de mapa de memoria
ColeStreamFile	Represente un flujo de datos en un archivo compuesto de almacenamiento OLE.
CsocketFile	Utilizado para enviar y para recibir información a través de una conexión de red TCP/IP.
CstdioFile	Proporciona servicios de acceso con buffer a disco parecidos a un flujo de archivo ejecutable de C
CsharedFile	Soporte de archivos de memoria que son compartidos.
CmonikerFile	Representa un flujo de datos en un archivo compuesto de la misma forma en que un nombre de ruta identifica un archivo de disco normal.
CasyncMonikerFile	Permite que un control Active X cargue información desde un flujo de datos de un modo asíncrono.
CDataPathProperty	Permite que un control Active X cargue Información de propiedades de control de un modo asíncrono.
CCachedDataPathProperty	Permite que un control Active X cargue información de propiedades de control de un modo asíncrono y hacer caché de los datos en memoria RAM.
CgopherFile	Incluye la transferencia de datos desde un servidor Gopher.
ChttpFile	Incluye la transferencia de datos desde un servidor web.

Tabla 24

Clases de Ventana Marco	
Nombre de clase	**Descripción**
CMDIChildWnd	Ventana hija MDI (múltiple documentos)
CMDIFrameWnd	Ventana marco MDI (múltiples documentos)

COleIPFrameWnd	Por hacer
CsplitterWnd	Ventana dividida tipo Explorador.

Tabla 25

Clases de Ventana Vista	
Nombre de clase	**Descripción**
Cview	Incluye una vista básica dentro de una aplicación document/view.
CctrlView	Aplica la arquitectura document/view a los controles comunes de Windows. Sirve como clase base para los cuatro siguientes.
CeditView	Aplica la arquitectura document/view al control común de edición de Windows.
ClistView	Aplica la arquitectura document/view al control común de lista Windows.
CRichEditView	Aplica la arquitectura document/view al control común de edición de texto enriquecido correspondiente a Windows.
CTreeView	Aplica la arquitectura document/view al control común de árbol de Windows.
CScrollView	Proporciona una clase CView con capacidad de incluir barras de desplazamiento incorporadas.
CFormView	Clase base para vistas que contengan controles, como los cuadros de diálogo. La disposición del formato se basa en una fuente diálogo que se incluye en la aplicación al compilar.
CDaoRecordView	Un tipo de clase CView que muestra información del conjunto de registro de bases de datos jet en los controles.
CREcordView	Un tipo de clase CView que muestra información del conjunto de registros de bases de datos ODBC en los controles.

Tabla 26

Clases de Cuadros de Diálogo	
Nombre de clase	**Descripción**
CDialog	Clase base empleada para mostrar cuadros de diálogo.
CCommonDialog	Incluye la función base de los cuadros de diálogo común de Windows y sirve como base de las cinco clases siguientes.

CColorDialog	Ofrece el cuadro de diálogo común de Windows para selección de color.
CFileDialog	Incluye el cuadro de diálogo común de Windows para seleccionar un archivo.
CFindReplaceDialog	Incluye el cuadro de diálogo común de Windows para buscar y remplazar.
CFontDialog	Ofrece el cuadro de diálogo común de Windows para selección de fuente.
CPrintDialog	Incluye el cuadro de diálogo común de Windows de impresión. Proporciona una manera sencilla de incluir cuadros de diálogo "imprimir" y "configurar impresora".
CPropertyPage	Representa una sola página de una hoja de propiedades

Tabla 27

Clases de Controles	
Nombre de clase	**Descripción**
CButton	Botón normal.
CBitmapButton	Un botón con un gráfico de mapa de bits como etiqueta en vez de texto.
CComboBox	Un cuadro de edición con una lista desplegable asociada.
CEdit	Un cuadro de texto.
CHeaderCtrl	Una ventana colocada por encima de columnas de texto y que contiene los títulos de las columnas. Incluye también el control de cabecera común de Windows.
CHotKeyCtrl	Incluye el cuadro de diálogo común de Windows para teclas de método abreviado.
CListBox	Un control que contiene una lista de las opciones disponibles.
CCheckListBox	Un control que muestra una casilla que se puede marcar o quitar la marca.
CDragListBox	Incluye un cuadro de lista normal y permite al usuario reordenar la lista de elementos arrastrándolos a una nueva posición.
CListCtrl	Incluye la función de un control de vista de lista, y proporciona distintos modos para visualizar la lista. El explorador de Windows utiliza este tipo de vistas.
COleControl	Clase base para desarrollar controles Active X.

CProgressCtrl	Incluye el control común de Windows de barra de progreso y permite a las funciones miembro manipular la visualización.
CRichEditCtrl	Incluye el control común de Windows de edición enriquecida. Este control permite al usuario introducir, editar y formatear texto.
CScrollBar	Proporciona la funcionalidad de una barra de desplazamiento.
CSliderCtrl	Incluye el control común de Windows que permite a las funciones miembro manipular el tamaño, escala y frecuencia de las marcas de barras deslizantes.
CSpinButtonCtrl	Incluye el control común de Windows botón-spin, que se adosa a una ventana; al hacer click sobre el botón superior del control, hace que aumente el valor de la ventana adosada.
CStatic	Incluye el control estático que puedes mostrar una cadena de texto, un cuadro, un rectángulo, un icono, un mapa de bits o un cursor.
CStatusBarCtrl	Incluye el control común de Windows barra de estado; una barra de estado se usa para mostrar diversas clases de datos de la aplicación.
CTabCtrl	Incluye el control común de Windows de ficha que aparece en las ventanas con más de una página, como en las agendas.
CToolBarCtrl	Incluye el control común de Windows barra de herramientas que puede contener los botones y controles de edición, lista o cuadros combinados.
CToolTipCtrl	Una ventana de sugerencias de herramientas es la que aparece al pasar el puntero del ratón sobre una herramienta durante aproximadamente medio segundo.
CTreeCtrl	Proporciona el control común de Windows árbol. Un árbol muestra información en orden jerárquico, como los que usa el explorador de Windows o el antiguo Administrador de Archivos para mostrar información de directorios y archivos.

Tabla 28

Clases de Gráficos	
Nombre de Clase	**Descripción**
CDC	Incluye un contexto de dispositivo y proporciona funciones miembro para manipular el contexto.
CclientDC	Construye un contexto de dispositivos que se asocia con el área cliente de una ventana.
CmetaFileDC	Construye un contexto de dispositivo que se asocia con un meta-archivo de Windows.

CpaintDC	Construye un contexto de dispositivo que solamente se puede usar para responder a los mensajes WM_PAINT.
CBitmap	Incluye un mapa de bits de Windows y proporciona funciones miembro para manipular la imagen.
CBrush	Incluye una brocha GDI de Windows.
CFont	Incluye una fuente GDI de Windows.
CPalette	Proporciona una paleta de color GDI de Windows.
CPen	Incluye una pluma GDI de Windows.
CRgn	Incluye una región GDI de Windows.

Tabla 29

Clases para Bases de Datos	
Nombre de Clase (Tipo de base de datos)	**Descripción**
CDataBase (ODBC)	Indica una conexión con una base de datos ODBC.
CRecordset (ODBC)	Representa un juego de registro ODBC.
CLongBinary (ODBC)	Incluye la manipulación de objetos binarios grandes, como imágenes, en una base de datos ODBC.
CDaoDatabase (DAO)	Indica una conexión con una base de datos Microsoft Jet (Access).
CDaoQueryDef (DAO)	Representa una definición de vista de jet
CDaoRecordset (DAO)	Indica un juego de registro jet.
CDaoTableDef (DAO)	Representa una definición de tabla de Jet.
CDaoWorkspace (DAO)	Indica un área de trabajo de Jet.

Tabla 30

El programador debe ser consciente de que la utilización de las clases MFC no siempre es la panacea. Muchas de las aplicaciones actuales son el producto de la combinación de distintas herramientas: bases de datos, servidores de internet, entornos visuales, etc. Trate de optimizar los recursos y no realizar toda la aplicación bajo un único paraguas.

Como resumen, la Figura 8 muestra la jerarquía de clases de las MFC.

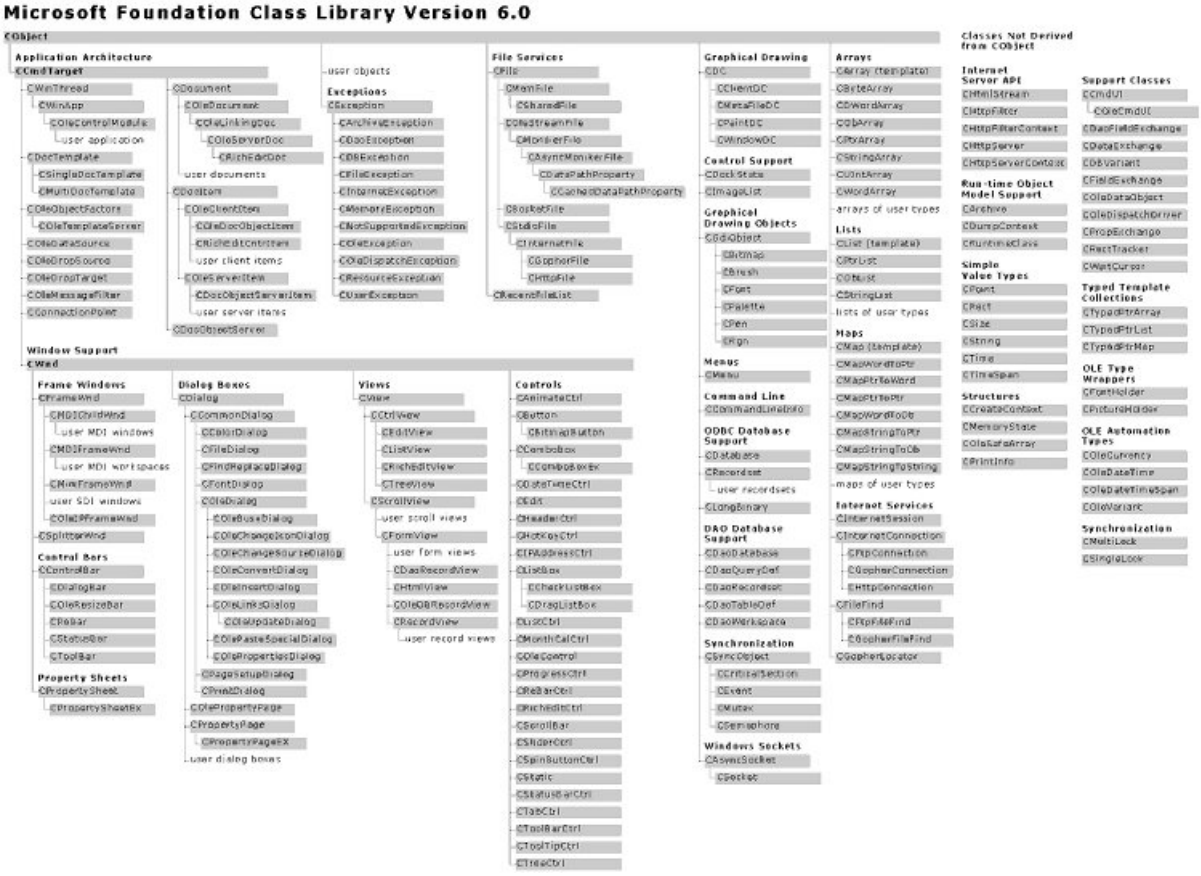

Microsoft Foundation Class Library Version 6.0
CObject
Application Architecture
CCmdTarget
CWinThread
CWinApp
COleControlModule
user application
CDocTemplate
CSingleDocTemplate
CMultiDocTemplate
COleObjectFactory
COleTemplateServer
COleDataSource
COleDropSource
COleDropTarget
COleMessageFilter
CConnectionPoint
CDocument
COleDocument
COleLinkingDoc
COleServerDoc
CRichEditDoc
user documents
CDocItem
COleClientItem
COleDocObjectItem
CRichEditCntrItem
user client items
COleServerItem
CDocObjectServerItem
user server items
CDaoObjectServer
user objects
Exceptions
CException
CArchiveException
CDaoException
CDBException
CFileException
CInternetException
CMemoryException
CNotSupportedException
COleException
COleDispatchException
CResourceException
CUserException
File Services
CFile
CMemFile
CSharedFile
COnDiskFile
CMonikerFile
CAsyncMonikerFile
CDataPathProperty
CCachedDataPathProperty
CRecentFileList
Graphical Drawing
CDC
CClientDC
CMetaFileDC
CPaintDC
CWindowDC
Control Support
CDockState
CImageList
Graphical Drawing Objects
CGdiObject
CBitmap
CBrush
CFont
CPalette
CPen
CRgn
Menus
CMenu
Command Line
CCommandLineInfo
ODBC Database Support
CDatabase
CRecordset
user recordsets
CLongBinary
DAO Database Support
CDaoDatabase
CDaoQueryDef
CDaoRecordset
CDaoTableDef
CDaoWorkspace
Synchronization
CSyncObject
CCriticalSection
CEvent
CMutex
CSemaphore
Windows Sockets
CAsyncSocket
CSocket
Arrays
CArray (template)
CByteArray
CDWordArray
CObArray
CPtrArray
CStringArray
CUintArray
CWordArray
arrays of user types
Lists
CList (template)
CPtrList
CObList
CStringList
lists of user types
Maps
CMap (template)
CMapWordToPtr
CMapPtrToWord
CMapPtrToPtr
CMapWordToOb
CMapStringToPtr
CMapStringToOb
CMapStringToString
maps of user types
Internet Services
CInternetSession
CInternetConnection
CFtpConnection
CGopherConnection
CHttpConnection
CFileFind
CFtpFileFind
CGopherFileFind
CGopherLocator
Classes Not Derived from CObject
Internet Server API
CHtmlStream
CHttpFilter
CHttpFilterContext
CHttpServer
CHttpServerContext
Run-time Object Model Support
CArchive
CDumpContext
CRuntimeClass
Simple Value Types
CPoint
CRect
CSize
CString
CTime
CTimeSpan
Structures
CCreateContext
CMemoryState
COleSafeArray
CPrintInfo
Support Classes
CCmdUI
COleCmdUI
CDaoFieldExchange
CDataExchange
CDBVariant
CFieldExchange
COleDataObject
COleDispatchDriver
CPropExchange
CRectTracker
CWaitCursor
Typed Template Collections
CTypedPtrArray
CTypedPtrList
CTypedPtrMap
OLE Type Wrappers
CFontHolder
CPictureHolder
OLE Automation Types
COleCurrency
COleDateTime
COleDateTimeSpan
COleVariant
Synchronization
CMultiLock
CSingleLock
Window Support
CWnd
Frame Windows
CFrameWnd
CMDIChildWnd
user MDI windows
CMDIFrameWnd
user MDI workspaces
CMiniFrameWnd
user SDI windows
COleIPFrameWnd
CSplitterWnd
Control Bars
CControlBar
CDialogBar
COleResizeBar
CRebar
CStatusBar
CToolBar
Property Sheets
CPropertySheet
CPropertySheetEx
Dialog Boxes
CDialog
CCommonDialog
CColorDialog
CFileDialog
CFindReplaceDialog
CFontDialog
COleDialog
COleBusyDialog
COlePasteSpecialDialog
COlePropertiesDialog
CPageSetupDialog
CPrintDialog
COlePropertyPage
CPropertyPage
CPropertyPageEx
user dialog boxes
Views
CView
CCtrlView
CEditView
CListView
CRichEditView
CTreeView
CScrollView
user scroll views
CFormView
user form views
CDaoRecordView
CHtmlView
COleDBRecordView
CRecordView
user record views
Controls
CAnimateCtrl
CButton
CBitmapButton
CComboBox
CComboBoxEx
CDateTimeCtrl
CEdit
CHeaderCtrl
CHotKeyCtrl
CIPAddressCtrl
CListBox
CCheckListBox
CDragListBox
CListCtrl
CMonthCalCtrl
COleControl
CProgressCtrl
CReBarCtrl
CRichEditCtrl
CScrollBar
CSliderCtrl
CSpinButtonCtrl
CStatic
CStatusBarCtrl
CTabCtrl
CToolBarCtrl
CToolTipCtrl
CTreeCtrl

Figura 8

14

Elementos de programación en Windows

Introducción

En este tema vamos a analizar algunos de los elementos principales de los programas en Windows. Así veremos como manipular menús, eventos de ratón y teclado, controles, diálogos, etc.

Menús

Los menús son de los elementos más importantes para el desarrollo de una aplicación. En ellos deben estar contenidos todas las posibilidades de la aplicación. Los menús creados con Visual C++ residen en un fichero de recursos que posee la extensión .rc. En estos archivos residen también otro tipo de recursos del sistema. Dentro de la misma categoría que los menús residen también las teclas de acceso rápido, definidas dentro de su clase. Las teclas de acceso rápido son combinaciones de teclas (como por ejemplo, Ctrl+S, para guardar los cambios en un archivo editado dentro de MS Visual C++) que permiten acceder a los elementos del menú.

Por ejemplo, abramos de nuevo la aplicación HolaMundo1 desarrollada en el capítulo 1 de esta parte dedicada a Visual C++. Para ello, abramos en primer lugar el proyecto correspondiente: File + Open Workspace ... . Una vez abierto, observamos que al hacer click en la Vista de Recursos (ResourceView), aparece una lista de recursos en forma de árbol. Seleccionando el elemento menu, aparece un elemento con el identificador IDR_MAINFRAME. Haciendo doble click sobre este elemento nos aparece en la zona de edición una imagen del menú.

Si seleccionamos cualquier elemento del menú, por ejemplo, Ayuda+Acerca de HolaMundo1..., y pulsamos la combinación de teclas Alt+Enter (Menú View + Properties), se obtiene una ventana de propiedades de menú, tal como se aprecia en la Figura 9. Conviene "fijar" el menú en la pantalla "clavando" la chincheta que aparece en la parte superior izquierda de la ventana Menu Item Properties.

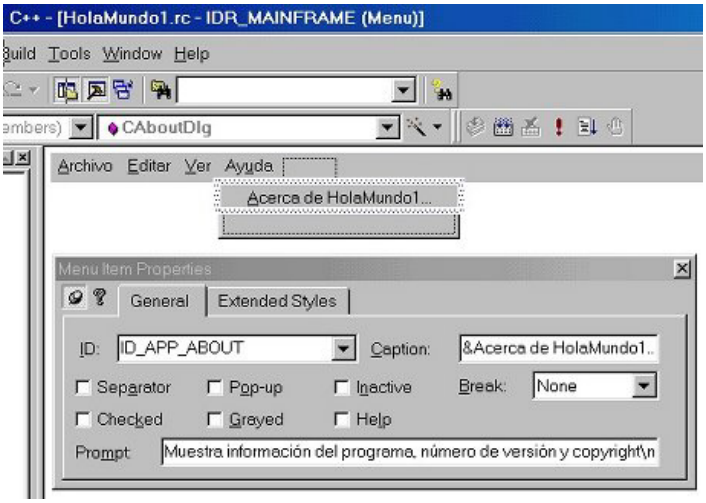


Figura 9

Cada elemento identificador de menú tiene las siguientes propiedades:

- ID: identificador que es una constante numérica. Las MFC tienen sus propios identificadores, que están recogidos en el archivo: afxres.h.

- Caption: texto del menú. La tecla de acceso rápido o mnemónico se distingue por el símbolo &. Así, para acceder al menú de salir, pulsamos Alt+A+S (Archivo+Salir).

Prompt: es el texto que muestra la barra de tareas al seleccionar el menú. Se puede añadir un "tooltip" (texto que aparece en pantalla, al lado del puntero del ratón cuando éste navega por el menú) de forma sencilla:

```
Texto barra de tareas\nTooltip
```

Las MFC cargan los menús a través de la clase CFrameWnd con el método CframeWnd:: LoadFrame(). Los manipuladores de mensajes para los menús se construyen a través del manipulador WM_COMMAND, asociado a cada elemento del menú. El código para cada elemento de menú se muestra en el Código fuente 216.

```
void CMainFrame::OnMenuEnable()
{
   // Código
}
```

Código fuente 216

El ratón y el teclado

La introducción de las MFC facilitó de forma notable la programación en Windows por la introducción de los eventos de teclado y ratón. Antes de éstas, la programación y gestión de la interacción con el usuario requería codificación compleja de hilos, comprobación del teclado, etc. a través de la API de Windows. Aunque el concepto de programación orientada a eventos parece compleja, en realidad simplifica la gestión de los mismos. De hecho, el desarrollador solamente debe comprobar la aparición de determinados mensajes, y gestionarlos.

El ratón

MFC tiene los mensajes de entrada aplicados al ratón que muestra la Tabla 31.

Mensaje	**Evento disparador (trigger)**
WM_LBUTTONDOWN	Botón primario pulsado
WM_LBUTTONUP	Botón primario libre
WM_RBUTTONDOWN	Botón secundario pulsado
WM_RBUTTONUP	Botón primario libre
WM_MBUTTONDOWN	Botón central pulsado
WM_MBUTTONUP	Botón central libre
WM_MOUSEMOVE	Ratón moviéndose

Tabla 31

Para analizar en detalle estos mensajes, vamos a utilizar el Wizard de MFC, y vamos a crear una nueva aplicación SDI que vamos a denominar MouseHandler. Dentro del entorno de vista de clases, hacemos click con el botón derecho del ratón sobre la clase MouseHandlerView, y elegimos la opción Add Windows Message Handler (añadir manipulador de mensajes de ventanas). Se obtiene la ventana que aparece en la Figura 10.

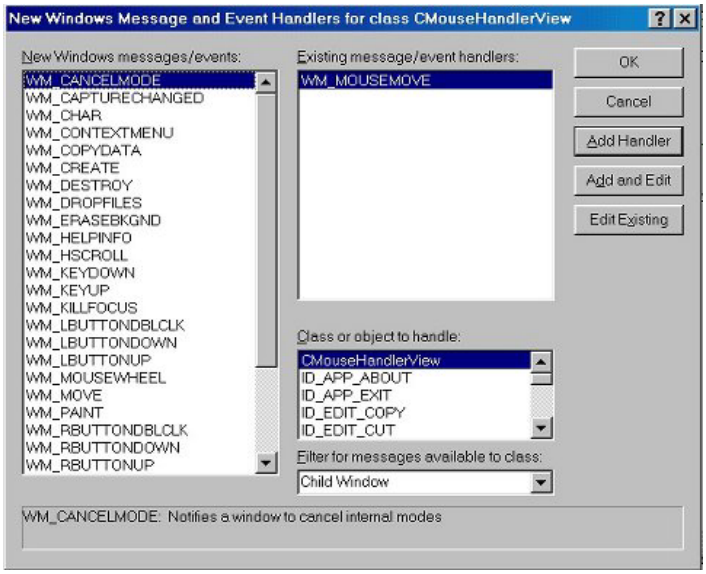


Figura 10

Si seleccionamos en la ventana de mensajes WM_MOUSEMOVE con doble click, aparecerá en la ventana de "Existing messages/event handlers". Con un doble click, podremos editar su código. Modifiquemos este código generado por defecto con el Código fuente 217.

```
void CMouseHandlerView::OnMouseMove(UINT nFlags, CPoint point)
{
  CView::OnMouseMove(nFlags, point);
  CClientDC ClientDC(this);
  CString strInfo;
  strInfo.Format("X:%04d, Y:%04d", point.x, point.y);
  ClientDC.TextOut(10, 10, strInfo, strInfo.GetLength());
}
```

Código fuente 217

Compile y enlace la aplicación. Obtendrá una aplicación que sigue los movimientos del ratón, imprimiendo sus coordenadas en pantalla. Observe que el puntero del ratón tiene unas coordenadas en la clase de MFC llamada CPoint. El alumno observará que las coordenadas son relativas a la ventana. Si deseamos obtener coordenadas absolutas respecto a la pantalla, debemos convertir las coordenadas usando la función ClientToScreen(). El Código fuente 218 muestra cómo.

```
void CMouseHandlerView::OnMouseMove(UINT nFlags, CPoint point)
{
  CView::OnMouseMove(nFlags, point);
  CClientDC ClientDC(this);
  CString strInfo;
  CPoint pt = point;
  ClientToScreen(&pt);
  strInfo.Format("X:%04d, Y:%04d", pt.x, pt.y);
  ClientDC.TextOut(10, 10, strInfo, strInfo.GetLength());
}
```

Código fuente 218

La manipulación de eventos adicionales es trivial. Supongamos que deseamos imprimir en pantalla cuando el botón primario del ratón se encuentra pulsado, y en que posición se pulsó. Siguiendo el mismo razonamiento anterior, basta con añadir los mensajes WM_LBUTTONDOWN y WM_LBUTTONUP. Además, creamos una función miembro ShowMouseStatus de la clase CMouseHandlerView, para imprimir la información. El código adicional es el Código fuente 219.

```
void CMouseHandlerView::OnLButtonDown(UINT nFlags, CPoint point)
{
  CView::OnLButtonDown(nFlags, point);
  ShowMouseStatus("Boton primario abajo", point, nFlags);
}

void CMouseHandlerView::OnLButtonUp(UINT nFlags, CPoint point)
{
  CView::OnLButtonUp(nFlags, point);
  ShowMouseStatus("Boton primario arriba", point, nFlags);
}

void CMouseHandlerView::ShowMouseStatus(
        const char *lpszText, CPoint point, int nFlag )
{
  CClientDC ClientDC(this);
  CString strInfo;

  strInfo.Format("X:%d Y:%d - %-25s",
                point.x, point.y, lpszText );
  ClientDC.TextOut(10, 30, strInfo, strInfo.GetLength());
}
```

Código fuente 219

Con ello, el usuario sabe cuando se pulsó o se liberó el botón primario del ratón por última vez. Además, el usuario puede controlar las pulsaciones del ratón en áreas no-cliente (bordes de la ventana, barra de título, barra de estado, barra de herramientas, etc.). Para ello existen los mensajes que aparecen en la Tabla 32.

Mensaje	**Evento disparador (trigger)**
WM_NCLBUTTONDOWN	Botón primario pulsado en área no-cliente
WM_NCLBUTTONUP	Botón primario libre en área no-cliente
WM_NCRBUTTONDOWN	Botón secundario pulsado en área no-cliente
WM_NCRBUTTONUP	Botón primario libre en área no-cliente
WM_NCMBUTTONDOWN	Botón central pulsado en área no-cliente
WM_NCMBUTTONUP	Botón central libre en área no-cliente
WM_NCMOUSEMOVE	Ratón moviéndose en área no-cliente
WM_NCHITTEST	Comprueba en qué área se encuentra el ratón (cliente o no-cliente)

Tabla 32

También es posible manipular el tipo de cursor que aparece en pantalla. Por ejemplo, si queremos mostrar el típico reloj de arena para realizar algún cálculo, es necesario llamar a la función BeginWaitCursor(), y EndWaitCursor() para terminar. El tratamiento general de los cursores se hace a través del mensaje WM_SETCURSOR. El siguiente ejemplo es muy sencillo, y sirve como ilustración. Creamos la aplicación SDI cursores que divide el área de la ventana cliente en 16 áreas rectangulares iguales. Según donde situemos el ratón en cada una de esas áreas se cargará un cursor HCURSOR. Recomendamos al lector el estudio detallado de la aplicación. Incluimos en el Código fuente 220 el código de cursoresView.cpp.

```
// cursoresView.cpp : implementation of the CCursoresView class
//

#include "stdafx.h"
#include "cursores.h"

#include "cursoresDoc.h"
#include "cursoresView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CCursoresView

IMPLEMENT_DYNCREATE(CCursoresView, CView)

BEGIN_MESSAGE_MAP(CCursoresView, CView)
        //{{AFX_MSG_MAP(CCursoresView)
        ON_WM_MOUSEMOVE()
        ON_WM_SETCURSOR()
        //}}AFX_MSG_MAP
        // Standard printing commands
        ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
        ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CCursoresView construction/destruction

CCursoresView::CCursoresView()
{
        // Construimos lista de cursores
        static char *szCursor[] = {
                IDC_ARROW, IDC_IBEAM, IDC_WAIT,
                IDC_CROSS, IDC_UPARROW, IDC_SIZENWSE,
                IDC_SIZENESW, IDC_SIZEWE, IDC_SIZENS,
                IDC_SIZEALL, IDC_NO, IDC_APPSTARTING,
                IDC_HELP, IDC_ARROW, IDC_ARROW, IDC_ARROW };

        // Cargamos la lista
        for(int i=0; i<16; i++)
                m_hCursor[i] = ::LoadCursor( NULL, szCursor[i] );
}

CCursoresView::~CCursoresView()
{
}
```

```
BOOL CCursoresView::PreCreateWindow(CREATESTRUCT& cs)
{
        // TODO: Modify the Window class or styles here by modifying
        //  the CREATESTRUCT cs

        return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CCursoresView drawing

void CCursoresView::OnDraw(CDC* pDC)
{
        CCursoresDoc* pDoc = GetDocument();
        ASSERT_VALID(pDoc);
        // TODO: add draw code for native data here
}

/////////////////////////////////////////////////////////////////////////////
// CCursoresView printing

BOOL CCursoresView::OnPreparePrinting(CPrintInfo* pInfo)
{
        // default preparation
        return DoPreparePrinting(pInfo);
}

void CCursoresView::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
        // TODO: add extra initialization before printing
}

void CCursoresView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
        // TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
// CCursoresView diagnostics

#ifdef _DEBUG
void CCursoresView::AssertValid() const
{
        CView::AssertValid();
}

void CCursoresView::Dump(CDumpContext& dc) const
{
        CView::Dump(dc);
}

CCursoresDoc* CCursoresView::GetDocument() // non-debug version is inline
{
        ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CCursoresDoc)));
        return (CCursoresDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CCursoresView message handlers

void CCursoresView::OnMouseMove(UINT nFlags, CPoint point)
{
        // Lista de cursores
        static CString strCursor[] = {
                "IDC_ARROW", "IDC_IBEAM", "IDC_WAIT",
                "IDC_CROSS", "IDC_UPARROW", "IDC_SIZENWSE",
```

```
                "IDC_SIZENESW", "IDC_SIZEWE", "IDC_SIZENS",
                "IDC_SIZEALL", "IDC_NO", "IDC_APPSTARTING",
                "IDC_HELP", "IDC_ARROW", "IDC_ARROW",
                "IDC_ARROW" };
        // Obtiene la región adecuada para cada punto
        int nCursor = GetCursorRegion(&point);

        // Obtiene DC
        CClientDC ClientDC(this);
        CString strInfo;
        strInfo = "Cursor:" + strCursor[nCursor] + "           ";
        ClientDC.TextOut(0, 0, strInfo, strInfo.GetLength());
        CView::OnMouseMove(nFlags, point);
}

BOOL CCursoresView::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
{
  // HTCLIENT test.
  if( nHitTest == HTCLIENT )
  {
    // Posicion del raton
                CPoint pt;
                GetCursorPos(&pt);
                // Convertir coordenadas a ventana cliente
                ScreenToClient(&pt);
                // Funcion que clasifica coordenadas
                int nCursor = GetCursorRegion(&pt);
                // Establece el cursor
                ::SetCursor(m_hCursor[nCursor]);
                return(TRUE);
  }
        return CView::OnSetCursor(pWnd, nHitTest, message);
}

int CCursoresView::GetCursorRegion(CPoint *lpPt)
{ // Divide el rectangulo area cliente en 4x4 areas
        // x vale 0-3, e y vale 0-3
        RECT Rect;
        GetClientRect(&Rect);
        int x = (lpPt->x * 4) / Rect.right;
        if(x > 3) x = 3;
        int y = (lpPt->y * 4) / Rect.bottom;
        if(y > 3) y = 3;
        return (y * 4 + x);
}
```

Código fuente 220

El lector debe recordar modificar el archivo de encabezado de la clase apropiadamente (cursoresView.h).

Otro elemento importante es la captura del ratón. Imaginemos que dentro del área cliente, el usuario pulsa el botón primario del ratón. El puntero del ratón abandona la ventana cliente y se dirige a otra aplicación, donde el usuario libera el botón. Cuando el puntero regresa a la ventana de nuestra aplicación, ésta creerá que está todavía pulsado cuando en realidad no lo está, dando lugar a comportamientos extraños. Para esto dispone la clase CWnd las funciones CWnd::SetCapture() y CWnd::ReleaseCapture() que deben ser invocadas convenientemente.

El teclado

La gestión del teclado la realizan las MFC a través de tres mensajes característicos:

- VM_KEYDOWN: tecla pulsada
- VM_KEYUP: tecla liberada
- VM_CHAR: código de la tecla

La gestión de los códigos de teclado de Windows con MFC es completamente diferente de la realizada con la API de Windows. Una función muy común era ::TranslateMessage() que "traducía" los códigos correspondientes. Con la MFC se manipula el teclado lógico de Windows que es independiente del hardware (teclado usado), idioma del teclado, e incluso el país.

De esta forma las MFC gestionan internamente estos códigos. Además, aparecen lo que se denominan teclas virtuales (VK_) que corresponden a caracteres no-imprimibles. Veamos la Tabla 33.

Tecla virtual	Equivalente	Tecla virtual	Equivalente	Tecla virtual	Equivalente
VK_MENU	Menú derecho ratón (teclados Win9x)	VK_Fx	F1 ... F12	VK_PRIOR	Re. Pág.
VK_APPS	Aplicaciones Windows (teclados Win9x)	VK_HOME	Inicio	VK_SNAPSHOT	Impr. Pant
VK_CONTROL	Control	VK_INSERT	Insert	VK_RIGHT	Flecha derecha
VK_DELETE	Supr	VK_LEFT	Flecha izquierda	VK_SHIFT	Mayúsculas
VK_DOWN	Flecha abajo	VK_PAUSE	Pausa	VK_UP	Flecha arriba
VK_END	Fin	VK_NEXT	Av. Pág.		

Tabla 33

Como ejercicio simple vamos a crear una aplicación SDI con nuestro ya conocido Wizard, que llamaremos VKey. A la clase VKeyView le añadimos los tres mensajes arriba indicados. A sus funciones manipuladoras les introducimos el Código fuente 221. Básicamente el funcionamiento será el siguiente: cuando se mantenga una tecla pulsada aparecerá el mensaje "Tecla pulsada"; cuando se libere aparecerá el mensaje "Tecla liberada", y cuando el carácter sea imprimible aparecerá en la línea inferior.

```
void CVKeyView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  CClientDC ClientDC(this);
  CString strInfo;
  strInfo.Format("%-30s", "Tecla pulsada");
  ClientDC.TextOut(10, 10, strInfo, strInfo.GetLength());
        CView::OnKeyDown(nChar, nRepCnt, nFlags);
```

```
}

void CVKeyView::OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  CClientDC ClientDC(this);
  CString strInfo;
  strInfo.Format("%-30s", "Tecla liberada");
  ClientDC.TextOut(10, 10, strInfo, strInfo.GetLength());
        CView::OnKeyUp(nChar, nRepCnt, nFlags);
}

void CVKeyView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  CClientDC ClientDC(this);
  CString strInfo;
  strInfo.Format("caracter: %2s", (LPCTSTR)&nChar);
  ClientDC.TextOut(10, 30, strInfo, strInfo.GetLength());
        CView::OnChar(nChar, nRepCnt, nFlags);
}
```

Código fuente 221

Es conveniente introducir muchas veces cursores de teclado. Las MFC introducen las funciones que aparecen en la Tabla 34, para gestionarlos.

Función	Descripción
CreateCaret()	Crea un cursor de teclado usando un mapa de bits que se le proporcione.
CreateGrayCaret()	Crea un cursor gris sólido con el tamaño que se haya especificado.
CreateSolidCaret()	Crea un cursor negro sólido con el tamaño que se haya especificado.
GetCaretPos()	Devuelve la localización del cursor. Sólo se podrá obtener la posición del cursor en Windows 95, Windows 98 y Windows NT si el cursor se encuentra dentro de una ventana que ha creado el mismo hilo que llamó a la función.
DestroyCaret()	Destruye un cursor.
HideCaret()	Oculta un cursor.
SetCaretPos()	Mueve un cursor a una posición de una ventana. Si se utilizan las barras de desplazamiento en una ventana que contiene un cursor, éste se desplaza con la pantalla.
ShowCaret()	Muestra un cursor.

Tabla 34

Estamos seguros de que los lectores están familiarizados con los puntos de inserción o cursores de teclado. Es recomendable usarlos cuando la ventana correspondiente tenga el foco. Para ello debemos procesar los mensajes WM_SETFOCUS, y WM_KILLFOCUS. Creando una aplicación de ejemplo SDI Focus con estos dos mensajes en CFocusView, debemos escribir el Código fuente 222.

```
void CMainFrame::OnSetFocus(CWnd* pOldWnd)
{
        CFrameWnd::OnSetFocus(pOldWnd);
  CreateSolidCaret(4, 40);
  SetCaretPos(d_ptCaretLocation);
  ShowCaret();
}
void CMainFrame::OnKillFocus(CWnd* pNewWnd)
{
        CFrameWnd::OnKillFocus(pNewWnd);
  d_ptCaretLocation = GetCaretPos();
  HideCaret();
  DestroyCaret();
}
void CMainFrame::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  CSize sizeTextBox;
  CPoint pt = GetCaretPos();
  {
    CClientDC ClientDC(this);
    ClientDC.TextOut(pt.x, pt.y, (LPCTSTR)&nChar, 1);
    sizeTextBox = ClientDC.GetTextExtent((LPCTSTR)&nChar, 1);
  }
  // Avanza el símbolo de inserción
  pt.x += sizeTextBox.cx;
  SetCaretPos(pt);
  ShowCaret();
  CFrameWnd::OnChar(nChar, nRepCnt, nFlags);
}
```

Código fuente 222

Controles

En todos los sistemas operativos Windows de 32 bits hay 20 controles típicos. De estos 20 controles, 6 existen desde la primera versión y se encuentran en el archivo USER.EXE. Los restantes 14 se denominan controles comunes y residen en la biblioteca COMCTL32.DLL. Los controles son elementos del interfaz gráfico de usuario para interaccionar con el usuario, y son ventanas "hija" unidas a una ventana "padre". Dentro de las MFC, los controles derivan de la clase base CWnd. Tenemos los controles que muestra la Tabla 35.

Control	**Clase MFC**	**Clase de ventanas (WNDCLASS)**
Botón	Cbutton	BUTTON
Cuadro de Lista	CListBox	LISTBOX
Cuadro de Lista Combinado	CComboBox	COMBOBOX
Control de Edición	Cedit	EDIT
Control Estático	Cstatic	STATIC
Barra de desplazamiento	CScrollBar	SCROLLBAR

Tabla 35

La creación de controles se puede realizar por dos métodos:

1. Incluir el control dentro de un recurso de cuadro de diálogo, e invocar posteriormente ClassWizard para que genere el código necesario; o

2. Llamar directamente a la función Create:

```
m_ListBox.Create(WS_CHILD | WS_VISIBLE | WS_VCSCROLL,
   ctlRect, this, IDC_CUSTS);
```

   Si deseamos crear un control 3D, debemos invocar la función:

```
TempEdit.CreateEx(WS_EX_CLIENTEDGE, "EDIT", NULL,
   WS_CHILD | WS_VISIBLE, TempRect, this, IDC_EDIT1);
```

CButton

CButton es la clase utilizada para generar distintos tipos de botones. Los flags de estilo más usados se muestran en la Tabla 36.

Estilo	Descripción
BS_PUSHBUTTON	Botón de comando
BS_DEFPUSHBUTTON	Botón de comando seleccionado por defecto
BS_RADIOBUTTON	Botón de opciones
BS_AUTORADIOBUTTON	Botón de opciones excluyentes entre sí
BS_GROUPBOX	Cuadro de grupo
BS_CHECKBOX	Casilla de verificación
BS_AUTOCHECKBOX	Casilla de verificación que se selecciona / des-selecciona al hacer click
BS_3STATE	Cuadro de verificación con 3 estados: seleccionado, no-seleccionado, indeterminado
BS_AUTO3STATE	Equivalente al anterior con cambio entre los 3 estados haciendo click

Tabla 36

Para crear una aplicación simple rápidamente, vamos a arrancar nuestro conocido Wizard para crear una aplicación de MFC llamada boton1. Esta vez, sin embargo, no elegimos la opción SDI, sino "Dialog Based" (basada en controles). Visual C++ creará la aplicación con dos botones: aceptar y cancelar. En la Figura 11, se observa como en la vista de recursos (Resource View) se pueden añadir

distintos tipos de controles de forma gráfica. En la figura hemos añadido un botón de radio y una casilla de verificación.

La captura de eventos se realiza del mismo modo que en el caso del ratón o del teclado. Por ejemplo, imaginemos que añadimos un botón Button1. Ahora, la función de manipulación para indicar qué hacer cuando se pulsa se denominará OnButton1. Por ejemplo, para contar el número de veces que pulsamos el botón e imprimirlo, el código será el que aparece en el Código fuente 223.

```
void CBoton1Dlg::OnButton1()
{
  static int i = 0;
  CClientDC ClientDC(this);
  CString strInfo;
  strInfo.Format("Boton1 pulsado: %2d veces", ++i);
  ClientDC.TextOut(10, 90, strInfo, strInfo.GetLength());
}
```

Código fuente 223

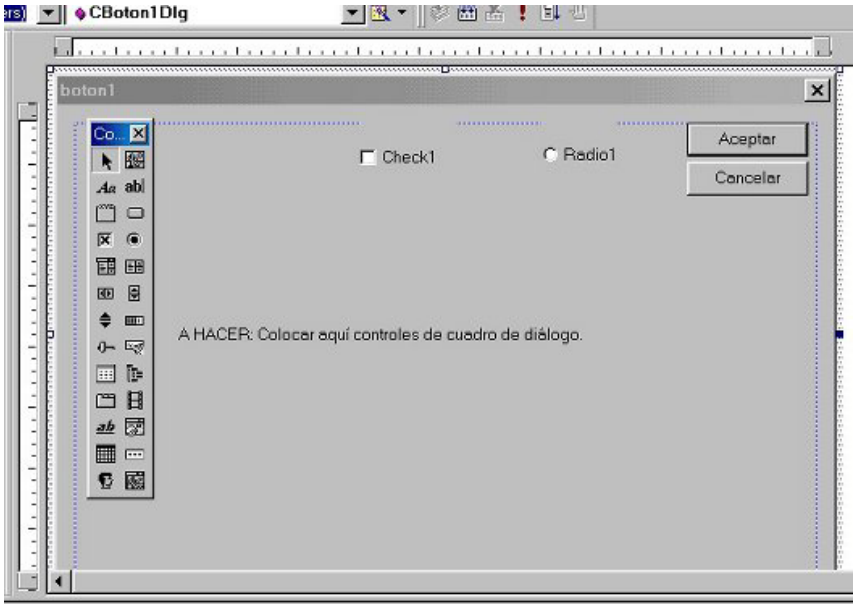


Figura 11

CListBox

CListBox, como ya hemos señalado antes es una aplicación de cuadro de lista. Es decir, la aplicación presenta una lista de cadenas de texto (opciones) de las que el usuario elige una. Sus flags de estilo se muestran en la Tabla 37.

Estilo	**Descripción**
LBS_Standard	Cuadro de lista por defecto con un borde y una barra de desplazamiento vertical, con los elementos ordenados alfabéticamente; envía mensajes de notificación a la ventana padre al cambiar una selección o hacer doble clic en un

	elemento.
LBS_SORT	Ordena los elementos alfabéticamente. Sin este estilo, los elementos se ordenan por orden de introducción.
LBS_NOSEL	Permite que se vean los elementos pero no permite seleccionarlos.
LBS_MULTIPLESEL	Permita seleccionar múltiples.
LBS_EXTENDEDSEL	Permite al usuario usar las teclas Control y Mayúsculas para hacer selecciones. Normalmente se encuentra emparejado a LBS_MULTIPLESEL.
LBS_NOTIFY	Notifica a la ventana padre cuando se cambia una selección o se hace doble clic en un elemento.
LBS_DISABLENOSCROLL	Muestra siempre una barra de desplazamiento vertical. Si el cuadro de lista no tiene suficiente elementos para desplazar, la barra de desplazamiento se desactiva. Por defecto los cuadro de lista ocultan las barras de desplazamiento cuando no son necesarias.
LBS_MULTICOLUMN	Permite mostrar varias columnas.
LBS_USETABSTOPS	Expande los tabuladores contenidos en las cadenas de caracteres. El uso de tabuladores se establece con la función CListBox::SetTabStops().
LBS_NOREDRAW	No actualiza el trazado automáticamente si se añade o elimina un elemento.
LBS_HASHSTRINGS	Recuerda los elementos que están añadidos. LBS_STANDARD incluye esta característica.
LBS_WANTKEYBOARDINPUT	Envía a la ventana padre un mensaje WM_KEYTOITEM o WM_CHARTOITEM cuando se pulsa una tecla y el cuadro de lista se encuentra activo para recibir entradas.
LBS_NOINTEGRALHEIGHT	La altura del cuadro de lista no tiene que ser múltiplo de la altura del elemento.

Tabla 37

Para añadir, insertar y borrar elementos a la lista se usan las funciones AddString(), InsertString() y DeleteString(), respectivamente. Para seleccionar y obtener el elemento seleccionado de la lista, debemos usar GetCurSel() y SetCurSel().

CEdit

Es un control que recupera texto del usuario. El texto introducido se puede Cortar, Copiar y Pegar. También existe la posibilidad de deshacer. Se puede limitar el número de caracteres introducidos,

aunque existe un máximo del sistema a 60k de texto. Para manipular textos mayores, usar el control CRichEditCtrl. Las funciones de añadir y recuperar texto son respectivamente SetWindowText() y GetWindowText(). Sus flags de estilo aparecen en la Tabla 38.

Estilo	**Descripción**
ES_LEFT	Hace que el texto contenido en el control se alinee a la izquierda
ES_RIGHT	Hace que el texto del control se alinee a la derecha
ES_CENTER	Hace que el texto del control se alinee al centro
ES_LOWERCASE	Muestra todo el texto en minúsculas
ES_UPPERCASE	Muestra todo el texto en mayúsculas
ES_WANTRETURN	Hace que Intro salte las líneas
ES_AUTOVSCROLL	Scroll vertical automático
ES_AUTOHSCROLL	Scroll horizontal automático
ES_OEMCONVERT	Convierte caracteres ANSI a OEM y viceversa
ES_NOHIDESEL	El control no oculta la selección cuando pierde el foco
WS_VSCROLL	Muestra barra de desplazamiento vertical
WS_HSCROLL	Muestra barra de desplazamiento horizontal

Tabla 38

CStatic

Los controles CStatic son de los más usados en las aplicaciones. Sirven para mostrar un texto o una imagen estática dentro de una aplicación. Los primeros son muy parecidos a los controles CEdit. Sus banderas de estilo principal son las que muestra la Tabla 39.

stilo	**Descripción**
SS_LEFT	Alinea el texto a la izquierda, y divide el texto cuando no cabe
SS_CENTER	Alineación al centro
SS_RIGHT	Alineación a la derecha
SS_LEFTNOWORDWRAP	Alineación a la izquierda sin mostrar el texto que no cabe
SS_CENTERNOWORDWRAP	Alineación al centro sin mostrar el texto que no cabe

SS_RIGHTNOWORDWRAP	Alineación a la derecha sin mostrar el texto que no cabe
SS_BITMAP	Muestra una imagen de mapa de bits
SS_ICON	Muestra un icono
SS_NOTIFY	Hace que el control envíe información a la ventana padre

Tabla 39

Otros controles

Existen multitud de controles que no podemos desarrollar aquí en toda su extensión. Referimos al lector a los ejercicios 4 y 5 para más detalles.

Diálogos modales y no modales

Los diálogos son elementos comunes en todas las aplicaciones y se usan para mostrar información o recabar información del usuario. Los diálogos desarrollan esta tarea mediante los controles mostrados en la sección anterior. El enfoque orientado a objetos de MFC se obtiene porque los diálogos derivan de la clase CDialog, y la clase derivada contiene los controles hijo. El fundamento de los diálogos es el intercambio de información de diálogo (DDX) y la validación de los datos de diálogo (DDV).

Como ya hemos mencionado, Visual C++ ofrece un editor de diálogos (Dialog Editor) que permite crearlos de forma cómoda. Vamos a ver algunos ejemplos sencillos. Para ello creemos con el AppWizard una aplicación basada en diálogos (agenda). Observe que al seleccionar todas las opciones por defecto aparece una aplicación con una plantilla de diálogo con dos botones básicos “Aceptar” y “Cancelar”. Desde el menú Insert, seleccionamos Insert Resource dónde aparecen las opciones de la Figura 12.

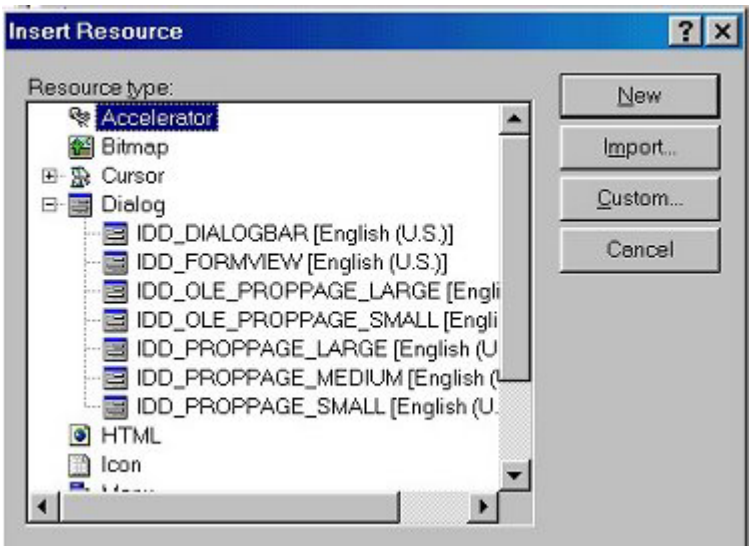


Figura 12

Al abrir el árbol de los posibles diálogos, vemos que existen varias posibilidades. En realidad, es un único diálogo con distintas opciones por defecto. Elija Dialog, y el botón New. Con la plantilla de diálogo seleccionada, pulsamos Alt+Intro, y nos aparecen sus propiedades. Vemos que el sistema asigna una identificación por defecto (ID) y un título (Caption) que podemos modificar si deseamos (agenda, por ejemplo). Dentro de las plantillas de diálogo que más hemos de modificar está la de estilos (Styles). En ella podremos asignar barra de título, botones de maximizar o minimizar, bordes, etc.

Asignemos ahora controles nuevos mediante la barra de controles (Controls Toolbar). El alumno debe observar que justo encima de la barra de estado aparecen una serie de botones para alineación de los controles, tamaño, etc. Estos botones se aplican a los botones seleccionados (mediante arrastre del ratón o mediante Control+botón primario del ratón). El orden de activación se establece pulsando Control+D. En este caso, aparecen los números 1 y 2 asociados al cuadro de diálogo. Haciendo click con el botón primario del ratón se cambia la secuencia de activación (orden de tabulación).

La clase CDialog

La clase de MFC que controla los diálogos se denomina CDialog. De esta clase se derivan tanto diálogos modales como no-modales. Cuando el diálogo es modal, Windows no permite al usuario interactuar con la ventana padre. En el caso no-modal, sí está permitido. La creación de un diálogo modal se realiza mediante:

```
CDialog(LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL);
CDialog(UINT nIDTemplate, CWnd* pParentWnd = NULL);
```

El diálogo se convierte en modal mediante CDialog::DoModal(). Para crear un diálogo no-modal usamos el constructor Dialog() sin argumentos, y luego invocamos a:

```
BOOL Create(LPCTSTR lpszTemplateName, CWnd* pParentWnd = NULL);
BOOL Create(UINT nIDTemplate, CWnd* pParentWnd = NULL);
```

La creación de clases derivadas de CDialog se realiza mediante View+ClassWizard, y luego seleccionamos New Class. Se introduce el nombre que se desea dar a la clase, tal como aparece en la Figura 13.

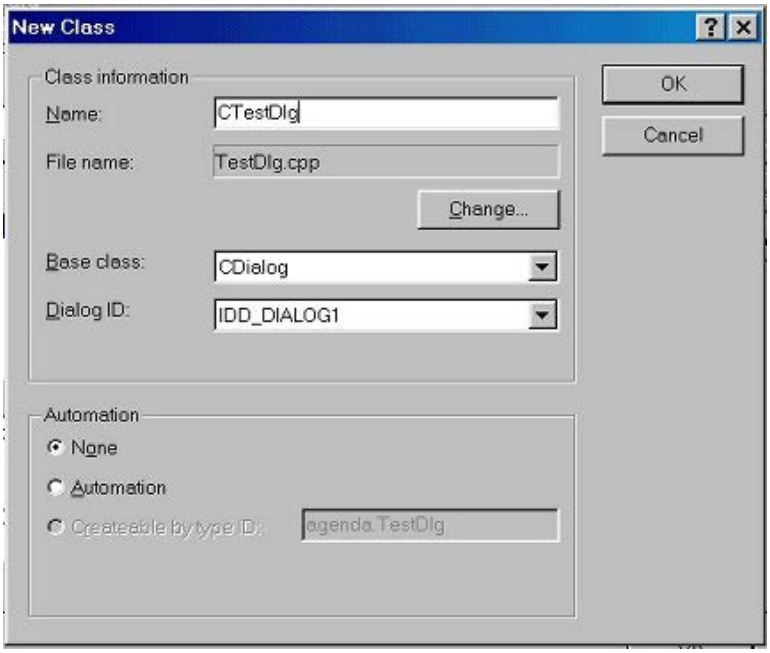


Figura 13

Intercambio de información y validación

El aspecto final de los diálogos es el intercambio de información con la propia aplicación. Como ya hemos mencionado, DDX permite crear variables miembro en su clase derivada de CDialog para asociarlas a cada control de la plantilla de diálogo. Los tipos válidos son los siguientes: CString, int, UINT, long, DWORD, float, double, BYTE, short, BOOL, COleDateTime y COleCurrency. Conviene recordar que para un control de casilla de verificación el único valor posible es BOOL.

Las variables miembro se añaden a través del cuadro de diálogo Add Member Variable de ClassWizard. Las variables miembro suelen comenzar por m_, y se asocian a un control determinado. La función para recuperar información de los diálogos invoca a la función CWnd::DoDataExchange(). Veamos el Código fuente 224.

```
void CTestDlg::DoDataExchange(CDataExchange *pDX)
{
   CDialog::DoDataExchange(pDX);
   DDX_Control(pDX, IDC_EDIT1, m_edtMyEdit);
   DDX_Text(pDX, IDC_EDIT1, m_strMyEdit);
}
```

Código fuente 224

La función DDX_Control() asocia objetos derivados de CWnd con un control, de forma que éste se puede manipular mediante llamadas a las funciones miembro de objetos derivados de CWnd. El primer argumento es un puntero a un objeto DoDataExchange, que hemos pasado desde el sistema hasta la función DoDataExchange(). El segundo elemento es la ID de un control del diálogo, mientras que el tercer argumento se usa para introducir un argumento a un objeto derivado de CWnd. En este caso, m_edtMyEdit es una referencia a un objeto CEdit. La función DDX_Text() se usa para transferir el valor de un control a una variable miembro, y viceversa. En este caso, se asocia el control con una variable CString.

A la función DoDataExchange() se le llama como resultado de una llamada a CWnd::UpdateData, y toma un único argumento que indica si el control o la variable miembro se ha actualizado:

```
BOOL UpdateData(BOOL bSaveAndValidate);
```

Si bSaveAndValidate es TRUE, las variables miembro se actualizan con los valores de los controles. En caso de necesitar que el control tome otro valor, actualizaría la variable miembro correspondiente y llamaría a UpdateData() con el valor FALSE. Es importante observar que el sistema llama dos veces a la función UpdateData() de forma automática. La primera vez es a través de la función virtual CDialog::OnInitDialog(), que se hace al iniciarse el diálogo, y la segunda al mostrarse en pantalla (que es llamada con FALSE):

```
BOOL CTestDlg::OnInitDialog()
{
   m_strMyEdit = _T("Mi valor inicial");
   CDialog::OnInitDialog();
   return TRUE;
}
```

Código fuente 225

Si la aplicación necesita actualizar las variables miembro después de la llamada a OnInitDialog(), debe llamar a UpdateData() por sí mismo.

```
BOOL CTestDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   m_strMyEdit = _T("Mi valor inicial");
   UpdateData();
   return TRUE;
}
```

Código fuente 226

Recapitulación

En este tema hemos descrito todos los elementos en la programación básica de interfaces gráficos de usuario en C++ bajo Windows de un modo introductorio. También se ha presentado el intercambio de datos entre la aplicación y el usuario. Recomendamos al alumno el estudio detallado de estos elementos de forma que se familiarice con las peculiaridades de la programación con MFC.

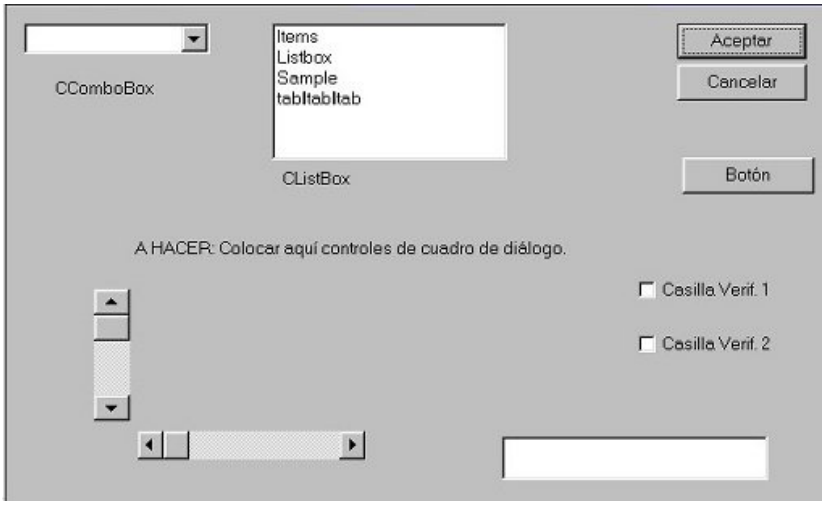
Ejercicios

1. En la aplicación HolaMundo1 cuál es el identificador del menú Editar+Cortar. ¿Cuál es su ToolTip?

2. Crear una copia del workspace HolaMundo1 en otro directorio HolaMundo3. Modificar el menú de ayuda para incluir una notificación de Copyright.

3. Modificar el ejemplo MouseHandler para imprimir coordenadas absolutas y relativas.

4. Crear una aplicación basada en diálogo que muestre los controles de esta figura.



Realizar paso a paso el tutorial para la aplicación Scribble incluido en la ayuda de Microsoft Visual C++.

Respuestas a los ejercicios

Ejercicio 1: ID_EDIT_CUT. Cortar.

Al tratarse de un código especial, las respuestas se encuentran en: VC-cap3

15

Introducción a ActiveX con Visual C++

Introducción

En los capítulos anteriores, el lector se ha ido familiarizando con diversos elementos de la programación en C y C++. También se le han presentado los fundamentos básicos de la programación en Windows con la utilización de Microsoft Visual C++ y las Microsoft Foundation Classes. No podíamos cerrar este curso introductorio sin mencionar uno de los elementos que están cobrando cada vez mayor auge en la programación de objetos distribuidos en la red como son los "Objetos ActiveX" u "Objetos COM" (COM son las siglas de Component Object Model, Modelo de Objetos con Componentes).

En el espacio disponible trataremos de describir estos elementos y orientar al alumno dentro del mar de posibilidades que estos objetos ofrecen. Dentro de los elementos de ayuda de Microsoft Visual C++ existe gran documentación al respecto.

Orígenes de ActiveX

Según el mundo de las aplicaciones se va haciendo más y más complejo, resulta difícil no encontrarse con la necesidad de incluir dentro de nuestra aplicación datos de distintas fuentes. Así, una presentación multimedia puede encontrarse con la necesidad de incluir texto con formato, audio, vídeo y acceso a bases de datos en tiempo real. Estas tecnologías incluyentes se basan en el OLE (Object Linking and Embedding, enlazado e inclusión de objetos). Como no podía ser menos, las MFC pueden trabajar con este tipo de documentos con la clase CCmdTarget.

El aspecto básico, tanto en objetos enlazados (o vinculados) como en objetos incluidos es la flexibilidad en la edición de los mismos. Con el paso de los años, lo que en principio se denominó OLE, a pasado a denominarse ActiveX. Sin embargo, no está claro si la necesidad de componentes pequeños y reutilizables (objetos COM) es debida a la existencia de Internet, o era una necesidad empresarial. De hecho, la mayoría de las aplicaciones desarrolladas a partir de la definición de la tecnología ActiveX admiten la posibilidad de incluir este tipo de componentes, con lo que la capacidad de personalización es infinita.

Clasificación de la tecnología ActiveX

La tecnología ActiveX, como ampliadora o continuadora de OLE, presenta seis categorías

1. Servidores de automatización

   Son componentes que pueden controlar otras aplicaciones. Tienen una o más interfaces controladas por IDispatch. Puede tener una interfaz de usuario, y se puede ejecutar como un proceso, en local, o en remoto.

2. Controladores de automatización

   Son aquellas aplicaciones que controlan a los servidores de automatización.

3. Controles ActiveX

   Son controles de 32 bits equivalentes a los controles OLE y OCX. Típicamente constan de un interfaz de usuario, un interfaz IDispatch que define sus métodos y propiedades, y un interfaz IConnectionPoint para controlar los sucesos que disparan el control.

4. Objetos COM

   Son elementos parecidos a los servidores y controladores de automatización. Tienen una o más interfaces COM, y casi nunca tienen interfaz de usuario.

5. Documentos ActiveX

   Son elementos denominados anteriormente DocObjects, y representan cualquier cosa, desde una hoja de cálculo hasta una llamada a otra aplicación. Estos poseen interfaz de usuario. El ejemplo más típico es el cambio en el menú de la aplicación cuando cambia el tipo de documento.

6. Contenedores ActiveX

   Son aplicaciones que pueden albergar servidores de automatización, controles y documentos.

Qué puede hacer ActiveX

Para desarrollar aplicaciones ActiveX conviene estudiar los siguientes puntos:

- Requisitos de la aplicación: determinará si es necesario usar una o más tecnologías ActiveX.
- Elección del tipo de componente y de su arquitectura.

- Selección de la herramienta correcta.
- Arquitectura básica de un componente ActiveX
- Herramienta de soporte ActiveX (en su caso, Microsoft Visual C++).

ActiveX Template Library

La introducción de la biblioteca de clases ActiveX Template Library (ATL) supuso en 1996 un gran espaldarazo para el uso de tecnologías ActiveX. Hasta la fecha, la creación de controles ActiveX estaba sujeto al uso de MFC, con lo que la "huella" (o tamaño) mínima de los elementos desarrollados rondaba los 100kb. Esto que puede no parecer mucho, suponía la imposibilidad de utilizar estos elementos a través de la red por su velocidad de descarga.

Las ATL son elementos tan flexibles como las MFC y permiten la creación de componentes de pequeño tamaño. De hecho, se ha convertido en la alternativa preferida al uso de BaseControl (BaseCtl) Framework y ActiveX SDK.

Herramientas necesarias para construir componentes ActiveX

Los siguientes elementos son necesarios para construir componentes ActiveX:

- Compilador MIDL: incorporado en Visual C++ desde la versión 5.
- Mktyplib: compilador de librerías.
- GUIGEN: generador de identificadores únicos para interfaces y clases.
- RegEdit: editor de registros de Win9x/NT. Es una herramienta muy potente del sistema operativo. Recomendamos su uso con mucha prudencia porque puede originar un mal funcionamiento del sistema (aplicación regedt32.exe).
- Servidor de registro (aplicación regsvr32.exe).
- Ole2View: lista todos los componentes ActiveX del sistema (incluida en las Tools de Visual C++).

Usando ATL para crear controles ActiveX

Como hemos mencionado anteriormente, el uso de ATL para crear controles ActiveX es una de las técnicas más útiles. ATL, al igual que las librerías STL implantan el uso de plantillas. ATL es una biblioteca, aparentemente menos flexible que las MFC, sin embargo, permite generar código de pequeño tamaño para controles ActiveX.

Vamos a crear desde el inicio un control ATL que dibuje un péndulo doble en una ventana (DoublePendulum). Se añadirán también funciones de actualización. Vamos a crearlo con el AppWizard, seleccionando el asistente de ATL COM AppWizard. En la siguiente pantalla, aparecerán diversas opciones de como cargar el servidor. Elegiremos la opción por defecto (DLL). Una vez terminado el asistente, y revisadas las opciones por defecto en el cuadro de diálogo correspondiente,

agregamos un nuevo objeto COM al proyecto: Insert+New ATL Object, obteniendo el correspondiente Wizard, que nos conduce a la Figura 14.

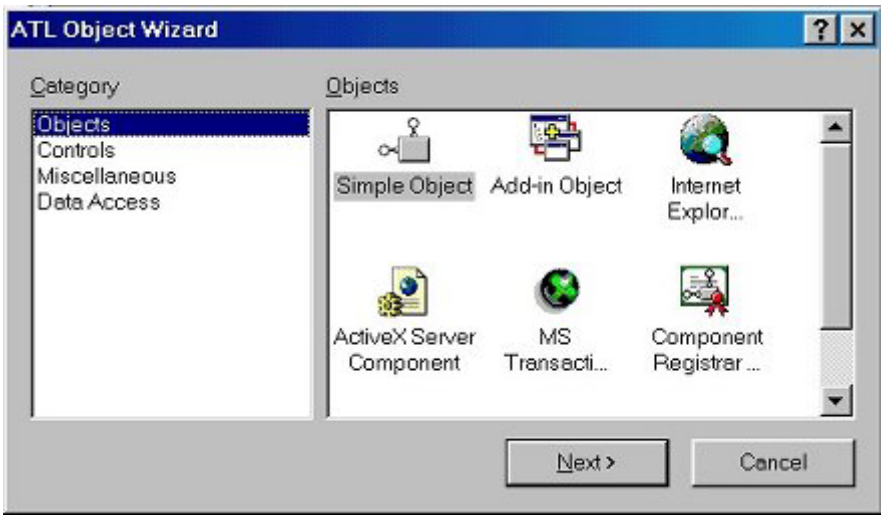


Figura 14

Observamos que existen cuatro componentes COM: Objects (objetos), Controls (controles), Miscellaneous (varios) y Data Access (acceso a bases de datos). Seleccionaremos Controls, y el objeto Full Control (control completo). Ahora tendrá que escribir un nombre para el control, y elegimos DoublePend. No modifique ninguna opción por defecto, y tendrá su nuevo control.

Ahora debemos modificar el control. Es evidente que la función OnDraw() proporcionada por el Wizard no basta para dibujar un péndulo. Sin entrar en el dibujo del péndulo que no corresponde a este curso, debemos modificar el archivo DoublePend.h para incluir las funciones de dibujo. El código a incluir es el Código fuente 227.

```
// DoublePend.h : Declaration of the CDoublePend

#ifndef __DOUBLEPEND_H_
#define __DOUBLEPEND_H_

#include "resource.h"       // main symbols
#include <atlctl.h>

/////////////////////////////////////////////////////////////////////////////
// CDoublePend
class ATL_NO_VTABLE CDoublePend :
        public CComObjectRootEx<CComSingleThreadModel>,
        public IDispatchImpl<IDoublePend, &IID_IDoublePend,
&LIBID_DOUBLEPENDULUMLib>,
        public CComControl<CDoublePend>,
        public IPersistStreamInitImpl<CDoublePend>,
        public IOleControlImpl<CDoublePend>,
        public IOleObjectImpl<CDoublePend>,
        public IOleInPlaceActiveObjectImpl<CDoublePend>,
        public IViewObjectExImpl<CDoublePend>,
        public IOleInPlaceObjectWindowlessImpl<CDoublePend>,
        public IPersistStorageImpl<CDoublePend>,
        public ISpecifyPropertyPagesImpl<CDoublePend>,
        public IQuickActivateImpl<CDoublePend>,
        public IDataObjectImpl<CDoublePend>,
        public IProvideClassInfo2Impl<&CLSID_DoublePend, NULL,
```

```
&LIBID_DOUBLEPENDULUMLib>,
        public CComCoClass<CDoublePend, &CLSID_DoublePend>
{
public:
        double m_dLength;
        double x[4], h;

        CDoublePend()
        {
                m_dLength = 100.0;
                x[0] = .7854;
                x[1] = .7854;
                x[2] = x[3] = 0.0;

                h = .1;
        }

        void Function( double *, double * );
        void FillOval( HDC, int, int, int, int, COLORREF );
        void DrawPend( HDC, RECT & );

DECLARE_REGISTRY_RESOURCEID(IDR_DOUBLEPEND)

BEGIN_COM_MAP(CDoublePend)
        COM_INTERFACE_ENTRY(IDoublePend)
        COM_INTERFACE_ENTRY(IDispatch)
        COM_INTERFACE_ENTRY(IViewObjectEx)
        COM_INTERFACE_ENTRY(IViewObject2)
        COM_INTERFACE_ENTRY(IViewObject)
        COM_INTERFACE_ENTRY(IOleInPlaceObjectWindowless)
        COM_INTERFACE_ENTRY(IOleInPlaceObject)
        COM_INTERFACE_ENTRY2(IOleWindow, IOleInPlaceObjectWindowless)
        COM_INTERFACE_ENTRY(IOleInPlaceActiveObject)
        COM_INTERFACE_ENTRY(IOleControl)
        COM_INTERFACE_ENTRY(IOleObject)
        COM_INTERFACE_ENTRY(IPersistStreamInit)
        COM_INTERFACE_ENTRY2(IPersist, IPersistStreamInit)
        COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
        COM_INTERFACE_ENTRY(IQuickActivate)
        COM_INTERFACE_ENTRY(IPersistStorage)
        COM_INTERFACE_ENTRY(IDataObject)
        COM_INTERFACE_ENTRY(IProvideClassInfo)
        COM_INTERFACE_ENTRY(IProvideClassInfo2)
END_COM_MAP()

BEGIN_PROP_MAP(CDoublePend)
        PROP_DATA_ENTRY("_cx", m_sizeExtent.cx, VT_UI4)
        PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
        // Example entries
        // PROP_ENTRY("Property Description", dispid, clsid)
        // PROP_PAGE(CLSID_StockColorPage)
END_PROP_MAP()

BEGIN_MSG_MAP(CDoublePend)
        CHAIN_MSG_MAP(CComControl<CDoublePend>)
        DEFAULT_REFLECTION_HANDLER()
END_MSG_MAP()
// Handler prototypes:
//  LRESULT MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, BOOL&
bHandled);
//  LRESULT CommandHandler(WORD wNotifyCode, WORD wID, HWND hWndCtl, BOOL&
bHandled);
//  LRESULT NotifyHandler(int idCtrl, LPNMHDR pnmh, BOOL& bHandled);

// IViewObjectEx
        DECLARE_VIEW_STATUS(VIEWSTATUS_SOLIDBKGND | VIEWSTATUS_OPAQUE)
```

```
// IDoublePend
public:
        STDMETHOD(Update)();
        STDMETHOD(SetValue)(double dValue);

        HRESULT OnDraw(ATL_DRAWINFO& di)
        {
                RECT& rc = *(RECT*)di.prcBounds;
                DrawPend( di.hdcDraw, rc );
                return S_OK;
        }
};

#endif //__DOUBLEPEND_H_
```

Código fuente 227

Por supuesto, no hemos terminado. Ahora es necesario incluir métodos en el interfaz. En particular, necesitamos establecer la velocidad de oscilación del péndulo con una función SetValue(). Para ello, vamos a la vista de clases, elegimos la clase IDoublePend, hacemos click con el botón derecho, y seleccionamos Add Method. La información se añade como muestra la Figura 15.

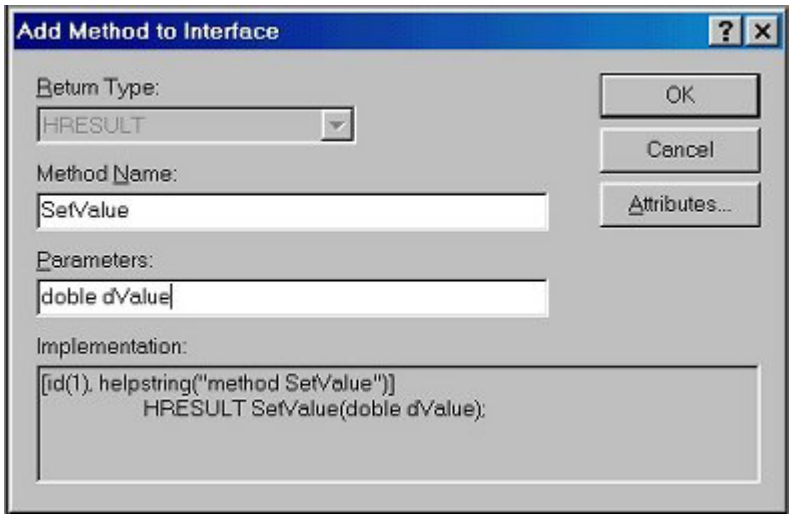


Figura 15

De igual modo, debemos añadir las funciones de dibujo, y el código de la función de actualizar el péndulo. El código a incluir en DoublePend.cpp es el que indica el Código fuente 228.

```
// DoublePend.cpp : Implementation of CDoublePend

#include "stdafx.h"
#include "DoublePendulum.h"
#include "DoublePend.h"

#include <math.h>

/////////////////////////////////////////////////////////////////////////////
// CDoublePend

void CDoublePend::Function( double *xx, double *f )
{
```

```
        double c, c0, c1, c10, x22, x33;
        // Gravity divided by the length.
        c = 9.8 / m_dLength;
        // Calculate motion vectors.
        c0 = cos( xx[0] ) * c;
        c1 = cos( xx[1] ) * c;
        c10 = cos( xx[1] - xx[0] );
        c = sin( xx[1] - xx[0] );
        // Calculate the positions.
        x22 = xx[2] * xx[2] * c;
        x33 = xx[3] * xx[3] * c;
        c = 9.0 / ( 16.0 - 9.0 * c1 * c10 );

        // Store the values.
        f[0] = xx[2];
        f[1] = xx[3];
        f[2] = ( 2.0 * x33 / 3.0 + c10 * x22 +
                c10 * c1 - 2.0 * c0 ) * c;
        f[3] = ( -8.0 * x22 / 3.0 - c10 * x33 +
                3.0 * c10 * c0 - 8.0 * c1 / 3.0 ) * c;
}

void CDoublePend::FillOval( HDC hDC, int nX, int nY,
        int nXR, int nYR, COLORREF cColor )
{
        HBRUSH hBrush, hOldBrush;
        HPEN hPen, hOldPen;
        // Create the brush, select it into the dc,
        // and remember the old brush.
        hBrush = CreateSolidBrush( cColor );
        hOldBrush = (HBRUSH) SelectObject( hDC, hBrush );
        // Create the pen, select it into the dc.
        // and remember the old pen.
        hPen = CreatePen( PS_SOLID, 1, RGB( 0, 0, 0 ) );
        hOldPen = (HPEN) SelectObject( hDC, hPen );

        // Draw the ellipse.
        Ellipse( hDC, nX - ( nXR / 2 ), nY - ( nYR / 2 ),
                nX + ( nXR / 2 ), nY + ( nYR / 2 ) );

        // Select the old brush and pen back
        // into the dc.
        SelectObject( hDC, hOldBrush );
        SelectObject( hDC, hOldPen );

        // Delete the GDI objects.
        DeleteObject( hPen );
        DeleteObject( hBrush );
}

void CDoublePend::DrawPend( HDC hDC, RECT& rc )
{
        // We're going to double buffer in
        // order to reduce the redraw flicker.
        HBITMAP hOldBitmap, hBitmap;
        // Create a dc for the off-screen buffer.
        HDC hWorkDC = CreateCompatibleDC( hDC );
        // Create the off-screen buffer.
        hBitmap = CreateCompatibleBitmap( hWorkDC,
                rc.right - rc.left, rc.bottom - rc.top );
        // Select the bitmap into the newly-created
        // dc and remember the bitmap that was
        // previously selected.
        hOldBitmap =
                (HBITMAP) SelectObject( hWorkDC, hBitmap );

        // Draw a rectangle to clear the buffer.
```

```
        Rectangle( hWorkDC, rc.left, rc.top, rc.right,
                rc.bottom );

        // Calculate the width, height, and midpoint.
        int nWidth = rc.right - rc.left;
        int nHeight = rc.bottom - rc.top;
        int nMidX = rc.left + ( nWidth / 2 );
        int nMidY = rc.top + ( nHeight / 2 );

        // Calculate the endpoints of the pendulum.
        int nCx1 = nMidX + (int) ( m_dLength * cos( x[0] ) );
        int nCy1 = nMidY + (int) ( m_dLength * sin( x[0] ) );
        int nCx2 = nCx1 + (int) ( m_dLength * cos( x[1] ) );
        int nCy2 = nCy1 + (int) ( m_dLength * sin( x[1] ) );

        // Draw the three pendulum ellipses.
        FillOval( hWorkDC, nMidX,
                nMidY, 7, 7, RGB( 0, 255, 0 ) );
        FillOval( hWorkDC, nCx1,
                nCy1, 11, 11, RGB( 0, 255, 0 ) );
        FillOval( hWorkDC, nCx2,
                nCy2, 11, 11, RGB( 0, 255, 0 ) );

        // Select a black stock pen into the dc.
        SelectObject( hWorkDC, GetStockObject( BLACK_PEN ) );
        // Move to the midpoint.
        MoveToEx( hWorkDC, nMidX, nMidY, NULL );
        // Draw the first segment.
        LineTo( hWorkDC, nCx1, nCy1 );
        // Draw the second segment.
        LineTo( hWorkDC, nCx2, nCy2 );

        // Draw the off-screen buffer to the visual screen.
        BitBlt( hDC, 0, 0, nWidth, nHeight, hWorkDC, 0, 0,
                SRCCOPY );
        // Select the old bitmap into the dc, delete the
        // newly-created bitmap and dc.
        SelectObject( hWorkDC, hOldBitmap );
        DeleteObject( hBitmap );
        DeleteDC( hWorkDC );
}

STDMETHODIMP CDoublePend::Update()
{
        double x1[4], f1[4], f2[4], f3[4], f4[4], hh, hhh;
        hh = h / 2.0;
        hhh = h / 6.0;
        for( int k=0; k<16; k++ ){
                int i;
                // Update the first point.
                Function( x, f1 );
                for( i=0; i<4; i++ )
                        x1[i] = x[i] + hh * f1[i];

                // Update the second point.
                Function( x1, f2 );
                for( i=0; i<4; i++ )
                        x1[i] = x[i] + hh * f2[i];

                // Update the third point.
                Function( x1, f3 );
                for( i=0; i<4; i++ )
                        x1[i] = x[i] + h * f3[i];

                // Update the fourth point.
                Function( x1, f4 );
                for( i=0; i<4; i++ )
```

```
                        x[i] += hhh * ( f1[i] + 2.0 * f2[i] +
                                f2.0 * f3[i] + f4[i] );
                }

        // Cause a window redraw.
        InvalidateRect( NULL, FALSE );
        UpdateWindow();
        return S_OK;
}

STDMETHODIMP CDoublePend::SetValue(double dValue)
{
        // Set the update rate value.
        h = dValue;
        return S_OK;
}
```

Código fuente 228

Una vez compilado el código, Visual C++ registra automáticamente el control en el registro de Windows. Para ejecutarlo abrimos la aplicación auxiliar en Tools+ActiveX Control Test Container. Dentro de esta aplicación ejecutamos el control cargándolo mediante el menú Edit+Insert New Control, que muestra todos los controles registrados incluido DoublePendulum.

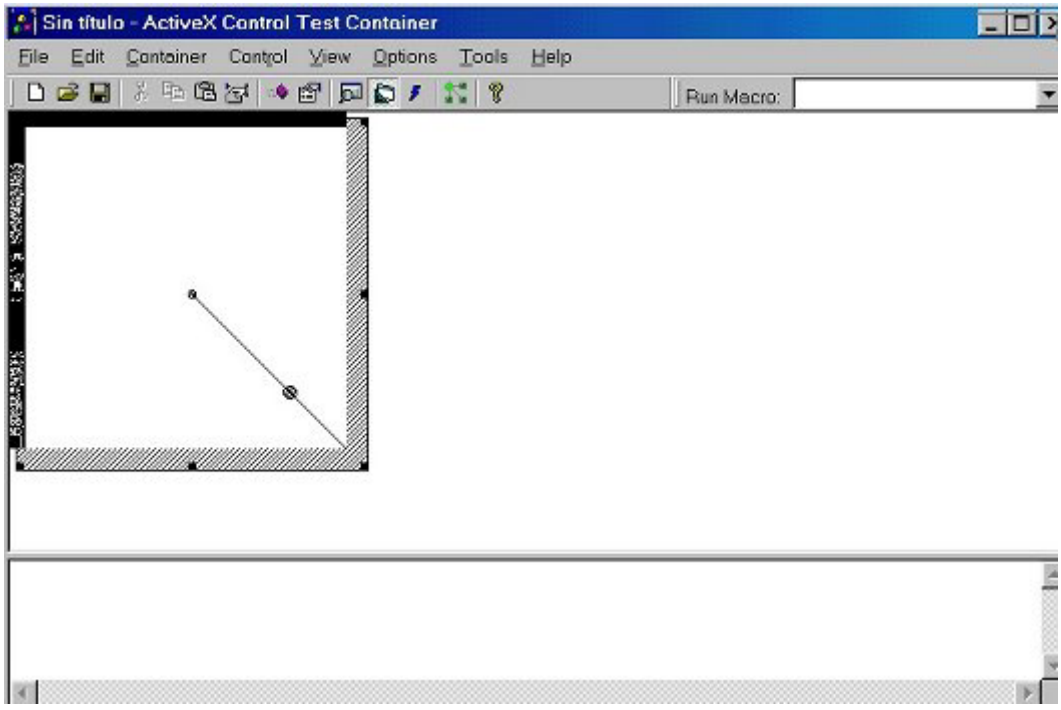


Figura 16

El paso final es crear una aplicación que utilice este control. Para ello creamos con el Wizard una aplicación simple SDI (UseDoublePendulum). Añadimos el control mediante el menú Project+Add To Project+Components and Controls. Elija la carpeta Registered ActiveX Controls, localice la clase DoublePend, e insértela en el proyecto. Luego editamos el archivo UseDoublePendulumView.h, e insertamos el archivo doublepend.h, y en la declaración pública de CUseDoublePendulumView:

```
CDoublePend m_DoublePendulum;
BOOL m_bCreated;
```

Mediante el ClassWizard creamos a continuación las funciones OnCreate(LPCREATESTRUCT lpCreateStruct) y OnTimer(UINT nIDEvent) con el Código fuente 229.

```
int CUseDoublePendulumView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        if (CView::OnCreate(lpCreateStruct) == -1)
                return -1;

        SetTimer( 1, 100, NULL );

        return 0;
}
void CUseDoublePendulumView::OnTimer(UINT nIDEvent)
{

        if( !m_bCreated ){
                m_bCreated = TRUE;
                RECT Rect;
                Rect.left = Rect.top = 0;
                Rect.right = Rect.bottom = 400;
                m_DoublePendulum.Create( "DoublePendulum",
                        WS_VISIBLE, Rect, this, 0x5005 );
                }
        else
                m_DoublePendulum.Update();

        CView::OnTimer(nIDEvent);
}
```

Código fuente 229

Con esto podemos compilar el proyecto y ver como se ejecuta. Finalmente, el control se puede ejecutar desde una página web con un navegador que acepte controles ActiveX, tal como el Internet Explorer 4.0 o superior. El código de la página HTML sería el que muestra el Código fuente 230

```
<HTML>
<HEAD>
<TITLE>ATL 3.0 - Pagina de prueba: DoublePend</TITLE>
</HEAD>
<BODY>
<OBJECT ID="DoublePend" CLASSID="CLSID:D6AA8FAD-E29D-11D1-B719-
0080AD17AF01"></OBJECT>
</BODY>
</HTML>
```

Código fuente 230

Recapitulación

La descripción en detalle de todos los elementos relacionados con la tecnología ActiveX y ATL podría ocupar fácilmente un libro. Esta presentación está dirigida a una introducción rápida que permita realizar objetos COM simples y consultar en detalle la ayuda de Visual C++ para construir elementos más complejos.

Ejercicios

5. En el proyecto UseDoublePendulum añadir un dialogo para cambiar la velocidad de desplazamiento del péndulo, cambiando SetValue.

6. Realizar los dos tutoriales incluidos en la documentación de Microsoft Visual C++ (MSDN) sobre objetos COM y ATL.

Respuestas a los ejercicios

Al tratarse de un código especial, las respuestas se encuentran en: VC-cap4