CAPÍTULO 4

ESTRUCTURAS DE CONTROL

El lenguaje C pertenece a la familia de lenguajes del paradigma de la programación estructurada. En este capítulo quedan recogidas las reglas de la programación estructurada y, en concreto, las reglas sintácticas que se exige en el uso del lenguaje C para el diseño de esas estructuras. **El objetivo del capítulo es aprender a crear estructuras condicionales y estructuras de iteración**. También veremos una estructura especial que permite seleccionar un camino de ejecución de entre varios establecidos. Y veremos las sentencias de salto que nos permiten abandonar el bloque de sentencias iteradas por una estructura de control.

Introducción.

Las reglas de la programación estructurada son:

1. Todo programa consiste en una serie de acciones o sentencias que

se ejecutan en **secuencia**, una detrás de otra.

2. Cualquier acción puede ser sustituida por dos o más acciones en secuencia. Esta **regla** se conoce como la **de apilamiento**.
3. Cualquier acción puede ser sustituida por cualquier estructura de control; y sólo se consideran tres estructuras de control: **la secuencia, la selección y la repetición**. Esta regla se conoce como **regla de anidamiento**. Todas las estructuras de control de la programación estructurada tienen un solo punto de entrada y un solo punto de salida.
4. Las reglas de apilamiento y de anidamiento pueden aplicarse tantas veces como se desee y en cualquier orden.

Ya hemos visto cómo se crea una sentencia: con un punto y coma precedido de una expresión que puede ser una asignación, la llamada a una función, una declaración de una variable, etc. O, si la sentencia es compuesta, agrupando entonces varias sentencias simples en un bloque encerrado por llaves.

Los programas discurren, de instrucción a instrucción, una detrás de otra, en una ordenación secuencial y nunca dos al mismo tiempo, como queda representado en la figura 4.1.

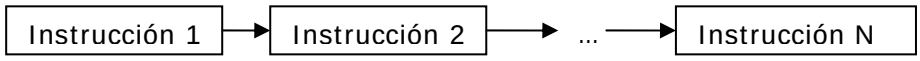

Figura 4.1.: Esquema de ordenación secuencial de sentencias.

Pero un lenguaje de programación no sólo ha de poder ejecutar las instrucciones en orden secuencial: es necesaria la capacidad para modificar ese orden de ejecución. Para ello están las estructuras de control. Al acabar este capítulo, una vez conocidas las estructuras de control, las posibilidades de resolver diferentes problemas mediante el lenguaje de programación C se habrán multiplicado enormemente.

A lo largo del capítulo iremos viendo abundantes ejemplos. Es conveniente pararse en cada uno: comprender el código que se propone en el manual, o lograr resolver aquellos que se dejan propuestos. En algunos casos ofreceremos el código en C; en otros dejaremos apuntado el modo de resolver el problema ofreciendo el pseudocódigo del algoritmo o el flujograma. Muchos de los ejemplos que aquí se van a resolver ya tienen planteado el flujograma o el pseudocódigo en el libro “Fundamentos de Informática. Codificación y Algoritmia”.

Conceptos previos.

La regla 3 de la programación estructurada habla de tres estructuras de control: la secuencia, la selección y la repetición. Nada nuevo hay ahora que decir sobre la secuencia, que vendría esquematizada en la figura 4.1. En la figura 4.2. se esquematizan diferentes posibles estructuras de selección; y en la figura 4.3. las dos estructuras básicas de repetición.

Las dos formas que rompen el orden secuencial de ejecución de sentencias son:

1. **Instrucción condicional**: Se evalúa una condición y si se cumple se transfiere el control a una nueva dirección indicada por la instrucción.
2. **Instrucción incondicional**. Se realiza la transferencia a una nueva dirección sin evaluar ninguna condición (por ejemplo, llamada a una función).

En ambos casos la transferencia del control se puede realizar con o sin retorno: en el caso de que exista retorno, después de ejecutar el bloque de instrucciones de la nueva dirección se retorna a la dirección que sucede a la que ha realizado el cambio de flujo.

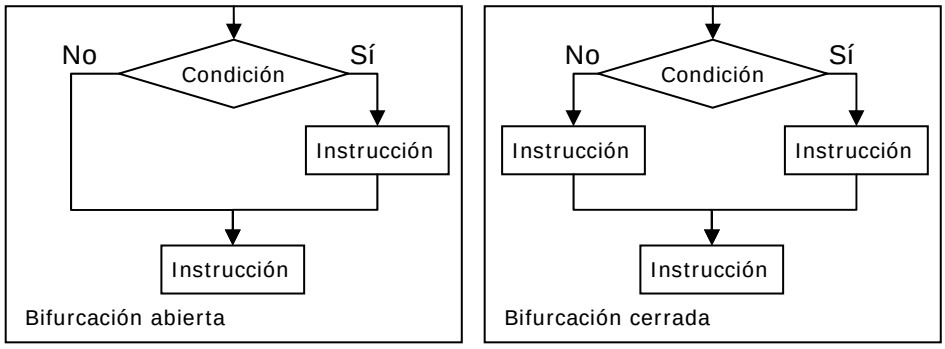

Figura 4.2.: Estructuras de selección.

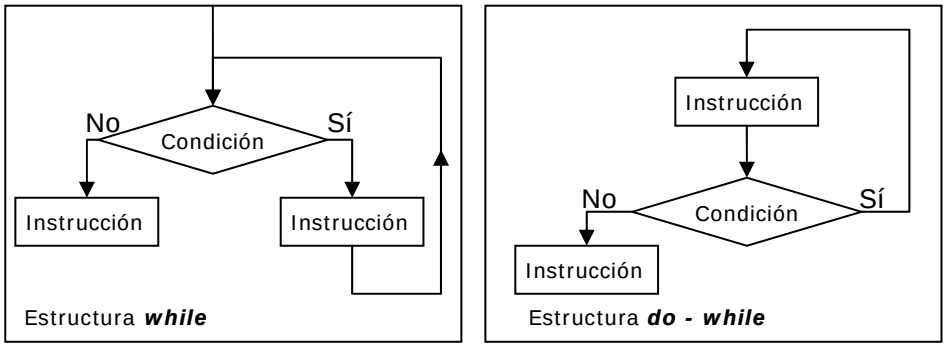

Figura 4.3.: Estructuras de repetición.

Las estructuras de control que se van a ver en este capítulo son aquellas con trasfieren el control a una nueva dirección, de acuerdo a una condición evaluada.

Estructuras de control condicionales.

Las estructuras de control condicionales que se van a ver son la bifurcación abierta y la bifurcación cerrada. Un esquema del flujo de ambas estructuras ha quedado recogido en la Figura 4.2.

- **La bifurcación abierta. La sentencia *if*.**

  La sentencia que está precedida por la estructura de control condicionada se ejecutará si la condición de la estructura de control es verdadera; en caso contrario no se ejecuta la instrucción condicionada y

continua el programa con la siguiente instrucción. En la figura 4.2. se puede ver un esquema del comportamiento de la bifurcación abierta.

La sintaxis de la estructura de control condicionada abierta es la siguiente:

if(condición) sentencia;

Si la condición es verdadera (distinto de cero en el lenguaje C), se ejecuta la sentencia. Si la condición es falsa (igual a cero en el lenguaje C), no se ejecuta la sentencia.

Si en lugar de una sentencia, se desean condicionar varias de ellas, entonces se crea una sentencia compuesta mediante llaves.

Ejemplo:

Programa que solicite dos valores enteros y muestre el cociente:

```
#include <stdio.h>
void main(void)
{
      short D, d;
      printf("Programa para dividir dos enteros...\n");
      printf("Introduzca el dividendo ... ");
      scanf("%hd",&D);
      printf("Introduzca el divisor ... ");
      scanf("%hd",&d);
      if(d != 0) printf("%hu / %hu = %hu", D, d, D / d);
}
```

Se efectuará la división únicamente en el caso en que se verifique la condición de que *d != 0*.

- **La bifurcación cerrada. La sentencia *if – else*.**

  En una bifurcación cerrada, la sentencia que está precedida por una estructura de control condicionada se ejecutará si la condición de la estructura de control es verdadera; en caso contrario se ejecuta una instrucción alternativa. Después de la ejecución de una de las dos sentencias (nunca las dos), el programa continúa la normal ejecución de las restantes sentencias que vengan a continuación.

La sintaxis de la estructura de control condicionada cerrada es la siguiente:

if(condición) sentencia1;
else sentencia2;

Si la condición es verdadera (distinto de cero en el lenguaje C), se ejecuta la sentencia llamada *sentencia1*. Si la condición es falsa (igual a cero en el lenguaje C), se ejecuta la sentencia llamada *sentencia2*.

Si en lugar de una sentencia, se desean condicionar varias de ellas, entonces se crea una sentencia compuesta mediante llaves.

Ejemplo:

El mismo programa anteriormente visto. Quedará mejor si se escribe de la siguiente forma:

```
#include <stdio.h>
void main(void)
{
      short D, d;
      printf("Programa para dividir dos enteros...\n");
      printf("Introduzca el dividendo ... ");
      scanf("%hd",&D);
      printf("Introduzca el divisor ... ");
      scanf("%hd",&d);
      if(d != 0) printf("%hu / %hu = %hu", D, d, D / d);
      else printf("No se puede realizar division por cero");
}
```

Se efectuará la división únicamente en el caso en que se verifique la condición de que *d != 0*. Si el divisor introducido es igual a cero, entonces imprime en pantalla un mensaje de advertencia.

- **Anidamiento de estructuras condicionales.**

  Decimos que se produce anidamiento de estructuras de control cuando una estructura de control aparece dentro de otra estructura de control del mismo tipo.

  Tanto en la parte ***if*** como en la parte ***else***, los anidamientos pueden llegar a cualquier nivel. De esa forma podemos elegir entre numerosas sentencias estableciendo las condiciones necesarias.

Una estructura de anidamiento tiene, por ejemplo, la forma:

```
if(expresión_1)                 /* primer if */
{
      if(expresión_2)           /* segundo if */
      {
            if(expresión_3)     /* tercer if */
                  sentencia_1;
            else                /* alternativa al tercer if */
                  sentencia_2;
      }
      else                      /* alternativa al 2º if */
            sentencia_3;
}
else                            /* alternativa al primer if */
      sentencia_4;
```

(se puede ver el organigrama de este código en la figura 4.4.)

Cada ***else*** se asocia al ***if*** más próximo en el bloque en el que se encuentre y que no tenga asociado un ***else***. No está permitido (no tendría sentido) utilizar un ***else*** sin un ***if*** previo. Y la estructura ***else*** debe ir inmediatamente después de la sentencia condicionada con su ***if***.

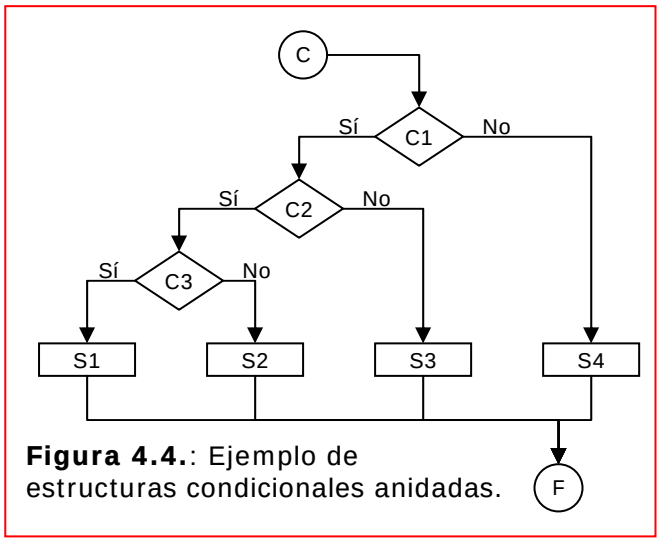

Figura 4.4.: Ejemplo de estructuras condicionales anidadas.

Un ejemplo de estructura anidada sería, siguiendo con los ejemplos anteriores, el caso de que, si el divisor introducido ha sido el cero, el programa preguntase si se desea introducir un divisor distinto.

```
#include <stdio.h>
void main(void)
{
	short D, d;
	char opcion;
	printf("Programa para dividir dos enteros...\n");
	printf("Introduzca el dividendo ... ");
	scanf("%hd",&D);
	printf("Introduzca el divisor ... ");
	scanf("%hd",&d);
	if(d != 0)
		printf("%hu / %hu = %hu", D, d, D / d);
	else
	{
		printf("No se puede dividir por cero.\n");
		printf("¿Introducir otro denominador (s/n)?");
		opcion = getchar();
		if(opcion == 's')
		{
			printf("\nNuevo denominador ... ");
			scanf("%hd",&d);
			if(d != 0)
				printf("%hu / %hu = %hu", D, d, D/d);
			else
				printf("De nuevo ha introducido 0.");
		}
	}
}
```

La función *getchar()* está definida en la biblioteca **stdio.h**. Esta función espera a que el usuario pulse una tecla del teclado y, una vez pulsada, devuelve el código ASCII de la tecla pulsada.

En este ejemplo hemos llegado hasta un tercer nivel de anidación.

- **Escala *if - else – if***

Cuando se debe elegir entre una lista de opciones, y únicamente una de ellas ha de ser válida, se llega a producir una concatenación de condiciones de la siguiente forma:

```
if(condición1) setencia1;
else
{
	if(condición2) sentencia2;
	else
	{
		if(condición3) sentencia3;
```

```
            else sentencia4;
      }
}
```

El flujograma recogido en la Figura 4.4. representaría esta situación sin más que intercambiar los caminos de verificación de las condiciones C1, C2 y C3 recogidas en él (es decir, intercambiando los rótulos de “Sí” y de “No”).

Este tipo de anidamiento se resuelve en C con la estructura ***else if***, que permite una concatenación de las condicionales. Un código como el antes escrito quedaría:

```
if(condición1) sentencia1;
else if (condición2) sentencia2;
else if(condición3) sentencia3;
else sentencia4;
```

Como se ve, una estructura así anidada se escribe con mayor facilidad y expresa también más claramente las distintas alternativas. No es necesario que, en un anidamiento de sentencias condicionales, encontremos un ***else*** final: el último ***if*** puede ser una bifurcación abierta:

```
if(condición1) sentencia1;
else if (condición2) sentencia2;
else if(condición3) sentencia3;
```

Un ejemplo de concatenación podría ser el siguiente programa, que solicita al usuario la nota de un examen y muestra por pantalla la calificación académica obtenida:

```
#include <stdio.h>
void main(void)
{
      float nota;
      printf("Introduzca la nota del examen ... ");
      scanf("%f",&nota);
      if(nota < 0 || nota > 10) printf("Nota incorrecta.");
      else if(nota < 5) printf("Suspenso.");
      else if(nota < 7) printf("Aprobado.");
      else if(nota < 9) printf("Notable.");
      else if(nota < 10) printf("Sobresaliente.");
      else printf("Matrícula de honor.");
}
```

Únicamente se evaluará un ***else if*** cuando no haya sido cierta la condición de ninguno de los anteriores ni del ***if*** inicial. Si todas las condiciones resultan ser falsas, entonces se ejecutará (si existe) el último ***else***.

- **La estructura condicional y el operador condicional**

  Existe un operador que selecciona entre dos opciones, y que realiza, de forma muy sencilla y bajo ciertas limitaciones la misma operación que la estructura de bifurcación cerrada. Es el **operador interrogante, dos puntos** (***?:***).

  La sintaxis del operador es la siguiente:

  expresión_1 ? expresión_2 : expresión3;

  Se evalúa *expresión_1*; si resulta ser verdadera, entonces se ejecutará la sentencia recogida en *expresión_2*; y si es falsa, entonces se ejecutará la sentencia recogida en *expresión_3*. Tanto *expresión_2* como *expresión_3* pueden ser funciones, o expresiones muy complejas, pero siempre deben ser sentencias simples.

  Es conveniente no renunciar a conocer algún aspecto de la sintaxis de un lenguaje de programación. Es cierto que el operador "*interrogante dos puntos*" se puede siempre sustituir por la estructura de control condicional ***if – else***. Pero el operador puede, en muchos casos, simplificar el código o hacerlo más elegante. Y hay que tener en cuenta que el resto de los programadores sí hacen uso del operador y el código en C está lleno de ejemplos de su uso.

  Por ejemplo, el código:

```
if(x >= 0)
        printf("Positivo\n");
else
        printf("Negativo\n");
```

  es equivalente a: *printf("%s\n", x >= 0* ***?*** *"Positivo"****:*** *"Negativo");*

Estructura de selección múltiple: Sentencia ***switch***.

La sentencia ***switch*** permite transferir el control de ejecución del programa a un punto de entrada etiquetado en un bloque de sentencias. La decisión sobre a qué instrucción del bloque se trasfiere la ejecución se realiza mediante una expresión entera.

La forma general de la estructura ***switch*** es:

```
switch(variable_del_switch)
{
        case expresionConstante1:
                [sentencias;]
                 [break;]
        case expresionConstante2:
                [sentencias;]
                 [break;]
        [...]
        case expresionConstanteN:
                [sentencias;]
                 [break;]
        [default
                 sentencias;]
}
```

El cuerpo de la sentencia ***switch*** se conoce como bloque ***switch*** y permite tener sentencias prefijadas con las etiquetas ***case***. Una etiqueta ***case*** es una constante entera (variables de tipo ***char*** ó ***short*** ó ***long***, con o sin signo). Si el valor de la expresión de ***switch*** coincide con el valor de una etiqueta ***case***, el control se transfiere a la primera sentencia que sigue a la etiqueta. No puede haber dos ***case*** con el mismo valor de constante. Si no se encuentra ninguna etiqueta ***case*** que coincida, el control se transfiere a la primera sentencia que sigue a la etiqueta ***default***. Si no existe esa etiqueta ***default***, y no existe una etiqueta coincidente, entonces no se ejecuta ninguna sentencia del ***switch*** y se continúa, si la hay, con la siguiente sentencia posterior a la estructura.

Por ejemplo, para el código que se muestra a continuación, y cuyo flujograma queda recogido en la figura 4.5.:

```
switch(a)
{
      case 1:  printf("UNO\t");
      case 2:  printf("DOS\t");
      case 3:  printf("TRES\t");
      default: printf("NINGUNO\n");
}
```

Si el valor de *a* es, por ejemplo, 2, entonces comienza a ejecutar el código del bloque a partir de la línea que da entrada el ***case 2:***. Producirá la siguiente salida por pantalla:

```
DOS   TRES  NINGUNO.
```

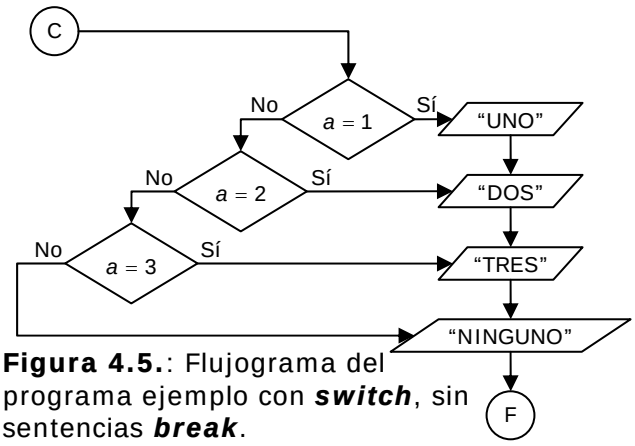

Figura 4.5.: Flujograma del programa ejemplo con ***switch***, sin sentencias ***break***.

Una vez que el control se ha trasferido a la sentencia que sigue a una etiqueta concreta, ya se ejecutan todas las demás sentencias del bloque ***switch***, de acuerdo con la semántica de dichas sentencias. El que aparezca una nueva etiqueta ***case*** no obliga a que se dejen de ejecutar las sentencias del bloque. Si se desea detener la ejecución de sentencias en el bloque ***switch***, debemos transferir explícitamente el control al exterior del bloque. Y eso se realiza utilizando la sentencia ***break***. Dentro de un bloque ***switch***, la sentencia ***break*** transfiere el control a la primera sentencia posterior al ***switch***. Ese es el motivo por el que en la sintaxis de la estructura ***switch*** se escriba (en forma opcional) las sentencias ***break*** en las instrucciones inmediatamente anteriores a cada una de las etiquetas.

En el ejemplo anterior, si colocamos la sentencia ***break*** en cada ***case***,

```
switch(a)
{
      case 1: printf(“UNO”);
      break;
      case 2: printf(“DOS”);
      break;
      case 3: printf(“TRES”);
      break;
      default: printf(“NINGUNO”);
}
```

Entonces la salida por pantalla, si la variable a tiene el valor 2 será únicamente:

```
DOS
```

(Puede verse el flujograma de este nuevo código en la figura 4.6.)

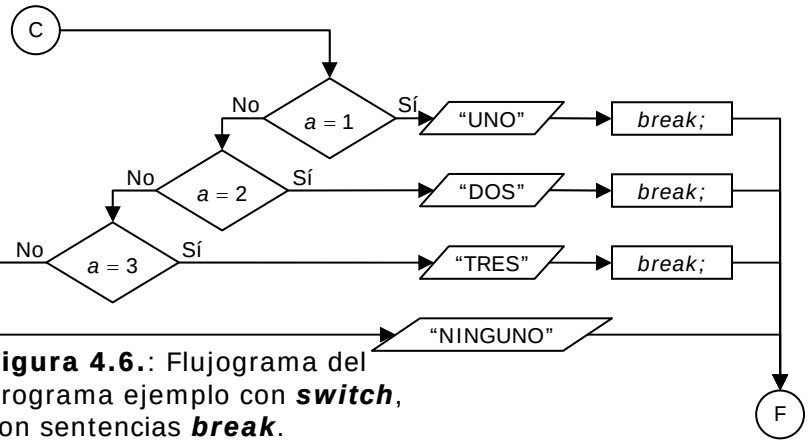

Figura 4.6.: Flujograma del programa ejemplo con ***switch***, con sentencias ***break***.

La ejecución de las instrucciones que siguen más allá de la siguiente etiqueta ***case*** puede ser útil en algunas circunstancias. Pero lo habitual será que aparezca una sentencia ***break*** al final del código de cada etiqueta ***case***.

Una sola sentencia puede tener más de una etiqueta ***case***. Queda claro en el siguiente ejemplo:

```
short int nota;
printf("Introduzca la nota del examen ... ");
scanf("%hd",&nota);
```

```
switch(nota)
{
      case 1:
      case 2:
      case 3:
      case 4:     printf("SUSPENSO");
                  break;
      case 5:
      case 6:     printf("APROBADO");
                  break;
      case 7:
      case 8:     printf("NOTABLE");
                  break;
      case 9:     printf("SOBRESALIENTE");
                  break;
      case 10:    printf("MATRÍCULA DE HONOR");
                  break;
      default:    printf("Nota introducida errónea.");
}
```

No se puede poner una etiqueta ***case*** fuera de un bloque ***switch***. Y tampoco tiene sentido colocar instrucciones dentro del bloque ***switch*** antes de aparecer el primer ***case***: eso supondría un código que jamás podría llegar a ejecutarse. Por eso, la primera sentencia de un bloque ***switch*** debe estar ya etiquetada.

Se pueden anidar distintas estructuras ***switch***.

El ejemplo de las notas, que ya se mostró al ejemplificar una anidación de sentencias ***if–else–if*** puede servir para comentar una característica importante de la estructura ***switch***. Esta estructura no admite, en sus distintas entradas ***case***, ni expresiones lógicas o relacionales, ni expresiones aritméticas, sino literales. La única relación aceptada es, pues, la de igualdad. Y además, el término de la igualdad es siempre entre una variable o una expresión entera (la del ***switch***) y valores literales: no se puede indicar el nombre de una variable. El programa de las notas, si la variable *nota* hubiese sido de tipo ***float***, como de hecho quedo definida cuando se resolvió el problema con los condicionales ***if–else–if*** no tiene solución posible mediante la estructura ***switch***.

Y una última observación: las sentencias de un ***case*** no forman un bloque y no tiene porqué ir entre llaves. La estructura ***switch*** entera, con todos sus ***case***'s, sí es un bloque.

Un ejercicio planteado.

Planteamos ahora un ejercicio a resolver: solicite del usuario que introduzca por teclado un día, mes y año, y muestre entonces por pantalla el día de la semana que le corresponde.

La resolución de este problema es sencilla si se sabe el cómo. Sin un correcto algoritmo que nos permita saber cómo procesar la entrada no podemos hacer nada.

Por lo tanto, antes de intentar implementar un programa que resuelva este problema, será necesario preguntarse si somos capaces de resolverlo sin programa. Porque si no sabemos hacerlo nosotros, menos sabremos explicárselo a la máquina.

Buscando en Internet he encontrado lo siguiente: Para saber a qué día de la semana corresponde una determinada fecha, basta aplicar la siguiente expresión:

$$d = \left[(26 \times M - 2)/10 + D + A + A/4 + C/4 - 2 \times C\right] \bmod 7$$

Donde d es el día de la semana ($d = 0$ es el domingo; $d = 1$ es el lunes,..., $d = 6$ es el sábado); D es el día del mes de la fecha; M es el mes de la fecha; A es el año de la fecha; y C es la centuria (es decir, los dos primero dígitos del año) de la fecha.

A esos valores hay que introducirle unas pequeñas modificaciones: se considera que el año comienza en marzo, y que los meses de enero y febrero son los meses 11 y 12 del año anterior.

Hagamos un ejemplo a mano: ¿Qué día de la semana fue el 15 de febrero de 1975?:

- $D = 15$
- $M = 12$: hemos quedado que en nuestra ecuación el mes de febrero es el décimo segundo mes del año anterior.
- $A = 74$: hemos quedado que el mes de febrero corresponde al último mes del año anterior.
- $C = 19$

Con todos estos valores, el día de la semana queda:

$$d = \left[(26 \times 12 - 2)/10 + 15 + 74 + 74/4 + 19/4 - 2 \times 19\right] \bmod 7$$

que es igual a 6, es decir, sábado.

Sólo queda hacer una última advertencia a tener en cuenta a la hora de calcular nuestros valores de A y de C: Si queremos saber el día de la semana del 1 de febrero de 2000, tendremos que $M = 12$, que $A = 99$ y que $C = 19$: es decir, primero convendrá hacer las rectificaciones al año y sólo después calcular los valores de A y de C. Ése día fue…

$$d = \left[(26 \times 12 - 2)/10 + 1 + 99 + 99/4 + 19/4 - 2 \times 19\right] \bmod 7 = 2$$

es decir… ¡martes!

Queda ahora hacer el programa que nos dé la respuesta al día de la semana en el que estamos. Hará falta emplear dos veces la estructura de control condicional ***if*** y una vez el ***switch***. El programa queda como sigue:

```
#include <stdio.h>

void main(void)
{
      unsigned short D, mm, aaaa;
      unsigned short M, A, C;

      printf("Introduzca la fecha ... \n");
      printf("Día ... ");
      scanf("%hu", &D);

      printf("Mes ... ");
      scanf("%hu", &mm);
```

```
      printf("Año ... ");
      scanf("%hu", &aaaa);

// Valores de las variables:

// El valor de D ya ha quedado introducido por el usuario.
// Valor de M:

      if(mm< 3)
      {
            M = mm + 10;
            A = (aaaa - 1) % 100;
            C = (aaaa - 1) / 100;
      }
      else
      {
            M = mm - 2;
            A = aaaa % 100;
            C = aaaa / 100;
      }

      printf("El día %2hu de %2hu de %4hu fue ",D, mm, aaaa);
      switch((70+(26*M-2)/10 + D + A + A/4 + C/4 - C*2 ) % 7)
      {
      case 0: printf("DOMINGO");     break;
      case 1: printf("LUNES");       break;
      case 2: printf("MARTES");      break;
      case 3: printf("MIÉRCOLES");   break;
      case 4: printf("JUEVES");      break;
      case 5: printf("VIERNES");     break;
      case 6: printf("SÁBADO");
      }
}
```

Si, por ejemplo, introducimos la fecha 25 de enero de 1956, la salida del programa tendrá el siguiente aspecto:

```
Introduzca la fecha ...
Día ... 25
Mes ... 1
Año ... 1956
El día 25 de  1 de 1956 fue MIÉRCOLES
```

Falta aclarar por qué he sumado 70 al valor de la expresión que calculamos en el **switch**. Veamos un ejemplo para justificar ese valor: supongamos la fecha 2 de abril de 2001. Tendremos que $D = 2$, $M = 2$, $A = 1$ y $C = 20$, y entonces el valor de d queda: $-27\%7$ que es igual a -6. Nuestro algoritmo trabaja con valores entre 0 y 6, y al salir

negativo el valor sobre el que se debe calcular el módulo de 7, el resultado nos sale fuera de ese rango de valores. Pero la operación módulo establece una relación de equivalencia entre el conjunto de los enteros y el conjunto de valores comprendidos entre 0 y el valor del módulo menos 1. Le sumamos al valor calculado un múltiplo de 7 suficientemente grande para que sea cual sea el valor de las variables, al final obtenga un resultado positivo. Así, ahora, el valor obtenido será $70 - 27\%7 = 43\%7 = 1$, es decir, lunes:

```
Introduzca la fecha ...
Día ... 2
Mes ... 4
Año ... 2001
El día  2 de  4 de 2001 fue LUNES
```

Estructuras de repetición. Iteración.

Una estructura de repetición o de iteración es aquella que nos permite repetir un conjunto de sentencias mientras que se cumpla una determinada condición.

Las estructuras de iteración o de control de repetición, en C, se implementan con las estructuras ***do–while***, ***while*** y ***for***. Todas ellas permiten la anidación de unas dentro de otras a cualquier nivel. Puede verse un esquema de su comportamiento en la figura 4.3., en páginas anteriores.

- **Estructura *while*.**

  La estructura ***while***, también llamada condicional, o centinela, se emplea en aquellos casos en que no se conoce por adelantado el número de veces que se ha de repetir la ejecución de una determinada sentencia o bloque: **ninguna, una o varias**.

  La sintaxis de la estructura ***while*** es la que sigue:

  while(condición) sentencia;

donde *condición* es cualquier expresión válida en C. Esta expresión se evalúa cada vez, antes de la ejecución de la sentencia iterada (o bloque de sentencias si se desea iterar una sentencia compuesta). Puede por tanto no ejecutarse nunca el bloque de sentencias de la estructura de control. Las sentencias se volverán a ejecutar una y otra vez mientras *condición* siga siendo verdadero. Cuando la condición resulta ser falsa, entonces el contador de programa se sitúa en la inmediata siguiente instrucción posterior a la sentencia gobernada por la estructura.

Veamos un ejemplo sencillo. Hagamos un programa que solicite un entero y muestre entonces por pantalla la tabla de multiplicar de ese número. El programa es muy sencillo gracias a las sentencias de repetición:

```
#include <stdio.h>
void main(void)
{
	short int n,i;
	printf("Tabla de multiplicar del ... ");
	scanf("%hd",&n);
	i = 0;
	while(i <= 10)
	{
		printf("%3hu * %3hu = %3hu\n",i,n,i * n);
		i++;
	}
}
```

Después de solicitar el entero, inicializa a 0 la variable *i* y entonces, mientras que esa variable contador sea menor o igual que 10, va mostrando el producto del entero introducido por el usuario con la variable contador *i*. La variable *i* cambia de valor dentro del bucle de la estructura, de forma que llega un momento en que la condición deja de cumplirse; ¿cuándo?: cuando la variable *i* tiene un valor mayor que 10.

Conviene asegurar que en algún momento va a dejar de cumplirse la condición; de lo contrario la ejecución del programa podría quedarse atrapada en un **bucle infinito**. De alguna manera, dentro de la sentencia gobernada por la estructura de iteración, hay que modificar

alguno de los parámetros que intervienen en la condición. Más adelante, en este capítulo, veremos otras formas de salir de la iteración.

Este último ejemplo ha sido sencillo. Veamos otro ejemplo que requiere un poco más de imaginación. Supongamos que queremos hacer un programa que solicite al usuario la entrada de un entero y que entonces muestre por pantalla el factorial de ese número. Ya se sabe la definición de factorial: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$.

Antes de mostrar el código de esta sencilla aplicación, conviene volver a una idea comentada capítulos atrás. Efectivamente, habrá que saber decir en el lenguaje C cómo se realiza esta operación. Pero previamente debemos ser capaces de expresar el procedimiento en castellano. En el capítulo 4 de “Fundamentos de Informática. Codificación y Algoritmia.” se encuentra extensamente documentado este y otros muchos algoritmos de iteración que veremos ahora implementados en C, en estas páginas.

Veamos una posible solución al programa del factorial:

```
#include <stdio.h>
void main(void)
{
      unsigned short n;
      unsigned long Fact;
      printf("Introduzca el entero ... ");
      scanf("%hu",&n);
      printf("El factorial de %hu es ... ", n);
      Fact = 1;
      while(n != 0)
      {
            Fact = Fact * n;
            n = n - 1;
      }
      printf("%lu.", Fact);
```

El valor de la variable *Fact* se inicializa a uno antes de comenzar a usarla. Efectivamente es muy importante no emplear esa variable sin darle el valor inicial que a nosotros nos interesa. La variable *n* se inicializa con la función *scanf*.

Mientras que *n* no sea cero, se irá multiplicando *Fact* (inicialmente a uno) con *n*. En cada iteración el valor de *n* se irá decrementando en uno.

La tabla de los valores que van tomando ambas variables se muestra en la tabla 4.1. (se supone que la entrada por teclado ha sido el número 5). Cuando la variable *n* alcanza el valor cero termina la iteración. En ese momento se espera que la variable *Fact* tenga el valor que corresponde al factorial del valor introducido por el usuario.

La iteración se ha producido tantas veces como el cardinal del número introducido. Por cierto, que si el usuario hubiera introducido el valor cero, entonces el bucle no se hubiera ejecutado ni una sola vez, y el valor de *Fact* hubiera sido uno, que es efectivamente el valor por definición ($0! = 1$).

n	5	4	3	2	1	0
Fact	5	20	60	120	120	120

Tabla 4.1.: Valores que van tomando las variables del bucle del programa del cálculo del factorial si la entrada del usuario ha sido *n* = 5.

La estructura de control mostrada admite formas de presentación más compacta y, de hecho, lo habitual será que así se presente. Por ejemplo:

```
while(n != 0)
{
      Fact *= n;
      n = n - 1;
}
```

donde todo es igual excepto que ahora se ha hecho uso del operador compuesto en la primera sentencia del bloque. Pero aún se puede compactar más:

1. La condición de permanencia será verdad siempre que *n* no sea cero. Y por definición de verdad en C (algo es verdadero cuando es

distinto de cero) se puede decir que la *n != 0* es verdadero sí y sólo si *n* es verdadero.

2. Las dos sentencias simples iteradas en el bloque pueden condensarse en una sola: *Fact *= n--;*

Así las cosas, la estructura queda:

```
while(n) Fact *= n--;
```

Hacemos un comentario más sobre la estructura ***while***. Esta estructura permite iterar una sentencia sin cuerpo. Por ejemplo, supongamos que queremos hacer un programa que solicite continuamente del usuario que pulse una tecla, y que esa solicitud no cese hasta que éste introduzca el carácter, por ejemplo, 'a'. La estructura quedará tan simple como lo que sigue:

```
while((ch = getchar()) != 'a');
```

Esta línea de programa espera una entrada por teclado. Cuando ésta se produzca comprobará que hemos tecleado el carácter 'a' minúscula; de no ser así, volverá a esperar otro carácter.

Una forma más sencilla ó fácil de ver el significado de esta última línea de código vista sería expresarlo de la siguiente manera:

```
while(ch != 'a')
      ch = getchar();
```

Un último ejemplo clásico de uso de la estructura ***while***. El cálculo del **máximo común divisor** de dos enteros que introduce el usuario por teclado.

No es necesario explicar el concepto de máximo común divisor. Sí es necesario en cambio explicar un método razonable de plantear al ordenador cómo se calcula ese valor: porque este ejemplo deja claro la importancia de tener no sólo conocimientos de lenguaje de programación, sino también de algoritmos válidos.

Euclides, matemático del siglo V a. de C. presentó un algoritmo muy fácil de implementar, y de muy bajo coste computacional. El algoritmo de Euclides dice que el máximo común divisor de dos enteros a_1 y b_1 (diremos $mcd(a_1, b_1)$), donde $b_1 \neq 0$ es igual a $mcd(a_2, b_2)$ donde $a_2 = b_1$ y donde $b_2 = a_1 \% b_1$, entendiendo por $a_1 \% b_1$ el resto de la división de a_1 con b_1. Y el proceso puede seguirse hasta llegar a unos valores de a_i y de b_i que verifiquen que $a_i \neq 0$, $b_i \neq 0$ y $a_i \% b_i = 0$. Entonces, el algoritmo de Euclides afirma que, llegado a estos valores el valor buscado es $mcd(a_1, b_1) = b_i$.

Ahora falta poner esto en lenguaje C. Ahí va:

```
#include <stdio.h>
void main(void)
{
	short int a, b, aux;
	short int mcd;
	printf("Valor de a ... ");
	scanf("%hd",&a);
	printf("Valor de b ... ");
	scanf("%hd",&b);
	printf("El mcd de %hd y %hd es ... ", a, b);
	while(b)
	{
		aux = a % b;
		a = b;
		b = aux;
	}
	printf("%hu", a);
}
```

(De nuevo recordamos que se encuentran en el manual complementario a éste, titulado “Fundamentos de Informática. Codificación y Algoritmia.” explicaciones a este nuevo algoritmo y a otros muchos que veremos en estas páginas.)

Hemos tenido que emplear una variable auxiliar, que hemos llamado *aux*, para poder hacer el intercambio de variables: que *a* pase a valer el valor de *b* y *b* el del resto de dividir *a* por *b*.

Así como queda escrito el código, se irán haciendo los intercambios de valores en las variables *a* y *b* hasta llegar a un valor de *b* igual a cero;

entonces, el anterior valor de *b* (que está guardado en *a*) será el máximo común divisor.

- **Estructura *do–while*.**

  La estructura ***do–while*** es muy similar a la anterior. La diferencia más sustancial está en que con esta estructura el código de la iteración se ejecuta, **al menos, una vez**. Si después de haberse ejecutado, la condición se cumple, entonces vuelve a ejecutarse, y así hasta que la condición no se cumpla. Puede verse un esquema de su comportamiento en la figura 4.3., en páginas anteriores.

  La sintaxis de la estructura es la siguiente:

  do sentencia while(condición);

  Y si se desea iterar un bloque de sentencias, entonces se agrupan en una sentencia compuesta mediante llaves.

  Habitualmente, toda solución a un problema resuelto con una estructura, es también solventable, de forma más o menos similar, por cualquiera de las otras dos estructuras. Por ejemplo, la tabla de multiplicar quedaría:

```
#include <stdio.h>
void main(void)
{
      short int n,i = 0;
      printf("Tabla de multiplicar del ... ");
      scanf("%hd",&n);
      do
      {
            printf("%3hu * %3hu = %3hu\n",i,n,i * n);
            i++:
      }while(i <= 10);
}
```

  Una estructura muy habitual en un programa es ejecutar unas instrucciones y luego preguntar al usuario, antes de terminar la ejecución de la aplicación, si desea repetir el proceso. Supongamos, por ejemplo, que queremos un programa que calcule el factorial de tantos números como desee el usuario, hasta que no quiera continuar. El

código ahora requiere de otra estructura de repetición, que vuelva a ejecutar el código mientras que el usuario no diga basta. Una posible codificación de este proceso sería (ver figura 4.7.):

```
#include <stdio.h>
void main(void)
{
      unsigned short n;
      unsigned long Fact;
      char opcion;
      do
      {
            Fact = 1;
            printf("\n\nIntroduzca el entero ... ");
            scanf("%hu",&n);
            printf("El factorial de %hu es ... ", n);
            while(n != 0) Fact *= n--;
            printf("%lu.", Fact);
            printf("\n\nCalcular otro factorial (s/n) ");
      }while(opcion = getchar() == 's');
}
```

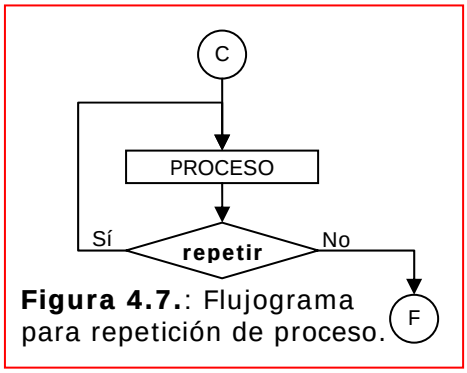

Figura 4.7.: Flujograma para repetición de proceso.

La estructura ***do–while*** repetirá el código que calcula el factorial del entero solicitado mientras que el usuario responda con una ‘s’ a la pregunta de si desea que se calcule otro factorial.

Podemos afinar un poco más en la presentación. Vamos a rechazar cualquier contestación que no sea o ‘s’ o ‘n’: si el usuario responde ‘s’, entonces se repetirá la ejecución del cálculo del factorial; si responde ‘n’ el programa terminará su ejecución; y si el usuario responde cualquier otra letra, entonces simplemente ignorará la respuesta y seguirá esperando una contestación válida.

El código queda ahora de la siguiente manera:

```
#include <stdio.h>
void main(void)
{
      unsigned short n;
      unsigned long Fact;
      char opcion;
      do
      {
            Fact = 1;
            printf("\n\nIntroduzca el entero ... ");
            scanf("%hu",&n);
            printf("El factorial de %hu es ... ", n);

            while(n != 0) Fact *= n--;

            printf("%lu.", Fact);
            printf("\n\nCalcular otro factorial (s/n) ");

            do
                  opcion = getchar();
            while (opcion != 's' && opcion != 'n');
      }while(opcion = getchar() == 's');
}
```

Ahora el valor de la variable *opcion* se irá pidiendo mientras que el usuario no introduzca correctamente una de las dos respuestas válidas: o sí ('s'), o no ('n'). El flujograma de esta nueva solución queda recogido en la figura 4.8.

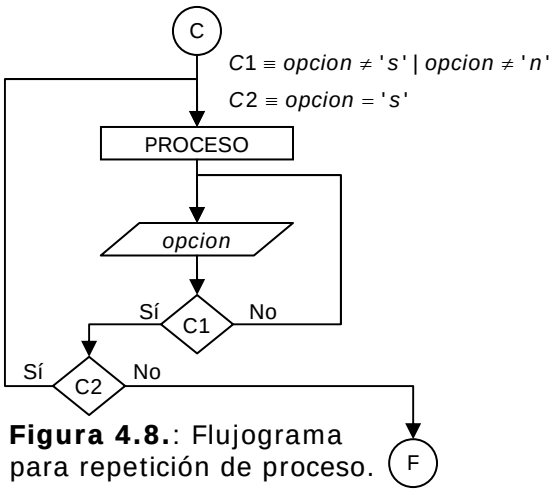

Figura 4.8.: Flujograma para repetición de proceso.

- **Estructura *for*.**

Una estructura ***for*** tiene una sintaxis notablemente distinta a la indicada para las estructuras ***while*** y ***do–while***. Pero la función que realiza es la misma. Con la palabra reservada ***for*** podemos crear estructuras de control que se dicen "**controladas por variable**".

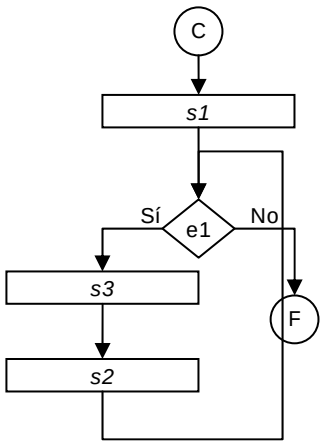

Figura 4.9.: Flujograma la estructura de control ***for****(s1 ; e1 ; s2) s3;*

La sintaxis de la estructura ***for*** es la siguiente:

for(sentencias_1 , expresión ; sentencias_2) sentencia_3;

Donde ***sentencia3*** es **la sentencia que se itera**, la que queda gobernada por la estructura de control ***for***.

Donde ***sentencias_1*** es un grupo de sentencias que se ejecutan antes que ninguna otra en una estructura ***for***, y siempre se ejecutan una vez y sólo una vez. Son sentencias, separadas por el operador coma, de **inicialización de variables**.

Donde ***expresión*** es la **condición de permanencia** en la estructura ***for***. Siempre que se cumpla expresión volverá a ejecutarse la sentencia iterada por la estructura ***for***.

Donde ***sentencias_2*** son un grupo de sentencias que se ejecutan **después de la sentencia iterada** (después de *sentencia_3*).

El orden de ejecución es, por tanto (ver flujograma en figura 4.9.):

1. Se inicializan variables según el código recogido en *sentencias_1*.
2. Se verifica la condición de permanencia recogida en *expresión*. Si *expresión* es verdadero se sigue en el paso 3; si es falso entonces se sigue en el paso 6.
3. Se ejecuta la sentencia iterada llamada, en nuestro esquema de sintaxis, *sentencia_3*.
4. Se ejecutan las sentencias recogidas en *sentencias_2*.
5. Vuelta al paso 2.
6. Fin de la iteración.

Así, una estructura ***for*** es equivalente a una estructura ***while*** de la forma:

```
sentencias_1;
while(expresión)
{
      sentencia_3;
      sentencias_2;
}
```

Por ejemplo, veamos un programa que muestra por pantalla los enteros pares del 1 al 100:

```
#include <stdio.h>
void main(void)
{
      short i;
      for(i = 2 ; i <= 100 ; i += 2)
            printf("%5hd",i);
}
```

Si queremos mejorar la presentación, podemos hacer que cada cinco pares comience una nueva fila:

```
#include <stdio.h>
void main(void)
```

```
{
      short i;
      for(i = 2 ; i <= 100 ; i += 2)
      {
            printf("%5hd",i);
            if(i % 10 == 0) printf("\n");
      }
}
```

Ya hemos dicho que en cada uno de los tres espacios de la estructura ***for*** destinados a recoger sentencias o expresiones, pueden consignarse una expresión, o varias, separadas por comas, o ninguna. Y la sentencia iterada mediante la estructura ***for*** puede tener cuerpo, o no. Veamos por ejemplo, el cálculo del factorial de un entero mediante una estructura ***for***:

```
for(Fact = 1 ; n ; Fact *= n--);
```

Esta estructura ***for*** no itera más que la sentencia punto y coma. Toda la trasformación que deseamos realizar queda en la expresión del cálculo del factorial mediante la expresión *Fact *= n--*.

El punto y coma debe ponerse: **toda estructura de control actúa sobre una sentencia**. Si no queremos, con la estructura ***for***, controlar nada, entonces la solución no es no poner nada, sino poner una sentencia vacía.

Todos los ejemplos que hasta el momento hemos puesto en la presentación de las estructuras ***while*** y ***do – while*** se pueden rehacer con una estructura ***for***. En algunos casos es más cómodo trabajar con la estructura ***for***; en otros se hace algo forzado. Veamos algunos ejemplos implementados ahora con la estructura ***for***:

Código para ver la tabla de multiplicar:

```
#include <stdio.h>
void main(void)
{     short int n,i;
      printf("Tabla de multiplicar del ... ");
      scanf("%hd",&n);
      for(i = 0 ; i <= 10 ; i++)
            printf("%3hu * %3hu = %3hu\n",i,n,i * n);
}
```

Bloquear la ejecución hasta que se pulse la tecla ‘a’:

```
for( ; ch != ‘a’ ; ch = getchar());
```

Búsqueda del máximo común divisor de dos enteros:

```
for( ; b ; )
{
      aux = a % b;
      a = b;
      b = aux;
}
printf("%hu", a);
```

Sentencias de salto: ***break*** y ***continue***.

Hay dos sentencias que modifican el orden del flujo de instrucciones dentro de una estructura de iteración. Son las sentencias ***break*** y ***continue***. Ambas sentencias, en una estructura de iteración, se presentan siempre condicionadas.

Una sentencia ***break*** dentro de un bloque de instrucciones de una estructura de iteración interrumpe la ejecución de las restantes sentencias iteradas y abandona la estructura de control, asignando al contador de programa la dirección de la siguiente sentencia posterior a la llave que cierra el bloque de sentencias de la estructura de control.

Por ejemplo, podemos hacer un programa que solicite al usuario que vaya introduciendo números por teclado hasta que la entrada sea un número par. En ese momento el programa abandonará la solicitud de datos e indicará cuántos enteros ha introducido el usuario hasta introducir uno que sea par. El código podría quedar así:

```
#include <stdio.h>
void main(void)
{
      short i, num;
      for(i = 1 ; ; i++)
      {
            printf("Introduzca un entero ... ");
            scanf("%hd",&num);
            if(num % 2 == 0) break;
```

```
        }
        printf("Ha introducido el entero par %hd",num);
        printf(" después de %hd impares. ",i - 1);
}
```

Se habrán introducido tantos enteros como indique la variable *i*. De ellos, todos menos el último habrán sido impares. Desde luego, el código haría lo mismo si la expresión que condiciona al ***break*** se hubiese colocado en el segundo espacio que ofrece la sintaxis del ***for*** (allí donde se colocan las condiciones de permanencia) y se hubiese cambiado en algo el código; pero no resulta intuitivamente tan sencillo. Si comparamos la estructura **for** vista arriba con otra similar, en la que la condición de salto queda recogida en el for tendremos:

```
for(i = 1 ; ; i++)
{
        printf("entero: ");
        scanf("%hd",&num);
        if(num % 2 == 0) break;
}
```

```
for(i = 1 ; num%2 == 0 ; i++)
{
        printf("entero: ");
        scanf("%hd",&num);
}
```

Aparentemente ambos códigos hacen lo mismo. Pero si comparamos sus flujogramas recogidos en la figura 4.10. veremos que se ha introducido una diferencia no pequeña.

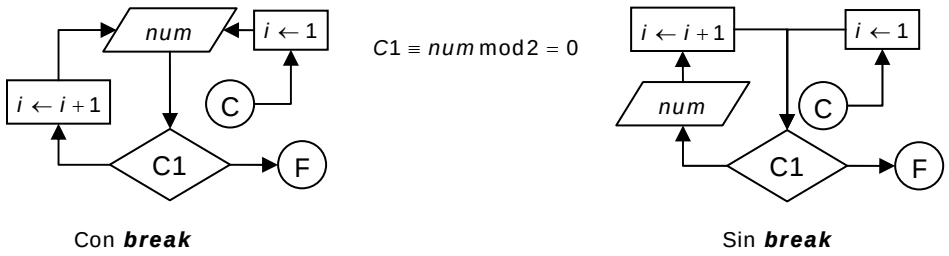

Figura 4.10.: Salto con o sin ***break***.

Habitualmente esta sentencia ***break*** siempre podrá evitarse con un diseño distinto de la estructura de control. Según en qué ocasiones, el código adquiere mayor claridad si se utiliza el ***break***. El uso de la sentencia de salto ***break*** es práctica habitual en la programación estructurada como es el caso del paradigma del lenguaje C.

Con la sentencia ***break*** es posible definir estructuras de control sin condición de permanencia, o con una condición que es siempre verdadera. Por ejemplo:

```
long int suma = 0, num;
do
{
      printf("Introduzca un nuevo sumando ... ");
      scanf("%ld",&num);
      if(num % 2 != 0) break;
      suma += num;
}while(1);
printf("La suma es ... %ld", suma);
```

En esta estructura, la condición de permanencia es verdadera siempre, por definición, puesto que es un literal diferente de cero. Pero hay una sentencia ***break*** condicionada dentro del bloque iterado. El código irá guardando en la variable *suma* la suma acumulada de todos los valores introducidos por teclado mientras que esos valores sean enteros pares. En cuanto se introduzca un entero impar se abandona la estructura de iteración y se muestra el valor sumado.

El diagrama de flujo de este último ejemplo queda recogido en la figura 4.11.

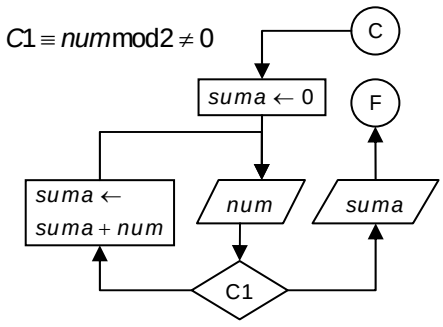

Figura 4.11.: Otro ejemplo de salto con ***break***.

También se pueden tener estructuras ***for*** sin ninguna expresión recogida entre sus paréntesis. Por ejemplo:

```
for( ; ; )
{
      ch = getchar();
```

```
        printf("Esto es un bucle infinito\n");
        if(ch == 'a') break;
}
```

que repetirá la ejecución del código hasta que se pulse la tecla 'a', y que ocasionará la ejecución del ***break***.

Una sentencia ***continue*** dentro de un bloque de instrucciones de una estructura de iteración interrumpe la ejecución de las restantes sentencias iteradas y vuelve al inicio de las sentencias de la estructura de control, si es que la condición de permanencia así lo permite.

Veamos, por ejemplo, el programa antes presentado que muestra por pantalla los 100 primeros enteros pares positivos. Otro modo de resolver ese programa podría ser el siguiente:

```
#include <stdio.h>
void main(void)
{
        short i;
        for(i = 1 ;i <= 100 ; i++)
        {
                if(i % 2) continue;
                printf("%4hd\t",i);
        }
}
```

Cuando el resto de la división entera entre el contador *i* y el número 2 es distinto de cero, entonces el número es impar y se solicita que se ejecute la sentencia ***continue***. Entonces se abandona la ejecución del resto de las sentencias del bloque iterado y, si la condición de

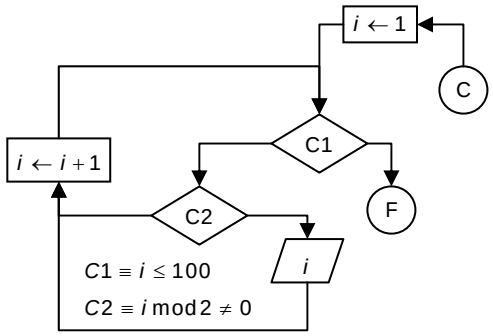

Figura 4.12.: Salto con ***continue***.

permanencia en la iteración así lo permite, vuelve a comenzar la iteración por su primera sentencia del bloque iterado. El diagrama de flujo del código escrito queda recogido en la figura 4.12.

Palabra reservada ***goto***.

La palabra reservada de C *goto* no debe ser empleada.

La sentencia de salto ***goto*** no respeta las reglas de la programación estructurada. Todo código que se resuelve empleando una sentencia ***goto*** puede encontrar una solución mejor con una estructura de iteración.

Si alguna vez ha programado con esta palabra, la recomendación es que se olvide de ella. Si nunca lo ha hecho, la recomendación es que la ignore.

Y, eso sí: hay que acordarse de que esa palabra es clave en C: no se puede generar un identificador con esa cadena de letras.

Variables de control de iteraciones.

Hasta el momento, hemos hecho siempre uso de las variables para poder almacenar valores concretos. Pero pueden tener otros usos. Por ejemplo, podemos hacer uso de ellas como chivatos o centinelas: su información no es el valor concreto numérico que puedan tener, sino la de una situación del proceso.

Como vamos viendo en los distintos ejemplos que se presentan, el diseño de bucles es tema delicado: acertar en la forma de resolver un problema nos permitirá llevar solución a muy diversos problemas. Pero, como estamos viendo, es importante acertar en un correcto control del bucle: decidir bien cuando se ejecuta y cuando se abandona.

Existen dos formas habituales de controlar un bucle o iteración:

1. **Control mediante variable contador**. En ellos una variable se encarga de contar el número de veces que se ejecuta el cuerpo del bucle. Esos contadores requieren una inicialización previa, externa al bucle, y una actualización en cada iteración para llegar así finalmente a un valor que haga falsa la condición de permanencia. Esa actualización suele hacerse al principio o al final de las sentencias iteradas. Hay que garantizar, cuando se diseña una iteración, que se llega a una situación de salida.
2. **Control por suceso**. Este tipo de control acepta, a su vez, diferentes modalidades:
   2.1. **Consulta explícita**: El programa interroga al usuario si desea continuar la ejecución del bucle. La contestación del usuario normalmente se almacena en una variable tipo ***char*** o ***int***. Es el usuario, con su contestación, quien decida la salida de la iteración.
   2.2. **Centinelas**: la iteración se termina cuando la variable de control toma un valor determinado. Este tipo de control es usado habitualmente para introducir datos. Cuando el usuario introduce el valor que el programador ha considerado como valor de fin de iteración, entonces, efectivamente, se termina esa entrada de datos. Así lo hemos visto en el ejemplo de introducción de números, en el programa del cálculo de la media, en el que hemos considerado que la introducción del valor cero era entendido como final de la introducción de datos.
   2.3. **Banderas**: Es similar al centinela, pero utilizando una variable lógica que toma un valor u otro en función de determinadas condiciones que se evalúan durante la ejecución del bucle. Así se puede ver, por ejemplo, en el ejercicio 7 planteado al final de capítulo: la variable que hemos llamado *chivato* es una variable bandera.

Normalmente la bandera siempre se puede sustituir por un centinela, pero se emplea en ocasiones donde la condición de terminación resulta compleja y depende de varios factores que se determinan en diferentes puntos del bucle.

Recapitulación.

Hemos presentado las estructuras de control posibles en el lenguaje C. Las estructuras condicionales de bifurcación abierta o cerrada, condicionales anidadas, operador interrogante, dos puntos, y sentencia ***switch***. También hemos visto las posibles iteraciones creadas con las estructuras ***for***, ***while*** y ***do – while***, y las modificaciones a las estructuras que podemos introducir gracias a las palabras reservadas ***break*** y ***continue***.

El objetivo final de este tema es aprender unas herramientas de enorme utilidad en la programación. Conviene ahora ejercitarse en ellas, resolviendo ahora los diferentes ejercicios propuestos.

Ejercicios.

En todos los ejercicios que planteamos a continuación quizá será conveniente que antes de abordar la implementación se intente diseñar un algoritmo en pseudocódigo o mediante un diagrama de flujo. Si en algún caso el problema planteado supone especial dificultad quedará recogido ese flujograma en estas páginas. En bastantes casos, puede consultarse éste en las páginas del manual “Fundamentos de informática. Codificación y algoritmia”.

20. ***Calcular la suma de los pares positivos menores o igual a 200.***

```
#include <stdio.h>
void main(void)
{
      long suma = 0;
      short i;
      for(i = 2 ; i <= 200 ; i += 2)
            suma += i;
      printf("esta suma es ... %ld.\n",suma);
}
```

21. ***Hacer un programa que calcule la media de todos los valores que introduzca el usuario por consola. El programa debe dejar de solicitar valores cuando el usuario introduzca el valor 0.***

```
#include <stdio.h>
void main(void)
{
      long suma;
      short i, num;
      for(i = 0, suma = 0 ; ; i++)
      {
            printf("Introduzca número ... ");
            scanf("%hd",&num);
            suma += num;
            if(num == 0) break;
      }
      if(i == 0) printf("No se han introducido enteros.");
      else printf("La media es ... %.2f.",(float)suma / i);
}
```

En la estructura ***for*** se inicializan las variables *i* y *suma*. Cada vez que se introduce un nuevo entero se suma al acumulado de todas las sumas de los enteros anteriores, en la variable *suma*. La variable *i* lleva la cuenta de cuántos enteros se han introducido; dato necesario para calcular, cuando se termine de introducir enteros, el valor de la media.

22. ***Mostrar por pantalla los números perfectos menores de 10000. Se entiende por número perfecto aquel que es igual a la suma de sus divisores. Por ejemplo, 6 = 1 + 2 + 3 que son, efectivamente, sus divisores.***

```
#include <stdio.h>
void main(void)
{
      short suma;
      short i, num;
      for(num = 2 ; num < 10000 ; num++)
      {
            for(i = 1, suma = 0 ; i <= num / 2 ; i++)
                  if(num % i == 0) suma += i;
            if(num == suma)
                  printf("En entero %hd es perfecto.\n",num);
      }
}
```

La variable *num* recorre todos los enteros entre 2 y 10000 en busca de aquellos que sean perfectos. Esa búsqueda se realiza con el ***for*** más externo.

Para cada valor distinto de *num*, la variable *suma*, que se inicializa a cero cada vez que se comienza de nuevo a ejecutar la estructura ***for*** anidada, guarda la suma de sus divisores. Eso se realiza en el ***for*** anidado.

Después del cálculo de cada suma, si su valor es el mismo que el entero inicial, entonces ese número será perfecto (esa es su definición) y así se mostrará por pantalla.

La búsqueda de divisores se hace desde el 1 (que siempre interviene) hasta la mitad de *num*: ninguno de los divisores de *num* puede ser mayor que su mitad. Desde luego, se podría haber inicializado la variable *suma* al valor 1, y comenzar a buscar los divisores a partir del 2, porque, efectivamente, el entero 1 es divisor de todos los enteros.

23. ***Solicitar del usuario cuatro números y mostrarlos por pantalla ordenados de menor a mayor.***

```
#include <stdio.h>
void main(void)
{
      unsigned short int a0,a1,a2,a3;
      printf("Introduzca cuatro enteros ... \n\n");
      printf("Primer entero ... ");
      scanf("%hu",&a0);
      printf("Segundo entero ... ");
      scanf("%hu",&a1);
      printf("Tercer entero ... ");
      scanf("%hu",&a2);
      printf("Cuarto entero ... ");
      scanf("%hu",&a3);
      if(a0 > a1)
      {
            a0 ^= a1;
            a1 ^= a0;
            a0 ^= a1;
      }
      if(a0 > a2)
      {
            a0 ^= a2;
            a2 ^= a0;
            a0 ^= a2;
      }
      if(a0 > a3)
      {
            a0 ^= a3;
            a3 ^= a0;
            a0 ^= a3;
      }
      if(a1 > a2)
      {
            a1 ^= a2;
            a2 ^= a1;
            a1 ^= a2;
      }
      if(a1 > a3)
      {
            a1 ^= a3;
            a3 ^= a1;
            a1 ^= a3;
```

```
	}
	if(a2 > a3)
	{
		a2 ^= a3;
		a3 ^= a2;
		a2 ^= a3;
	}
	printf("\nOrdenados... \n");
	printf("%hu <= %hu <= %hu <= %hu.", a0, a1, a2, a3);
}
```

Donde el código que se ejecuta en cada estructura ***if*** intercambia los valores de las dos variables, como ya vimos en un ejemplo de un tema anterior.

24. Mostrar por pantalla todos los caracteres ASCII.

(Antes de ejecutar el código escrito, termine de leer todo el texto recogido en este problema.)

```
#include <stdio.h>
void main(void)
{
	unsigned char a;
	for(a = 0 ; a <= 255 ; a++)
		printf("%3c - %hX - %hd\n",a,a,a);
}
```

Va mostrando todos los caracteres, uno por uno, y su código en hexadecimal y en decimal.

Si se desea ver la aparición de todos los caracteres, se puede programar para que se pulse una tecla cada vez que queramos que salga el siguiente carácter por pantalla. Simplemente habría que modificar la estructura ***for***, de la siguiente forma:

```
	for(a = 0 ; a <= 255 ; a++)
	{
		printf("%3c - %hX - %hd\n",a,a,a);
		getchar();
	}
```

Advertencia importante: De forma intencionada hemos dejado el código de este problema con un error grave. Aparentemente todo está

bien. De hecho no existe error sintáctico alguno. El error se verá en tiempo de ejecución, cuando se compruebe que se ha caído en un bucle infinito.

Para comprobarlo basta observar la condición de permanencia en la iteración gobernada por la estructura ***for***, mediante la variable que hemos llamado *a*: *a <= 255*. Si se tiene en cuenta que la variable *a* es de tipo ***unsigned char***, no es posible que la condición indicada llegue a ser falsa, puesto que, en el caso de que *a* alcance el valor 255, al incrementar en 1 su valor, incurrimos en overflow, y la variable cae en el valor cero: $(255)_{10} = (11111111)_2$; si solo tenemos ocho dígitos, entonces... $(11111111 + 1)_2 = (00000000)_2$, pues no existe el bit noveno, donde debería haber quedado codificado un dígito 1.

Sirva este ejemplo para advertir de que a veces puede aparecer un bucle infinito de la manera más insospechada. La tarea de programar no está exenta de sustos e imprevistos que a veces obligan a dedicar un tiempo no pequeño a buscar la causa de un error.

Un modo correcto de codificar este bucle sería, por ejemplo:

```
for(a = 0 ; a < 255 ; a++)
      printf("%3c - %hX - %hd\n",a,a,a);
printf("%3c - %hX - %hd\n",a,a,a);
```

Y así, cuando llegue al valor máximo codificable en la variable *a*, abandona el bucle e imprime, ya fuera de la iteración, una última línea de código, con el valor último.

25. ***Solicitar al usuario una valor entero por teclado y mostrar entonces, por pantalla, el código binario en la forma en que se guarda el número en la memoria.***

```
#include <stdio.h>
void main(void)
{
```

```
	signed long a;
	unsigned long Test;
	char opcion;
	do
	{
		Test = 0x80000000;
		printf("\n\nIndique el entero ... ");
		scanf("%ld", &a);
		while(Test)
		{
			Test & a ? printf("1") : printf("0");
			Test >>= 1;
		}
		printf("\n¿Desea introducir otro entero? ... ");
		do
			opcion = getchar();
		while (opcion != 's' && opcion != 'n');
	}while(opcion == 's');
}
```

La variable *Test* se inicializa al valor hexadecimal 80000000, es decir, con un 1 en el bit más significativo y en cero en el resto. Posteriormente sobre esta variable se va operando un desplazamiento a derecha de un bit cada vez. Así, siempre tenemos localizado un único bit en la codificación de la variable *Test*, hasta que este bit se pierda por la parte derecha en un último desplazamiento.

Antes de cada desplazamiento, se realiza la operación and a nivel de bit entre la variable *Test*, de la que conocemos donde está su único bit, y la variable de la que queremos conocer su código binario.

En el principio, tendremos así las dos variables:

```
Test         1000 0000 0000 0000 0000 0000 0000 0000
a            xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
Test & a     x000 0000 0000 0000 0000 0000 0000 0000
```

Tenemos certeza de que la operación and dejará un cero en todos los bits (menos el más significativo) de *Test & a*, porque *Test* tiene todos esos bits a cero. Todos… menos el primero. Entonces la operación dará un valor distinto de cero únicamente si en ese bit se encuentra un 1 en la variable *a*. Sino, el resultado de la operación será cero.

Al desplazar ahora *Test* un bit a la derecha tendremos la misma situación que antes, pero ahora el bit de *a* testeado será el segundo por la derecha.

```
Test         0100 0000 0000 0000 0000 0000 0000 0000
a            xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
Test & a     0x00 0000 0000 0000 0000 0000 0000 0000
```

Y así sucesivamente, imprimiremos un 1 cuando la operación and sea diferente de cero, e imprimiremos un 0 cuando la operación and dé un valor igual a cero.

El programa que hemos presentado permite al usuario ver el código de tantos números como quiera introducir por teclado: hay una estructura ***do–while*** que anida todo el proceso.

26. ***Escribir un programa que solicite al usuario un entero positivo e indique si ese número introducido es primo o compuesto.***

```
#include <stdio.h>
#include <math.h>
void main(void)
{
      unsigned long int numero, raiz;
      unsigned long int div;
      char chivato;
      printf("Dame el numero que vamos a testear ... ");
      scanf("%lu", &numero);
      chivato = 0;
      raiz = sqrt(numero);
      for(div = 2 ; div <= raiz ; div++)
      {
            if(numero % div == 0)
            {
                  chivato = 1;
                  break;
            }
      }
      if(chivato == 1)
            printf("El numero %lu es compuesto",numero);
      else  printf("El numero %lu es primo",numero);
}
```

Antes de explicar brevemente el algoritmo, conviene hacer una digresión sencilla matemática: todo entero verifica que, si tiene divisores distintos del 1 y del mismo número (es decir, si es compuesto), al menos uno de esos divisores es menor que su raíz cuadrada. Eso es sencillo de demostrar por reducción al absurdo: supongamos que tenemos un entero n y que tiene dos factores distintos de 1 y de n; por ejemplo, a y b, es decir, $n = a \times b$. Supongamos que ambos factores son mayores (estrictamente mayores) que la raíz cuadrada de n. Entonces tendremos:

$$n = a \times b < \sqrt{n} \times \sqrt{n} = n \Rightarrow n < n$$

En tal caso tendríamos el absurdo de que n es estrictamente menor que n. Por lo tanto, para saber si un entero n es primo o compuesto, basta buscar divisores entre 2 y n. Si en ese rango no los encontramos, entonces podemos concluir que el entero es primo.

En este programa vamos probando con todo los posibles enteros que dividen al número estudiado, que serán todos los comprendidos entre el 2 y la raíz cuadrada de ese número. En cuanto se encuentra un valor que divide al entero introducido, entonces ya está claro que ese número es compuesto y no es menester seguir buscando otros posibles divisores. Por eso se ejecuta la sentencia ***break***.

Al terminar la ejecución de la estructura ***for*** no sabremos si hemos salido de ella gracias a que hemos encontrado un divisor y nos ha expulsado la sentencia ***break***, o porque hemos terminado de testear entre todos los posibles candidatos a divisores menores que la raíz cuadrada y no hemos encontrado ninguno porque el entero introducido es primo. Por ello hemos usado la variable *chivato*, que se pone a 1, antes de la sentencia ***break***, en caso de que hayamos encontrado un divisor.

Otro modo de saber si hemos salido del bucle por haber encontrado un divisor o por haber terminado el recorrido de la variable de control de la

estructura ***for*** es verificar el valor de esa variable contador. Si la variable *div* es mayor que *raiz* entonces está claro que hemos salido del bucle por haber terminado la búsqueda de posibles divisores y *numero* es primo. Si *div* es menor o igual que *raiz*, entonces está también claro que hemos encontrado un divisor: es otra forma de hacer el programa, sin necesidad de crear la variable *chivato*.

El código, en ese caso, podría quedar de la siguiente manera:

```
#include <stdio.h>
#include <math.h>
void main(void)
{
      unsigned long int n, raiz;
      unsigned long int div;
      printf("Dame el numero que vamos a testear ... ");
      scanf("%lu", &n);
      raiz = sqrt(n);
      for(div = 2 ; div<=raiz ; div++) if(n%div == 0) break;
      printf("%lu es %s", n,n > raiz ? "primo":"compuesto");
}
```

27. ***Escriba un programa que muestre por pantalla un término cualquiera de la serie de Fibonacci.***

La serie de Fibonacci está definida de la siguiente manera: el primer y el segundo elementos son iguales a 1. A partir del tercero, cualquier otro elemento de la serie es igual a la suma de sus dos elementos anteriores.

Escribir el código es muy sencillo una vez se tiene el flujograma. Intente hacer el flujograma, o consulte página 60 del manual "*Fundamentos de Informática. Codificación y Algoritmia.*"

```
#include <stdio.h>

void main(void)
{
      unsigned long fib1 = 1, fib2 = 1, Fib = 1;
      unsigned short n, i = 3;
```

```
      printf("Indique el elemento que se debe mostrar ... ");
      scanf("%hu",&n);

      while(i <= n)
      {
            Fib = fib1 + fib2;
            fib1 = fib2;
            fib2 = Fib;
            i++;
      }
      printf("El término %hu de la serie es %lu.\n",n,Fib);
}
```

28. ***Escriba un programa que resuelva una ecuación de segundo grado. Tendrá como entrada los coeficientes*** *a*, *b* ***y*** *c* ***de la ecuación y ofrecerá como resultado las dos soluciones reales.***

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

void main(void)
{
      float a, b, c;
      double r;
      // introducción de parámetros...
      printf("Introduzca los coeficientes...\n\n");
      printf("a --> "); scanf("%f",&a);
      printf("b --> "); scanf("%f",&b);
      printf("c --> "); scanf("%f",&c);
      // Ecuación de primer grado...
      if(a == 0)
      {     // No hay ecuación ...
            if(b == 0)  printf("No hay ecuación.\n");
            else // Sí hay ecuación de primer grado
            {
                  printf("Ecuación de primer grado.\n");
                  printf("Tiene una única solución.\n");
                  printf("x1 --> %lf\n", -c / b);
            }
      }
      // Ecuación de segundo grado. Soluciones imaginarias.
      else if ((r = b * b - 4 * a * c) < 0)
        {
            printf("Ecuación sin soluciones reales.\n");
            r = sqrt(-r);
```

```
            printf("x1: %lf + %lf * i\n",-b/(2*a), r/(2*a));
            printf("x2: %lf + %lf * i\n",-b/(2*a),-r/(2*a));
      }
      // Ecuación de segundo grado. Soluciones reales.
      else
      {
            printf("Las soluciones son:\n");
            r = sqrt(r);
            printf("\tx1 --> %lf\n", (-b + r) / (2 * a));
            printf("\tx2 --> %lf\n", (-b - r) / (2 * a));
      }
}
```

29. ***Escriba un programa que muestre todos los divisores de un entero que se recibe como entrada del algoritmo.***

```
#include <stdio.h>

void main(void)
{
      long int numero;
      long int div;

      printf("Número --> ");
      scanf("%ld",&numero);

      printf("\n\nLos divisores de %ld son:\n\t", numero);
      printf("1, ");

      for(div = 2 ; div <= numero / 2 ; div++)
            if(numero % div == 0)
                  printf("%ld, ",div);

      printf("%ld.",numero);
}
```

El código ofrece, por ejemplo, la siguiente salida por pantalla:

Número --> 456
Los divisores de 456 son:
1, 2, 3, 4, 6, 8, 12, 19, 24, 38, 57, 76, 114, 152, 228, 456.

30. ***Escriba un programa que calcule el número π, sabiendo que este número verifica la siguiente relación:***

$$\frac{\pi^2}{6} = \sum_{k=1}^{\infty} \frac{1}{k^2}.$$

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

#define LIMITE 10000

void main(void)
{
      double PI = 0;

      for(int i = 1 ; i < LIMITE ; i++)
            PI += 1.0 / (i * i);
      PI *= 6;
      PI = sqrt(PI);
      printf("El valor de PI es ... %lf.",PI);
}
```

Que ofrece la siguiente salida por pantalla:

El valor de PI es ... 3.141497.

El programa va haciendo, en la iteración gobernada por la estructura ***for***, el sumatorio de los inversos de los cuadrados. El numerador se debe poner como 1.0 para que el resultado del cociente no sea un entero igual a cero sino un valor ***double***.

31. ***Escriba un programa que calcule el número π, sabiendo que este número verifica la siguiente relación:***

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2 \cdot k + 1}$$

```
#include <stdio.h>
#define LIMITE 100000

void main(void)
{
      double PI = 0;

      for(int i = 1 , e = 4 ; i < LIMITE ; i += 2 , e = -e)
            PI += e / (double)i;

      printf("El valor de PI es ... %lf.",PI);
}
```

Que ofrece la siguiente salida por pantalla:

El valor de PI es ... 3.141573.

La variable PI se inicializa a cero. En cada iteración irá almacenando la suma de todos los valores calculados. En lugar de calcular $\pi/4$, calculamos directamente el valor de π: por eso el numerador (variable que hemos llamado *e*) no varía entre -1 y +1, sino entre -4 y +4. El valor de la variable *i* aumenta de dos en dos, y va tomando los diferentes valores del denominador $2 \cdot k + 1$. Es necesario forzar el tipo de la variable *i* a double, para que el resultado de la operación cociente no sea un entero, que a partir de *i* igual a cero daría como resultado el valor cero.

32. ***Escriba un programa que calcule el número π, sabiendo que este número verifica la siguiente relación:***

$$\frac{\pi}{2} = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdot \ldots$$

De nuevo el cálculo del valor del número pi. Estos ejercicios son muy sencillos de buscar (Internet está llena de definiciones de propiedades del número pi) y siempre es fácil comprobar si hemos realizado un buen código: basta ejecutarlo y comprobar si sale el famoso 3.14.

En esta ocasión, el código podría tomar la siguiente forma:

```
#include <stdio.h>
#define LIMITE 100000
void main(void)
{
      double PI = 2;
      for(int num = 2 , den = 1, i = 1 ;
       i < LIMITE ; i++ , i % 2 ? num+=2 : den+=2)
      PI *= num / (double)den;
      printf("El valor de PI es ... %lf.",PI);
}
```

Por problemas de espacio se ha tenido que mostrar la estructura for truncada en dos líneas. Esta forma de escribir es perfectamente válida en C. El compilador considera lo mismo un espacio en blanco que tres líneas blancas. Para el compilador estos dos códigos significan lo mismo:

```
short int a = 0 ,  b = 1, c = 2;
double x, y, z;
short int
      a = 0,
      b = 1,
      c = 2,
double
      x,
      y, z;
```

33. ***Cinco marineros llegan, tras un naufragio, a una isla desierta con un gran número de cocoteros y un pequeño mono. Dedican el primer día a recolectar cocos, pero ejecutan con tanto afán este trabajo que acaban agotados, por lo que deciden repartirse los cocos al día siguiente.***

Durante la noche un marinero se despierta y, desconfiando de sus compañeros, decide tomar su parte. Para ello, divide el montón de cocos en cinco partes iguales, sobrándole un coco, que regala al mono. Una vez calculada su parte la esconde y se vuelve a acostar.

Un poco más tarde otro marinero también se despierta y vuelve a repetir la operación, sobrándole también un coco que regala al mono. En el resto de la noche sucede lo mismo con los otros

tres marineros.

Al levantarse por la mañana procedieron a repartirse los cocos que quedaban entre ellos cinco, no sobrando ahora ninguno.

¿Cuántos cocos habían recogido inicialmente? Mostrar todas las soluciones posibles menores de 1 millón de cocos.

Este programa es sencillo de implementar. Pero hay que saber resolver el problema. Es un ejemplo de cómo saber un lenguaje no lo es todo en programación; más bien podríamos decir que saber un lenguaje es lo de menos: lo importante es saber qué decir.

Este algoritmo está ya explicado en el otro manual, ya tantas veces referenciado en éste. Aquí dejamos sólo el código resultante.

```
#include <stdio.h>

void main(void)
{
      unsigned long N = 6, n;
      unsigned long soluciones = 0;

      printf("Valores posibles de N ...\n\n");

      while(N < 4000000)
      {
            unsigned short int i;
            N += 5;
            n = N;
            for(i = 0 ; i < 5 ; i++)
            {
                  if((n - 1) % 5) break;
                  n = 4 * (n - 1) / 5;
            }
            if(i == 5 && !(n % 5))
            {
                  printf("(%4lu) %-8lu",++soluciones, N);
                  if(!(soluciones % 5)) printf("\n");
            }
      }
}
```

34. ***El calendario juliano (debido a julio Cesar) consideraba que el año duraba 365.25 días, por lo que se estableció que los años tendrían una duración de 365 días y cada cuatro años se añadiese un día más (año bisiesto).***

Sin embargo se comprobó que en realidad el año tiene 365.2422 días, lo que implica que el calendario juliano llevase un desfase de unos once minutos. Este error es relativamente pequeño, pero, con el transcurrir del tiempo, el error acumulado puede ser importante.

El papa Gregorio XII, en 1582, propuso reformar el calendario juliano para evitar los errores arrastrados de años anteriores. Los acuerdos tomados entonces, que son por los que nos aún nos regimos, fueron los siguientes:

- ***Para suprimir el error acumulado por el calendario juliano, se suprimieron diez días. De tal manera que el día siguiente al 4 de octubre de 1582 fuel el día 15 del mismo mes.***
- ***La duración de los años sería de 365 días o 366 en caso de ser bisiesto.***
- ***Serán bisiestos todos los años que sean múltiplos de 4, salvo los que finalizan en 00, que sólo lo serán cuando también sean múltiplos de 400, por ejemplo 1800 no fue bisiesto y el 2000 sí.***
- ***Con esta reforma el desfase existente entre el año civil y el año real se reduce a menos de treinta segundos anuales.***

Definir un algoritmo que solicite al usuario una fecha introducida mediante tres datos: día, mes y año; ese programa debe validar la fecha: es decir comprobar que la fecha es correcta cumpliendo las siguientes reglas:

- ***El año debe ser mayor que 0.***

- ***El mes debe ser un número entre uno y doce.***
- ***El día debe estar entre 1 y 30, 31,28 ó 29 dependiendo el mes de que se trate y si el año es bisiesto o no.***

Se deja propuesto. En otro lugar está recogido el flujograma del algoritmo.

35. ***Calcule el valor del número e. sabiendo que verifica la siguiente relación:***

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

```
#include <stdio.h>
void main(void)
{
      double p = 1, e = 1;
      for(short n = 1 ; n < 100 ; n++)
      {
            p *= 1.0 / n;
            e += p;
      }
      printf("El numero e es ... %20.17lf.",e);
}
```

36. ***Juego de las 15 cerillas: "Participan dos jugadores. Inicialmente se colocan 15 cerillas sobre una mesa y cada uno de los dos jugadores toma, alternativamente 1, 2 o 3 cerillas pierde el jugador que toma la última cerilla".***

Buscar el algoritmo ganador para este juego, generalizando: inicialmente hay *N* ***cerillas y cada vez se puede tomar hasta un máximo de*** *k* ***cerillas. Los valores*** *N* ***y*** *k* ***son introducidos por***

teclado y decididos por un jugador (que será el jugador perdedor), el otro jugador (que será el ordenador, y que siempre debe ganar) decide quien empieza el juego.

```
#include <stdio.h>

void main(void)
{
// Con cuántas cerillas se va a jugar.
      unsigned short cerillas;
// Cuántas cerillas se pueden quitar cada vez.
      unsigned short quitar;
// Cerillas que quita el ganador y el perdedor.
      unsigned short qp, qg;
      char ganador;

      do
      {
            printf("\n\n\nCon cuántas cerillas
                    se va a jugar ... ");
            scanf("%hu",&cerillas);
            if(cerillas == 0) break;
            do
            {
                  printf("\Cuántas cerillas pueden
                    quitarse de una vez... ");
                  scanf("%hu",&quitar);
                  if(quitar >= cerillas)
                        printf("No pueden quitarse tantas
                                cerillas.\n");
            }while(quitar >= cerillas);

            qg = (cerillas - 1) % (quitar + 1);
// MOSTRAR CERILLAS ...
            printf("\n");
            for(short i = 1 ; i <= cerillas ; i++)
            {
                  printf(" |");
                  if(!(i % 30))printf("\n\n");
            }
            printf("\n");
// Fin de MOSTRAR CERILLAS
            if(qg)
            {
                  printf("\nComienza la máquina...");
                  printf("\nMáquina Retira %hu cerillas...
                           \n", qg);
```

```
                cerillas -= qg;
// MOSTRAR CERILLAS ...
                printf("\n");
                for(short i = 1 ; i <= cerillas ; i++)
                {
                      printf(" |");
                      if(!(i % 30))printf("\n\n");
                }
                printf("\n");
          }
// FIN DE MOSTRAR CERILLAS
          else printf("\nComienza el jugador...");

          while(cerillas != 1)
          {
                do
                {
                      printf("\nCerillas que retira el
                              jugador ... ");
                      scanf("%hu",&qp);
                      if(qp > quitar)
                            printf("\nNo puede quitar mas
                                    de %hu.\n",quitar);
                }while(qp > quitar);
                cerillas -= qp;
// MOSTRAR CERILLAS ...
                printf("\n");
                for(short i = 1 ; i <= cerillas ; i++)
                {
                      printf(" |");
                      if(!(i % 30)) printf("\n\n");
                }
                printf("\n");
// Fin de MOSTRAR CERILLAS
                if(cerillas == 1)
                {
                      ganador = 'j';
                      break;
                }
                qg = quitar - qp + 1;
                printf("\nLa máquina retira %hu
                        cerillas.\n",qg);
                cerillas -= qg;
// MOSTRAR CERILLAS ...
                printf("\n");
                for(short i = 1 ; i <= cerillas ; i++)
                {
                      printf(" |");
                      if(!(i % 30))printf("\n\n");
                }
                printf("\n");
```

```
// Fin de MOSTRAR CERILLAS
                  if(cerillas == 1) ganador = 'm';
            }

            if(ganador == 'j')
                  printf("\nHa ganado el jugador...");
            else if(ganador == 'm')
                  printf("\nHe ganado yo, la máquina...");
      }while(cerillas);
}
```

El programa ha quedado un poco más largo que los anteriores. Se ha dedicado un poco de código a la presentación en el momento de la ejecución. Más adelante, cuando se haya visto el modo de definir funciones, el código quedará visiblemente reducido. Por ejemplo, en cuatro ocasiones hemos repetido el mismo código destinado a visualizar por pantalla las cerillas que quedan por retirar.

37. ***El número áureo (Φ) verifica muchas curiosas propiedades. Por ejemplo:***

$\Phi^2 = \Phi + 1$ $\quad$ $\Phi - 1 = 1/\Phi$ $\quad$ $\Phi^3 = (\Phi + 1)/(\Phi - 1)$

$\Phi = 1 + 1/\Phi$ $\quad$ $\Phi = \sqrt{1 + \Phi}$ $\quad$ ***y otras...***

La penúltima expresión presentada ($\Phi = 1 + 1/\Phi$) muestra un camino curioso para el cálculo del número áureo:

Si $\Phi = 1 + \frac{1}{\Phi} \Rightarrow \Phi = 1 + \frac{1}{1 + \frac{1}{\Phi}} \Rightarrow \Phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{\Phi}}} \ldots$

Y, entonces un modo de calcular el número áureo es: inicializar Φ al valor 1, e ir afinando en el cálculo del valor del número áureo a base de repetir muchas veces que $\Phi = 1 + 1/\Phi$:

$$\Phi \simeq 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots \cfrac{1}{1 + \cfrac{1}{1}}}}}$$

Escriba un programa para calcular el número áureo mediante este procedimiento haciendo, por ejemplo, la sustitución $\Phi = 1 + 1/\Phi$ ***mil veces. Mostrar luego el resultado por pantalla.***

```
#include <stdio.h>
#define LIMITE 1000

void main(void)
{
      double au = 1;

      for(int i = 0 ; i < LIMITE ; i++)
            au = 1 + 1 / au;

      printf("El número áureo es ..... %lf.\n",au);
      printf("Áureo al cuadrado es ... %lf.\n",au * au);
}
```

No se ha hecho otra cosa que considerar lo que sugiere el enunciado: inicializar el número áureo a 1, y repetir mil veces la iteración $\Phi = 1 + 1/\Phi$. Al final mostramos también el cuadrado del número áureo: es un modo rápido de verificar que, efectivamente, el número hallado es el número buscado: el número áureo verifica que su cuadrado es igual al número incrementado en uno.

La salida que ofrece este código por pantalla es la siguiente:

El número áureo es ..... 1.618034.
Áureo al cuadrado es ... 2.618034.

38. ***Siguiendo con el mismo enunciado, también podemos plantearnos calcular el número áureo a partir de la relación*** $\Phi = \sqrt{1 + \Phi}$ ***. De nuevo***

tenemos:

$$\Phi = \sqrt{1+\Phi} = \sqrt{1+\sqrt{1+\Phi}} = \sqrt{1+\sqrt{1+\sqrt{1+\Phi}}} = \ldots = \sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\ldots\sqrt{1+\Phi}}}}}}$$

```
#include <stdio.h>
#include <math.h>
#define LIMITE 100000

void main(void)
{
      double au = 1;

      for(int i = 0 ; i < LIMITE ; i++)
            au = sqrt(1 + au);

      printf("El número áureo es ..... %lf.\n",au);
      printf("Áureo al cuadrado es ... %lf.\n",au * au);
}
```

Este programa ofrece una salida idéntica a la anterior. Y el código es casi igual que el anterior: únicamente cambia la definición de la iteración.

39. Muestre por pantalla todos los enteros primos comprendidos entre dos enteros introducidos por teclado.

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

void main(void)
{
      long a, b, div;
      printf("Límite inferior ... ");
      scanf("%ld",&a);
      printf("Límite superior ... ");
      scanf("%ld",&b);

      printf("Los primos entre %ld y %ld son ...\n\t", a, b);
      for(long num = a, primos = 0 ; num <= b ; num++)
      {
```

```
            for(div = 2 ; div < sqrt(num) ; div++)
                  if(num % div == 0) break;
            if(num % div == 0) continue;
            if(primos != 0 && primos % 10 == 0)
                  printf("\n\t");
            primos++;
            printf("%6ld,",num);
      }
}
```

El primer ***for*** recorre todos los enteros comprendidos entre los dos límites introducidos por teclado. El segundo ***for*** averigua si la variable *num* codifica en cada iteración del primer ***for*** un entero primo o compuesto: si al salir del segundo ***for*** se tiene que *num % div* es igual a cero, entonces *num* es compuesto y se ejecuta las sentencia ***continue*** que vuelve a la siguiente iteración del primer ***for***. En caso contrario, el valor de *num* es primo, y entonces sigue adelante con las sentencias del primer ***for***, que están destinadas únicamente a mostrar por pantalla, de forma ordenada, ese entero primo, al igual que habrá mostrado previamente todos los otros valores primos y mostrará los que siga encontrando posteriormente.

La salida por pantalla del programa podría ser la siguiente:

Límite inferior ... 123
Límite superior ... 264
Los primos entre 123 y 264 son ...

127, 131, 137, 139, 149, 151, 157, 163, 167, 173,
179, 181, 191, 193, 197, 199, 211, 223, 227, 229,
233, 239, 241, 251, 257, 263,

Para este último programa se recomienda que se dibuje el diagrama de flujo.