Escuela Politécnica Superior de Alcoy

Ingeniería Técnica en Informática de Gestión

Estructuras de Datos y Algoritmos

Prácticas

Francisco Nevado, Jordi Linares

Índice general

1. El lenguaje de programación C **4**
1.1. Estructura de un programa en C 4
1.2. Un primer programa en C 5
1.3. Compilación 6
1.4. Tipos de datos 6
1.5. Declaración de variables y constantes 6
1.6. Tipos estructurados 8
1.7. Expresiones y operadores 9
1.8. Entrada y salida de datos 11
1.8.1. Salida 11
1.8.2. Entrada 12
1.9. Estructuras de decisión 13
1.9.1. Decisión simple 13
1.9.2. Decisión doble 13
1.9.3. Decisión múltiple 14
1.10. Estructuras de repetición 15
1.11. Contracciones 18
1.12. Funciones 19
1.12.1. Argumentos: llamadas por valor 20
1.13. Punteros 21
1.13.1. Punteros y argumentos de funciones 23
1.13.2. Punteros y vectores 24
1.13.3. Punteros y estructuras 27
1.14. Strings 27
1.15. Keywords 30
1.16. Variables externas y alcance 30
1.17. Manejo de ficheros 31

2. Introducción al IDE de Turbo-C **33**
2.1. Objetivos 33
2.2. ¿Qué es el IDE? 33

2.3. Empezando con el IDE del Turbo-C 34
2.3.1. El Menú 34
2.3.2. Las ventanas del IDE 35
2.3.3. La línea de estado 36
2.4. Trabajo con ficheros 37
2.5. Trabajando con bloques de texto 38
2.6. Saliendo del Turbo-C 39
2.7. La ayuda del IDE 39
2.8. Empezando a programar 40

3. Ejercicios de programación 43
3.1. Estructuras de decisión 43
3.2. Estructuras de repetición 44
3.3. Funciones 47
3.4. Vectores 48
3.5. Punteros y Memoria dinámica 51
3.6. Strings 53
3.7. Manejo de ficheros 54

4. Introducción al diseño modular 57
4.1. Introducción al diseño modular 57
4.2. Listas de importación/exportación 58
4.3. Sintaxis de un módulo 58
4.4. Utilización de módulos en C 58
4.5. Compilación de módulos en Turbo C 59
4.6. Ejemplo de programa dividido en módulos 60

5. Evaluación de expresiones aritméticas 62
5.1. Introducción 62
5.2. Expresiones postfijas 63
5.3. Conversión de notación infija a postfija 65
5.4. Un ejemplo completo 67
5.5. Implementación 72
5.6. Códigos fuente para la sección 5.5 75

6. Estructurando programas 82
6.1. Listas de números enteros 82
6.2. Listas de fichas de datos 84
6.3. Códigos fuente para la sección 1 86
6.4. Códigos fuente para la sección 2 94

7. Utilizando otros recursos para programar **102**

8. Evaluando el coste temporal empírico de un programa **107**
8.1. Medición empírica del coste temporal 107
8.2. Quicksort: otro algoritmo de Partición. 109
8.2.1. Elección del pivote 109
8.2.2. Evaluando algoritmos de ordenación 111
8.3. Códigos fuente para la sección 1 113
8.4. Códigos fuente para la sección 2 115

9. Estudio de tablas de dispersión **121**
9.1. Introducción 121
9.2. Tablas de dispersión 121
9.3. Actividades en el laboratorio 122
9.4. Eligiendo el número de cubetas y la función hash 122
9.5. Encriptando un fichero de texto 125
9.6. Desencriptando un fichero de texto 126
9.7. Librería de tablas hash 127
9.8. Códigos fuente para la Sección 9.4 134
9.9. Códigos fuente para la Sección 9.5 136
9.10. Códigos fuente para la Sección 9.6 140

10. Estudio de Montículos (Heaps) **144**
10.1. Introducción 144
10.2. Montículos (Heaps) 144
10.3. El algoritmo *Heapsort* 145
10.4. Implementación de una cola de prioridad:
Simulación de una CPU 147
10.5. Librería para Montículos (Heap) 151
10.6. Códigos fuente para la sección 10.3 154
10.7. Librería para Colas de Prioridad 156
10.8. Códigos fuente para la sección 10.4 159

11. Aciclidad de Grafos **167**
11.1. Introducción 167
11.2. Grafos acíclicos 167
11.3. Representación de grafos 168
11.4. El algoritmo *acíclico* 170
11.5. Librería para Grafos 174
11.6. Código fuente del programa principal 179

Práctica 1

El lenguaje de programación C

La siguiente práctica no puede considerarse una práctica como tal. Podemos decir que es un pequeño manual de introducción al lenguaje de programación C.

Para ello, se describen todos los aspectos básicos de programación en C junto con pequeños ejemplos que ilustran el uso de los mecanismos del lenguaje.

Esta práctica se complementa con la práctica 3, en la cual se proponen ejercicios de programación con la idea de que al resolverlos, el alumno aprenda a utilizar la sintaxis y los mecanismos del lenguaje C para programar.

Se recomienda realizar una primera lectura de esta práctica, después pasar a realizar los ejercicios propuestos en la práctica siguiente y, cuando durante la resolución de los mismos, el alumno tenga dudas, volver a releer los apartados correspondientes de esta práctica.

Comenzamos ahora a describir las generalidades del lenguaje C.

1.1. Estructura de un programa en C

En todo programa C es necesaria una función principal main. Opcionalmente (aunque también habitualmente) existen otros elementos, de manera que la estructura suele ser la siguiente:

```
[líneas de precompilación]

[declaraciones globales]
[declaración de funciones]
[tipo] main([argumentos]) {
  [declaración de variables]
  [instrucciones]
}
```

1.2. Un primer programa en C

Vamos a editar el siguiente programa en C, el clásico ”Hola, mundo”que será nuestro primer programa C, y cuya única misión es escribir el mensaje Hola, mundo por pantalla.

```c
#include <stdio.h>

main () {
  /* Primer programa en C */
  printf("Hola, mundo\n");
}
```

Sobre este programa ejemplo se pueden comentar ciertas cosas:

- `#include` es una línea (o directiva) de precompilación, y como todas ellas empieza por `#`. En este caso, indica que la librería **stdio.h** va a usarse en el programa actual.

  Los ficheros o librerías que acompañen a la sentencia `#include` deben ir entre comillas () o bien entre <>. Ejemplo:

  ```c
  #include "defines.h"
  #include <defines.h>
  ```

  La diferencia entre estas dos variantes es que el fichero incluido con comillas se busca primero en el directorio de trabajo, mientras que el segundo (con <>) se busca en los directorios de librerías definidos en el entorno de programación.

- La línea `/* Primer programa en C */` es un comentario. En C, los comentarios comienzan con `/*` y acaban con `*/`, pudiendo ocupar varias líneas de código fuente. Para comentarios que ocupen una única línea, se puede escribir // al principio del comentario. C obviará todo lo que venga a continuación, hasta el salto de línea (retorno de carro).

- Todas las instrucciones (como la invocación a `printf` de este programa) deben acabar en `;` necesariamente.

- La secuencia `\n` que se pone al final de la cadena `Hola, mundo` provoca el salto de línea (retorno de carro).

1.3. Compilación

Para estas prácticas se empleará el entorno de desarrollo integrado Turbo-C (IDE, *Integrated Development Environment*). En el boletín de prácticas de Introducción al IDE se detalla tanto el uso de dicho entorno de desarrollo como la manera en la que se deben compilar los programas.

1.4. Tipos de datos

En C tenemos los siguientes tipos de datos básicos:

- **int**, que son los enteros. Son de 16 bits (aunque en muchas máquinas ya se consideran de 32 bits) y su rango está comprendido entre el -32768 y +32767.
- **float** y **double**, que son los reales con simple y doble precisión, respectivamente. Admiten la notación científica. Un número float típicamente es de 32 bits, por lo menos con seis dígitos significativos y una magnitud generalmente entre 10-38 y 10+38. Ejemplos de reales serían 6.3, -0.01, 5.5e3, -8.16E-8.
- **char**, que son caracteres ASCII, como es el caso de '`a`', '`A`', '`\n`', ... Como vemos, los caracteres ASCII se notan entre comillas simples. Ocupan un solo byte.

Aparte de estos tipos, hay un tipo especial llamado **void**, que se puede interpretar como *nada* o como *cualquier cosa*, según la situación en la que se emplee.

Además, C aporta modificadores para los datos de tipo entero. Estos modificadores son **short**, **long** y **unsigned**. Como su propio nombre indica, hacen que el rango entero sea más corto (menos bits para representarlo), más largo (más bits) o que sea sólo positivo, respectivamente.

La longitud de un entero no es la misma en todas las máquinas. Puede ser de 16 o de 32 bits. La función `sizeof(int)` nos devolverá el número de bytes usados por un entero.

1.5. Declaración de variables y constantes

Para declarar una variable en C, se usa la notación

```
tipo identificador;
```

Es decir, si queremos declarar una variable `i` como de tipo `int`, lo haríamos con la siguiente línea de código:

```
int i;
```

También se pueden declarar varias variables del mismo tipo en una misma línea de declaración, separadas por comas. Es decir:

```
int i,j,k;
```

En cuanto a la declaración de constantes, se hace mediante la directiva de precompilación `#define`, usando la sintaxis:

```
#define identificador_constante valor
```

Como vemos, no acaba en `;` ya que es una directiva de precompilación, no una instrucción. Tampoco se declara qué tipo tiene la constante. El compilador únicamente va a cambiar en el código cada ocurrencia del texto `identificador_constante` por `valor`.

Llegados a este punto, hay que indicar una característica importante de C. El lenguaje C es lo que se denomina *case-sensitive*, es decir distingue mayúsculas de minúsculas. Para C, el identificador `a` es distinto del identificador `A`. Igualmente `dato`, `DATO` y `DaTo` serían identificadores distintos.

Si operamos con variables de diferentes tipos, C realiza conversiones de tipos de manera automática. Por ejemplo, si sumamos un `int` y un `float` entonces el `int` será convertido a `float` y el resultado será un `float`. Por otro lado, si asignamos el valor de una variable de tipo `float` a una variable `int`, entonces se truncará el valor real, dejando solamente la parte entera. Así:

```
unsigned int i;
float f;

f=3.14;
i=f;
```

pondrá automáticamente `i` a 3, truncando el valor real. En este ejemplo podemos ver también cómo se realiza la asignación de valores a las variables, empleando el operador `=`.

La conversión explícita de tipos usa el denominado *casting*. Por ejemplo,

```
i= (unsigned int) f;
```

transforma el valor del float a un entero sin signo. La mayor parte de las conversiones preservan el valor pero puede ocurrir que no sea posible y se produzca un *overflow* o se pierda precisión en los resultados. El C no nos avisará si esto ocurre.

Una variable también puede ser inicializada en su declaración:

```
int i=0;
float eps=1.0e-5;
```

Cuando queramos que una variable no cambie nunca de valor, debemos definirla como const. Por ejemplo, si se declara un entero de la forma

```
const int i = 6;
```

entonces será ilegal hacer posteriormente i=7.

1.6. Tipos estructurados

Las variables del mismo tipo pueden ser colocadas en vectores o *arrays*. Así:

```
char letters[50];
```

define un array de 50 caracteres, siendo `letter[0]` el primero y `letter[49]` el último carácter. C no comprueba el índice referenciado por lo que si nos salimos de los límites del array, C no nos advertirá. El programa fallará durante la ejecución y no en el momento de la compilación.

Podemos definir también arrays multidimensionales (ej: matrices). Por ejemplo,

```
char values[50][30][10];
```

define un array de 3 dimensiones. Observad que no podemos referenciar un elemento con `values[3,6,1]`, sino que tendremos que hacer `values[3][6][1]`.

Las variables de diferentes tipos pueden ser agrupadas en las llamadas structures (estructuras).

```
struct person {
  int age;
  int height;
  char surname[20];
} fred, jane;
```

define 2 estructuras del tipo `person` cada una con 3 campos. Cada campo es accedido usando el operador '`.`' . Por ejemplo, `fred.age` es un entero que puede ser usado en asignaciones como si fuese una variable sencilla.

`typedef` crea un tipo nuevo. Por ejemplo,

```
typedef struct{
  int age;
  int height;
  char surname[20];
} person;
```

crea un tipo llamado `person`. Observad que `typedef` crea nuevos tipos de variables pero no crea variables nuevas. Éstas son creadas como variables de los tipos predefinidos:

```
person fred, jane;
```

Las estructuras pueden ser asignadas, pasadas a funciones y devueltas por las mismas, pero no pueden ser comparadas, por lo que:

```
person fred, jane;

...

fred=jane;
```

es posible (los campos de `jane` son copiados en los de `fred`) pero no podemos ir más allá haciendo:

```
if (fred == jane)       /* esto no funciona!!! */
  printf("la copia trabaja bien \n");
```

Para comparar dos estructuras entre sí, tendríamos que ir campo a campo. Por ejemplo:

```
if ((fred.age==jane.age) && (fred.height==jane.height))
        /* comparaci\'on de los campos age y height */
```

1.7. Expresiones y operadores

Una vez que se han definido variables, se puede operar sobre ellas para construir expresiones. Las expresiones son combinaciones de variables con operadores que dan un cierto resultado. En C podemos distinguir operadores aritméticos, de comparación y lógicos:

- Aritméticos: son operadores que combinan expresiones aritméticas para dar un resultado aritmético. Son los siguientes:

  + : suma.

  - : resta.

  ***** : multiplicación (cuidado, también se usa como operador *contenido de* en los punteros, que veremos más adelante).

  / : división tanto entera como real. Su resultado dependerá del tipo de datos de los operadores. Así, 3/2=1 ya que ambos operadores son de tipo entero y el resultado de la división entera vale 1, pero 3.0/2=1.5 ya que 3.0 es un valor real y el resultado será el de la división real.

  % : resto de la división entera (por tanto, sólo es aplicable a enteros).

- De comparación: son operadores que combinan expresiones aritméticas para dar un resultado lógico o booleano. C no tiene un tipo booleano predefinido, por tanto usa la convención de que 0 equivale a falso y distinto de 0 (en general, 1) equivale a cierto. Los operadores de comparación son los siguientes:

  == : comparación de igualdad.
  MUY IMPORTANTE: no confundir nunca con el operador de asignación = , ya que el compilador no avisa si se pone por error en una condición. Esto se debe a que es correcto hacer una asignación dentro de una expresión lógica.

  != : comparación de desigualdad (distinto de).

  > : mayor que.

  < : menor que.

  >= : mayor o igual que.

  <= : menor o igual que.

- Lógicos: son operadores que combinan expresiones booleanas (que como hemos dicho antes son realmente valores aritméticos) para dar un resultado booleano. Son los siguientes:

  && : Y lógico. Mucho cuidado, no confundir con & , que es el Y bitwise (el que hace el Y bit a bit de los datos).

  || : O lógico. Mucho cuidado también, pues existe el O bitwise que es el operador |.

! : NO lógico.

Para evaluar una expresión se usa el orden de precedencia habitual, que es modificable por el uso de paréntesis.

1.8. Entrada y salida de datos

Para que un programa en C pueda comunicarse con el exterior (mostrar salida por pantalla o recoger entrada desde teclado), debe usar una serie de funciones que se encuentran definidas en **stdio.h**, por lo cual es extremadamente frecuente que esta librería se encuentre dentro de los `#include` de un programa en C.

1.8.1. Salida

La función básica para mostrar datos por pantalla en C es `printf`. La sintaxis de la función es:

```
printf(cadena_de_formato, expresion, expresion, . . . , expresion);
```

La `cadena_de_formato` es una cadena de texto normal (entre comillas dobles) que tiene una serie de ocurrencias de los llamados caracteres de formato, que indican el tipo de la expresión que se va a mostrar y que viene como argumento posterior. Así por ejemplo, la siguiente línea de código:

```c
printf("El resultado es %d",15);
```

Al ejecutarse emitiría por pantalla el mensaje:

```
El resultado es 15
```

Como vemos, `%d` es el carácter de formato que indica que la expresión a mostrar es de tipo entero. Los distintos caracteres de formato que se usan son:

%d : enteros.

%u : enteros sin signo.

%f : números reales en coma flotante.

%e : números reales en notación científica.

%g : números reales, eligiendo la representación más adecuada (coma o científica).

%c : carácter ASCII.

%s : cadenas de caracteres o strings.

Hay que ser cuidadoso con los caracteres de formato y las expresiones asociadas, pues en ocasiones se pueden poner de manera incompatible resultando salidas por pantalla inesperadas (por ejemplo, si indicamos que vamos a imprimir un entero, de 4 bytes, con `%d` pero luego la expresión es de tipo float, con 8 bytes, se nos mostrarían los 4 primeros bytes del float resultado de la expresión, que posiblemente no tengan nada que ver con lo que pensábamos obtener).

A los caracteres de formato se le pueden añadir modificadores que indiquen cuánto espacio dejar en la cadena para representar los números. Por ejemplo, `%3d` indica que como mínimo hay que dejar 3 espacios para representar el entero correspondiente (si ocupara más posiciones, se extendería automáticamente hasta donde fuera necesario). `%3.5f` indica que hay que guardar al menos 3 posiciones para la parte entera y 5 como máximo para la parte decimal.

Por otra parte, hay una serie de caracteres útiles a la hora de usar printf que se suelen incluir en la cadena de formato. Entre ellos, destacan el ya conocido '`\n`' que es el retorno del carro y el '`\t`', que representa al tabulador.

Otra función que se usa con frecuencia para la salida de datos, pero sólo para el tipo char, es **putchar**. Su sintaxis es `putchar(c)`, siendo `c` una variable del tipo char o una constante ASCII (entre comillas simples), como 'a'.

1.8.2. Entrada

La función más usual para obtener datos desde teclado en C es la función `scanf`, también de la librería **stdio.h**. La sintaxis de esta función es:

```
scanf(cadena_formatos,puntero1,puntero2, puntero3, . . . );
```

En este caso, la cadena de formatos es una cadena que únicamente va a tener caracteres de formato (%d, %f, %c), tantos como variables que se vayan a leer y del mismo tipo que éstas. Sin embargo, una diferencia fundamental con `printf` es que `scanf` espera como argumentos direcciones de memoria (punteros) en las que depositar los datos leídos. Para conseguir ese efecto,

hay que preceder con un `&` la variable sobre la que se va a leer. Es decir, si vamos a leer sobre `i` de tipo `int`, su `scanf` correspondiente es:

```
scanf("%d",&i);
```

Otra función útil para leer datos pero únicamente del tipo char es getchar. Esta función se utiliza como `c=getchar();`, con `c` del tipo char, y deposita sobre la variable `c` el carácter que se lee desde teclado.

1.9. Estructuras de decisión

C, como casi todos los lenguaje de programación imperativos, dispone dentro de las estructuras de control de una serie de estructuras de decisión para decidir qué instrucción o instrucciones ejecutar según el resultado de una expresión booleana.

1.9.1. Decisión simple

Son las decisiones en las que se ejecuta la instrucción o instrucciones si la condición a comprobar es cierta. En C se hace mediante la instrucción if con la siguiente sintaxis:

```
if (expresion) instruccion;
```

De manera que cuando expresión es cierta, se ejecuta instrucción. Es obligatorio que la expresión a comprobar esté entre paréntesis.

En el caso de ser un bloque de instrucciones, simplemente se encierran entre las llaves que delimitan un bloque de instrucciones:

```
if (expresion) {
  instruccion1;
  instruccion2;
  . . .
  instruccionN;
}
```

1.9.2. Decisión doble

La decisión doble ejecuta una instrucción o grupo cuando la condición es cierta y otra instrucción o grupo cuando es falsa. En C se consigue con `if else`, con la siguiente sintaxis:

```
if (expresion) instruccion_v;
else instruccion_f;
```

Igualmente, se pueden sustituir las instrucciones por bloques de instrucciones. Ejemplo:

```
if (i==3)   /* no hace falta llaves para una sola sentencia */
   j=4;
else {
     /* las llaves son necesarias si hay mas de una sentencia */
     j=5;
     k=6;
}
```

1.9.3. Decisión múltiple

La decisión múltiple es una extensión de las anteriores a un número indeterminado de casos. En C se consigue mediante la estructura `switch`, la cual mantiene la siguiente sintaxis:

```
switch(expresion) {
  case v1 : instr1_1;
            ...
            instr1_N1;
            break;
  case v2 : instr2_1;
            ...
            instr2_N2;
            break;
            ...
            ...
  case vM : instrM_1;
            ...
            instrM_NM;
            break;
  default: instr_def_1;
           ...
           instr_def_Ndef;
}
```

Como vemos, no es necesario poner las secuencias de instrucciones asociadas a cada caso entre llaves. Los valores de cada caso sólo pueden ser constantes, y se utiliza la etiqueta `default` para indicar las instrucciones a

ejecutar en el caso en el que la expresión no tome ninguno de los valores dados previamente.

Otro elemento importante es la instrucción `break` al final de las instrucciones de cada caso. Es imprescindible, pues de otra manera se seguirían ejecutando el resto de instrucciones que van después hasta encontrar un `break`, lo cual provocaría una ejecución incorrecta.

También es posible poner el mismo código para diversos casos, usando para ello la acumulación de etiquetas:

```
switch(expresion) {
  case v1 :
  case v2 :
  case v3 : instr1_1;
            ...
            instr1_N1 ;
            break;
  ...
}
```

Ejemplo de sentencia switch:

```
int i;

...

switch(i){
  case 1: printf("i es uno \n");
          break;
          /* si no ponemos el break aqui, continuara */
          /* ejecutando el c\'odigo del siguiente case */
  case 2: printf("i es dos \n");
          break;
  default: printf("i no es ni uno ni dos \n");
           break;   /* para default, el break es opcional */
}
```

1.10. Estructuras de repetición

El lenguaje C también aporta estructuras para la repetición de una o varias instrucciones (bucles). El bucle fundamental es el `while`, con la siguiente sintaxis:

```
while (expresion) instruccion;
```

O para múltiples instrucciones:

```
while (expresion) {
  instruccion1;
  instruccion2;
  ...
  instruccionN;
}
```

Con esta estructura se ejecutará la instrucción o el bloque mientras la expresión se evalúe a cierto. Existe una estructura similar, el `do while`, que hace la comprobación de continuidad al final en vez de al principio, y que mantiene la siguiente sintaxis:

```
do instruccion; while (expresion);
```

O para múltiples instrucciones:

```
do {
  instruccion1;
  instruccion2;
  ...
  instruccionN;
} while (expresion);
```

La diferencia es que con esta última estructura siempre se ejecuta al menos una vez la instrucción o bloque asociados. De nuevo, la ejecución se repetirá mientras la expresión se evalúe a verdadero. Ejemplos de utilización:

```
while (i<30){      /* test al principio del bucle */
   something();
   ....
}

....

do {
   something();
   ....
} while (i<30);    /* test al final del bucle */

....
```

Por último, C aporta una estructura de repetición que puede llegar a ser extremadamente compleja, pero que también se puede manejar de manera simple. Hablamos de la instrucción `for`, cuya sintaxis básica es:

```
for (instr_inic; expresion; instr_n) instruccion;
```

O también para múltiples instrucciones:

```
for (instr_inic; expresion; instr_n) {
  instruccion1;
  instruccion2;
  ...
  instruccionN;
}
```

En esta estructura, antes de entrar al bucle por primera vez, se ejecutan las instrucciones de inicialización `instr_inic`. Tras ello, se pasa a evaluar la expresión `expresion`, y si es cierta se ejecutan las instrucciones asociadas. Al acabar esta ejecución se ejecutan las instrucciones `instr_n` y se vuelve al punto de evaluar la expresión, acabando el ciclo cuando la expresión se evalúa a falso.

Por tanto, la estructura `for` expresada anteriormente equivale al siguiente bucle `while`:

```
instr_inic;
while (expresion) {
  instruccion1;
  ...
  instruccionN;
  instr_n;
}
```

A pesar de que `instr_inic`, `expresion` y `instr_n` pueden ser todo lo complicadas que se quiera y pueden no guardar ninguna relación, en general se usará `instr_inic` para darle un valor inicial al contador, `expresion` para comprobar si ha llegado al final de la secuencia e `instr_n` para realizar el incremento/decremento del contador.

Ejemplo de la instrucción `for`:

```
...
for (i=0; i<5; i=i+1){
   something();
}
...
```

Este fragmento sería equivalente, como hemos comentado, al siguiente:

```
i=0;
while (i<5) {
  something();
  i=i+1;
}
```

Dentro de cualquier bucle (tanto while como for), la instrucción `continue` interrumpe el ciclo actual y va al siguiente ciclo, mientras que `break` detiene todas las iteraciones. Por ejemplo, en el siguiente fragmento 0 y 2 serán los valores impresos.

```
int i;

....
i=0;
while (i<5) {
   if (i==1) {
      i=i+1;
      continue;
   }
   if (i==3) break;
   printf("i=%d\n",i);
   i=i+1;
}
...
```

1.11. Contracciones

Los más ejercitados en C usan formas abreviadas para contraer el código. Algunas de estas formas son ampliamente aceptadas. Otras son excesivamente oscuras y difíciles de entender, por lo que, aunque produzcan código más rápido, malgastarán el tiempo de futuros programadores.

La expresión `i++` es equivalente a `i=i+1`. Esta contracción (y la del `i--`) es muy común. La operación se realiza una vez que la variable es usada. Aunque si usamos la expresión `--i` o `++i`, la variable se actualiza antes de ser usada. Por lo que si ejecutamos el siguiente código:

```
...
i=4;
printf("i=%d\n",i++);
/* le pasa primero el valor de i a printf y */
/* luego realiza la asignacion i=i+1 */

...
i=4;
printf("i=%d\n",++i);
/* primero realiza la asignacion i=i+1 y */
/* luego le pasa el valor de i a printf */
```

ambas dejarán `i` con un 5, pero en el primer fragmento 4 será el valor impreso mientras que será 5 para el segundo.

`i+=6` es equivalente a `i=i+6`. Este estilo de contracción no es tan común, pero puede usarse con la mayoría de los operadores aritméticos.

Las expresiones con operadores de comparación devuelven un 1 si la comparación es true, 0 si es false, por lo que `while(i!=0)` y `while(i)` son equivalentes.

La construcción

```
if (cond) exp1; else exp2;
```

puede ser abreviada usando

```
(cond)?exp1:exp2
```

Aunque este tipo de notación debería usarse prudentemente.

1.12. Funciones

El lenguaje C no tiene procedimientos, solamente funciones. Sus definiciones no pueden ser anidadas, pero todas excepto `main` pueden ser llamadas recursivamente. En el estándar ANSI C la definición de una función es:

```
<tipo_funcion> <nombre_funcion>( <lista_argumentos_formales> )
{
  <variables_locales>
  <cuerpo>
}
```

Por ejemplo:

```
int mean(int x, int y) {
   int tmp;

   tmp=(x+y)/2;
   return(tmp);
}
```

Las funciones finalizan cuando:

- La ejecución alcanza la llave } de la función. Si se supone que la función devuelve algún valor, dicho valor será indefinido.
- Se llega a la sentencia `return`, devolviendo el control a la función que la llamó (junto con el valor de la variable que acompañe a `return`). La función que hace la llamada puede ignorar el valor devuelto.
- Se llega a la sentencia exit, finalizando la ejecución del programa completo.

1.12.1. Argumentos: llamadas por valor

Todos los parámetros en C se pasan **por valor**. Cada vez que se llama a una función, los argumentos de la llamada se copian en los parámetros de la función, que pueden verse como variables locales de la función. Así, la función que se invoca recibe los valores de sus argumentos en variables temporales y no en las originales. De este modo, la función que se invoca no puede alterar directamente una variable de la función que hace la llamada. Sólo puede modificar su copia privada y temporal. Por ejemplo, si tenemos el siguiente código:

```c
#include <stdio.h>

void mean(int a, int b, int return_val) {
     return_val=(a+b)/2;
     printf("return_val en mean es %d\n", return_val);
     a=20;
     printf("la variable a dentro de la funcion mean vale: %d\n", a);
     /* imprimira el valor 20 */
}

main() {
   int a, b;
   int answer;

   a=7;
   b=9;
   mean(a,b,answer);
   printf("El valor para media(%d,%d) es %d\n", a,b, answer);
   printf("Valor de la variable a: %d \n", a);
   /* imprimira el valor 7 */
}
```

No funcionará. Aunque `return_val` es inicializado al valor correcto en `mean()`, no ocurre lo mismo con `answer`, pues tanto `answer` como `return_val` son variables distintas. El valor de `answer` se copia en `return_val` cuando llamamos a `mean()`. La función `mean()` no sabe dónde se almacena `answer`, por lo que no puede alterarla. Por otro lado, la variable `a`, aunque tenga el mismo nombre, es distinta para la función `main()` y para la función `mean()`. Las variables son locales a las funciones en las que se declaran.

Para llevar a cabo el paso por referencia se necesita el uso de punteros, que veremos en la siguiente sección. Esto no es necesario en el caso de vectores. Cuando un argumento es un nombre de vector, el valor que se pasa es un puntero al primer elemento del vector y no una copia de los elementos del vector.

1.13. Punteros

Un puntero es una variable que contiene la dirección de otra variable. Así, si `c` es un `int` y `p` es un puntero que apunta a él, podríamos representar la situación como se observa en la figura 1.1.

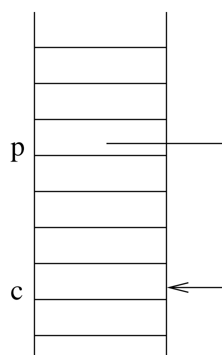


Figura 1.1: La variable **p** es un puntero que apunta a **c**.

El operador unario & da la dirección de un objeto, de modo que la proposición:

```
p = &c;
```

asigna la dirección de `c` a la variable `p`, y se dice que `p` *apunta a* `c`. El operador & sólo se aplica a variables y elementos de vectores. No puede aplicarse a expresiones o constantes.

El operador unario `*` es el operador de indirección o desreferencia. Cuando se aplica a un puntero, da acceso al objeto al que señala el puntero.

En el siguiente ejemplo se muestra cómo declarar un puntero y cómo emplear & y `*`:

```
int x=1, y=2;  /* se puede realizar la declaracion */
               /* y la asignacion de valores a la vez */
int z[10];
int *ip;     /* ip es un apuntador a int */

ip = &x;      /* ip ahora apunta a x */
y = *ip;      /* y es ahora 1 */
*ip = 0;      /* x es ahora 0 */
ip = &z[0]    /* ip ahora apunta a z[0] */
```

Si `ip` apunta al entero `x`, entonces `*ip` puede presentarse en cualquier contexto donde `x` pueda hacerlo.

```
*ip = *ip+10;  /* incrementa *ip en 10, es decir, el entero x */
*ip += 1 ;
++*ip ;
(*ip)++;       /* tres metodos para incrementar en */
               /* uno aquello a lo que ip apunta */
```

Puesto que los punteros son variables, se pueden emplear sin indirección. Por ejemplo, si `ip` e `iq` son punteros a entero, entonces:

```
iq = ip;
```

copia el contenido de `ip` en `iq`. Así, hace que `iq` apunte a lo que `ip` está apuntando.

Los punteros tienen unos tipos asociados. Un puntero a un `int` es `int *` y es de diferente tipo que un puntero a un `char` (que es `char *`). La diferencia surge principalmente cuando el puntero se incrementa: el valor del puntero se incrementa según el tamaño del objeto al que apunta.

Hay un valor especial para punteros nulos (`NULL`) y un tipo especial para punteros genéricos (`void *`).

1.13.1. Punteros y argumentos de funciones

Como hemos visto, en las funciones los argumentos se pasan por valor. Si queremos que la función que se invoca altere una variable de la función que la llama, se deberán emplear los punteros.

Por ejemplo, con la siguiente función:

```
void intercambio(int x, int y) {
  int temp;

  temp=x;
  x=y;
  y=temp;
}
```

La llamada `intercambio(a,b)` no consigue el efecto deseado (intercambiar los valores de las variables `a` y `b`), pues la función sí que intercambiará los valores entre las variables `x` e `y`, pero estas variables no son las mismas que las variables `a` y `b` (no tienen el mismo espacio de memoria asignado).

La solución al problema consiste en definir la función intercambio como sigue:

```
/* intercambia *px y *py */
void intercambio (int *px, int *py) {
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

Con la llamada `intercambio(&a,&b)`, se le pasará a la función intercambio las direcciones de las variables `a` y `b`, de modo que todas las operaciones sí que se realizarán sobre el mismo espacio de memoria asignado.

Podemos ver el resultado gráfico de la operación `intercambio` en la figura 1.2.

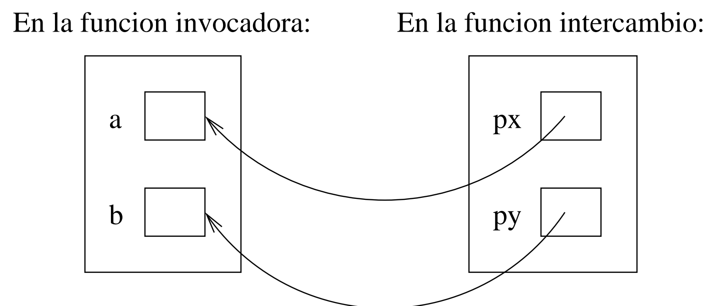


Figura 1.2: Esquema de la memoria al utilizar la función **intercambio()**.

1.13.2. Punteros y vectores

En C existe una fuerte relación entre punteros y vectores. Cualquier operación que pueda lograrse por la indexación de un vector, también podría lograrse mediante el uso de punteros.

Veamos algunos detalles previos. Definamos un vector de tamaño 10, esto es un bloque de 10 objetos consecutivos a los que podemos referenciar como `a[0]`, `a[1]`, ..., `a[9]`:

```
int a[10];
```

Se puede ver una representación gráfica en la figura 1.3.

Si `pa` es un puntero a entero, declarado como:

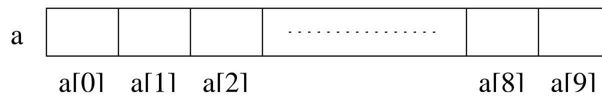


Figura 1.3: Representación gráfica del vector definido con **int a[10];**.

```
int * pa;
```

entonces la asignación `pa = &a[0]` hace que `pa` apunte al primer elemento de `a`, o lo que es lo mismo, que contenga su dirección. Se puede ver una representación gráfica en la figura 1.4

Figura 1.4: Representación gráfica de la asignación **pa = &a[0]**.

Entonces automáticamente y por definición tendremos que `pa+1` apunta a la siguiente posición del vector, como se puede ver en la figura 1.5.

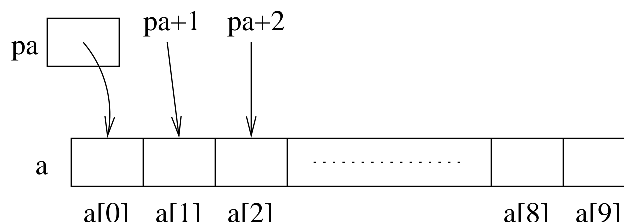


Figura 1.5: La expresión **pa+1** apunta a la siguiente posición del vector, etc.

Además, es posible referenciar a las componentes a través del nombre del vector, como si de un puntero se tratase. Por ejemplo, las expresiones `*(pa+1)`, `*(a+1)` y `a[1]` son equivalentes y se puede referenciar a la posición `a[i]` como `*(a+1)`. Es posible, por tanto, hacer una asignación `pa=a`, pues equivale a la asignación `pa=&a[0]`.

Existen, sin embargo, diferencias entre una variable de tipo vector y otra de tipo puntero. Una variable de tipo vector `a` es un objeto del sistema que

hace referencia a un conjunto de objetos consecutivos del tipo sobre el que se define. Sin embargo, si apuntamos con un puntero `pa` de tipo base del vector a una componente del vector, no tenemos el conjunto de objetos sino tan sólo la dirección de uno de ellos. Por tanto, de las siguientes construcciones solamente son legales las que se indican:

```
int a[10];
int *pa;

pa = a;     /* legal */
pa++;       /* legal */
a = pa;     /* ilegal */
a++;        /* ilegal */
```

Cuando se quieran pasar vectores a funciones, se deberá definir en los parámetros de la función un puntero al tipo base del vector. En la llamada de la función, se pasará el nombre del vector. Se estará pasando, en realidad, la dirección del primer elemento de la función. Dentro de la función, el puntero es una variable local, que contiene una dirección.

Por ejemplo:

```
/* strlen: retorna la longitud de la cadena s */
int strlen(char *s) {
  int n;

  for (n=0; *s!='\0';s++) n++;
  return(n);
}
```

Puesto que `s` es un puntero, es correcto incrementarlo. Sin embargo, `s++` no tienen ningún efecto sobre la cadena de caracteres de la función que llamó a `strlen`, sino que simplemente incrementa la copia local del puntero `strlen`. Las siguientes llamadas son correctas:

```
strlen("hola mundo"); /*longitud de una cadena constante */
strlen(array);        /* longitud de un vector: char array[100]*/
strlen(ptr);          /* longitud de un puntero: char *ptr */
```

En cuanto a los parámetros de una función, es equivalente decir `char *s` a decir `char s[]`. El primero de ambos es más explícito respecto al carácter de puntero, por lo que se utiliza mucho más. Cuando un nombre de vector se pasa como parámetro de una función, el programador puede interpretar a su conveniencia el carácter de puntero o de vector, incluso utilizando ambas expresiones en el interior de la función.

Es posible pasar un subvector a una función, pasando el puntero al elemento de inicio del subvector, que en la función será interpretado como un nuevo vector de longitud distinta. Por ejemplo, si `a` es un vector de 10 componentes, entonces invocar a la función `f` definida con un parámetro de tipo vector mediante la orden `f(&a[2])` consiste en invocar a la función con un subvector que contiene las 8 últimas componentes del original.

1.13.3. Punteros y estructuras

Los punteros son especialmente útiles cuando las funciones operan sobre estructuras. Usando punteros evitamos copias de grandes estructuras de datos.

```
typedef struct {
  int age;
  int height;
  char surname[20];
} person;

int sum_of_ages (person *person1, person *person2) {
  int sum;

  sum=(*person1).age + (*person2).age;
  return(sum);
}
```

Operaciones como `(*person1).age` son tan comunes que existe una forma más natural para notarlo: `person1->age`. Esta última forma será la más utilizada en clase y en prácticas.

1.14. Strings

En el lenguaje C una string es exactamente un vector de caracteres. El final de una string se denota por el byte de valor cero. Las funciones de manipulación de strings están declaradas en la librería **string.h**. Para hacer uso de estas funciones, se debe incluir la sentencia

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

A continuación se muestran las funciones para cadenas más comunes, suponemos las siguientes definiciones:

```c
char *s;
const char *cs;
int c;
```

- `char *strcpy(s,cs)`: Copia la cadena cs a la cadena s, incluyendo '\0'. Devuelve s.
- `char *strncpy(s,cs,n)`: Copia hasta `n` caracteres de la cadena `cs` a `s`. Devuelve `s`. Rellena con '\0' si `cs` tiene menos de `n` caracteres.
- `char *strcat(s,cs)`: Concatena la cadena `cs` al final de la cadena `s`. Devuelve `s`.
- `int strcmp(s,cs)`: Compara la cadena `s` con la cadena `cs`. Devuelve <0 si `s`<`cs`, 0 si `s`==`cs`, o >0 si `s`>`cs`.
- `char *strchr(cs,c)`: Retorna un puntero a la primera ocurrencia de `c` en `cs`, o `NULL` si no está presente.
- `size_t strlen(cs)`: Retorna la longitud de `cs`.
- `char *strstr(s,cs)`: Retorna un puntero a la primera ocurrencia de la cadena `cs` en `s`, o NULL si no está presente.
- `char *gets(s)`: Lee una string de la entrada estándar y la guarda en `s`, hasta que se introduce un salto de línea o EOF. No comprueba si se sobrepasa el tamaño que tiene `s`.

Ejemplos de utilización de estas funciones:

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

char str1[10];  /* reserva espacio para 10 caracteres */
char str2[10];
char str3[]="initial text";
    /* str3 se pondra a la dimension que deseemos y su */
    /* ultimo caracter sera el byte 0. Solo podemos */
    /* inicializar strings en la declaracion de esta manera. */
char *c_ptr; /* declara un puntero, pero no lo inicializa */
unsigned int len;

main() {
  strcpy(str1, "hello");
  /* copia "hello" en str1. Si str1 no es lo */
  /* suficientemente grande para contenerlo, mala suerte */
  /* En str1 se guardan los valores: */
  /* 'h','e','l','l','o','\0' */
  strcat(str1," sir");
  /* Concatena "sir" sobre str1. Si str1 es demasiado */
  /* corto, mala suerte */
  /* Valores en str1: */
  /* 'h','e','l','l','o',' ','s','i','r','\0' */

  len=strlen(str1); /* contar el numero de caracteres */

  if (strcmp(str1,str3))   /*comparar strings */
      printf("%s y %s son diferentes \n", str1, str3);
  else printf("%s y %s son iguales \n", str1, str3);

  if (strstr(str1, "boy") == (char*) NULL)   //Buscar subcadenas
      printf("El string <boy> no esta en %s\n",str1);
  else printf("El string <boy> esta en %s\n",str1);

  c_ptr=strchr(str1,'o');  /* encontrar la primera 'o' en str1 */

  if (c_ptr == (char *) NULL)
      printf("No hay ninguna o en %s\n",str1);
  else{
    printf("%s es la primera o en %s hasta el final.\n",c_ptr,str1);
    strcpy(str2,c_ptr); /*copia esta parte de str1 sobre str2 */
  }
}
```

1.15. Keywords

No podremos usar las siguientes palabras para nombres de variables, etc.

auto	**break**	**case**	**char**	**const**
continue	**default**	**do**	**double**	**else**
enum	**extern**	**float**	**for**	**goto**
if	**int**	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**	**struct**
switch	**typedef**	**union**	**unsigned**	**void**
volatile	**while**			

Algunas de ellas no han sido aún descritas.

auto : ésta es la clase de almacenamiento por defecto para variables, por lo que no se usa explícitamente.

static : las variables declaradas como static preservan su valor entre llamadas.

enum : C posee tipos enumerados. Una enumeración es una lista de valores enteros constantes, como en:

```
enum boolean {NO, YES};
```

El primer nombre en un enum tiene valor 0, el siguiente 1, y así sucesivamente, a menos que sean especificados con valores explícitos.

1.16. Variables externas y alcance

Las variables que se declaran en el interior de `main`, o de cualquier función son locales al lugar donde se declaran. Ninguna otra función puede acceder a dichas variables. Estas variables tan sólo existen mientras la función esté siendo invocada, de modo que no conservan sus valores de una llamada a la función a la siguiente llamada. Es por ello que es preciso inicializarlas en cada invocación.

Sin embargo, también es posible definir variables globales o externas que pueden ser utilizadas como medio de transferencia de información de unas

funciones a otras sin el uso de parámetros, aunque deben ser usadas con mucho cuidado para evitar problemas de modificaciones no deseadas.

Para definir una variable externa, debemos declararla una vez fuera de la función, y también dentro de las funciones que puedan usarla, precedida por la palabra reservada `extern`, salvo si la declaración de la variable se realiza dentro del archivo fuente antes del primer uso en las funciones.

Si un programa se encuentra repartido en varios archivos, entonces si una variable externa se define en el primer archivo y se usa en los siguientes, debe obligatoriamente hacerse uso de la palabra reservada `extern`. Lo habitual en programación en C consiste en reunir todas las declaraciones de variables externas y globales en un único archivo (por convención, con extensión .h) que será incluido en todos los demás mediante la sentencia `#include`.

1.17. Manejo de ficheros

Antes de leer o escribir un fichero hay que abrirlo y asociarle un nombre lógico (variable del programa, de tipo puntero) al nombre físico (nombre externo del fichero). Por ejemplo:

```
#include <stdio.h>

FILE *fp;
```

declara que `fp` es un puntero a un fichero. La sentencia:

```
fp= fopen(name,mode);
```

asocia el puntero `fp` con el fichero `name` (entre comillas dobles) y especifica el modo en el que se va a tratar al fichero: "r" (lectura), "w" (escritura) o "a" (añadido). Si se intenta escribir o añadir un fichero no existente, se creará el fichero. Si el fichero ya existe y se abre con modo escritura, se perderá el contenido anterior.

En la librería **stdio.h** están definidas las diversas funciones de manejo de ficheros. Entre ellas, destacamos:

- `FILE * fopen (const char * filename, const char *mode)`: abre el archivo en el modo indicado y lo asocia a un puntero.
- `int getc (FILE *fp)`: devuelve el siguiente carácter del fichero, o bien EOF si ocurre algún error (o se ha llegado al final del fichero).
- `int putc (int c, FILE *fp)`: escribe el carácter `c` en el archivo `fp` y devuelve el carácter escrito, o EOF si ocurre algún error.

- `int fscanf (FILE *fp, char *formato,...)`: similar a `scanf`, pero leyendo los datos de un archivo.
- `int fprintf (FILE *fp, char *formato, ..)`: similar a `printf`, pero escribiendo los datos en un archivo.
- `char *fgets(char *linea, int maxlinea, FILE *fp)`: lee la siguiente línea del archivo `fp` y la deja en el vector de caracteres `linea`. Se leen hasta `maxlinea-1` caracteres.
- `int fputs(char *linea, FILE *fp)`: escribe una cadena a un archivo. Devuelve EOF si ocurre un error y 0 si no ocurre.
- `int fflush(FILE *fp)`: ocasiona que se escriban todos los datos pendientes de escritura y se limpie el buffer.
- `int fclose (FILE *fp)`: cierra el archivo, liberando al puntero de archivo.

Práctica 2

Introducción al IDE de Turbo-C

2.1. Objetivos

Los objetivos básicos de esta práctica son:

- Conocer el IDE del Turbo-C, entorno en el cual se trabajará durante todo el curso.
- Empezar escribiendo programas para familiarizarnos con el entorno.

2.2. ¿Qué es el IDE?

Es un entorno de desarrollo integrado. Sus iniciales vienen del inglés Integrated Development Environment. El objetivo de esta práctica es dar a conocer al alumno el entorno de trabajo en el que se realizarán las prácticas en esta asignatura, así como conocer cuales son los procesos de trabajo más habituales con el mismo. De cualquier manera para ampliación de conocimientos del mismo remitimos al alumno a la guía de usuario del Turbo-C o bien a la consulta en línea que posee el entorno de trabajo.

Para acabar con la práctica el alumno escribirá un par de programas en C y realizará las acciones pertinentes para su ejecución. Es probable que en el momento de la realización de la práctica el alumno no tenga los conceptos suficientes como para entender todas las partes de los programas, pero insistimos que el objetivo de la práctica es sólo la familiarización con el entorno. No obstante creemos que sí se puede hacer una idea intuitiva de las partes del programa y de cual es la filosofía del lenguaje.

2.3. Empezando con el IDE del Turbo-C

Si estamos trabajando sobre el sistema operativo MS-DOS directamente, para arrancar el IDE se debe teclear tc desde la línea de comandos de MS-DOS. Es importante que en el PATH aparezca el directorio en el que se encuentra este comando. Si lo estamos haciendo sobre cualquiera de los S.O. Windows, debemos buscar el icono correspondiente bien a través del botón de Inicio, bien a través de un acceso directo ya construido o bien navegando por las carpetas. La pantalla del IDE aparece con tres componentes principales: el menú en la parte superior, el escritorio y la línea de estado en la parte inferior. Iremos ampliando los conocimientos de cada una de estas partes en las siguientes subsecciones.

2.3.1. El Menú

Es el acceso principal a los comandos del IDE. Cuando se activa el menú se ve la opción elegida marcada con un color significativo. Hay dos maneras de elegir los comandos, bien con el ratón o bien desde el teclado. Para hacerlo con el teclado:

- Pulsa F10. Esto activa el menú.
- Para elegir la acción se usan las teclas de cursor para situarse en la opción elegida y se pulsa la tecla ENTER. Otra forma es elegir la opción que se quiera tecleando la inicial del comando. Por ejemplo, para elegir edit, una vez activado el menú correspondiente, se puede pulsar e. O bien, sin activar el menú con la tecla ALT+e se hace lo mismo. Esta combinación aparece escrita a la derecha del comando en el menú.
- Si se desea cancelar una acción pulsar la tecla ESC.

Para hacerlo con el ratón:

- Pinchar en el menú en la parte correspondiente.
- Elegir la opción seleccionada pinchando sobre ella.

En cuanto a los comandos que aparecen en el menú:

- Si en el comando del menú aparecen unos puntos suspensivos (...), significa que al pinchar saldrá una caja de diálogo.
- Si aparece una flecha (->) significa que al pinchar aparecerá otro menú.

- Si no tiene ninguna de las dos cosas significa que al pincharlo se hará una acción.
- Si algún comando aparece sombreado significa que en ese momento la acción referente a la opción no se puede realizar.

2.3.2. Las ventanas del IDE

La mayoría de acciones que se hacen en Turbo-C son en alguna ventana del IDE. Una ventana es un área de la pantalla que se puede mover, dimensionar, partir, solapar, cerrar, abrir y ampliar.

Se pueden tener tantas ventanas como se quiera (en realidad tantas como la memoria del ordenador permita), pero sólo en la que en un momento dado se esté trabajando está activa. Toda acción que se realice afectará a la ventana activa. Si hay varias ventanas solapadas, la activa siempre es la que aparece en primer término. Las componentes principales de una ventana son:

- Una columna de título.
- Una caja de cierre.
- Barras de desplazamiento.
- Una esquina de dimensión.
- Una caja de zoom.
- Un número de ventana.

Iremos viendo aquellas partes de la ventana que pueden servir de ayuda:

- Si en la parte inferior izquierda de la pantalla aparece un *, significa que el contenido de la ventana ha sido modificado y no guardado todavía.
- En la parte superior izquierda de la ventana aparece la caja de cierre, pinchando en ella (tecleando ALT+F3 o eligiendo **Window|Close** del menú) se cierra la ventana. Las ventanas que aparecen con las cajas de diálogo o con las ayudas se pueden cerrar, además, con la tecla ESC.
- El título de la ventana aparece en medio del borde horizontal superior de la misma.
- En la parte superior derecha de la ventana aparece el número de esa ventana (con ALT+0 sale un listado de todas las ventanas abiertas).

- Al lado del número de ventana aparece la caja zoom. Pinchando la caja de zoom (o con **Window|Zoom** en el menú) se puede hacer la ventana más pequeña o más grande según del estado del que se parta.

- El borde superior horizontal puede servir para desplazar la ventana por la pantalla. Si se pincha en él y manteniendo pinchado el ratón lo movemos, se comprobará su efecto.

- Las barras de desplazamiento aparecen en el borde derecho y en el inferior izquierdo de la ventana. Sirven para mover el contenido de la misma hacia arriba-abajo, izquierda-derecha, respectivamente. Este desplazamiento puede ser página a página o línea a línea dependiendo de dónde se pinche en la barra de desplazamiento:

    - Para que se pase línea a línea, se pincha en las flechas que aparecen en los extrenos de las barras de desplazamiento.
    - Para que pasen de forma continua las líneas se mantiene pinchada la flecha.
    - Para que pase página a página se pincha en la zona sombreada de la barra de desplazamiento en la parte en la que se quiere que se desplace el texto de la ventana.
    - En caso de no disponer de ratón se usan las teclas del cursor del teclado.
    - En la esquina inferior derecha está la esquina de dimensión, manteniéndola pinchada y moviendo el ratón veremos cómo cambia el tamaño de la ventana. Si no se tiene ratón se debe elegir **Window|Size/Move** del menú.

2.3.3. La línea de estado

Aparece en la parte inferior de la pantalla y tiene 3 propósitos generales:

- Recuerda cuales son los principales usos de las teclas de función y las posibles aplicables a la ventana activa en ese momento.

- Dice qué hace el programa en ese momento; por ejemplo "**Saving prog1.c...**" cuando se guarda el programa prog1.c a disco.

- Da descripción en línea del comando de menú seleccionado.

2.4. Trabajo con ficheros

Supongamos que es la primera vez que se trabaja con el entorno o que se quiere crear un nuevo programa.

Lo primero que debemos hacer es que nuestro directorio de trabajo sea el disco flexible y no el lugar que indica el IDE por defecto que suele ser un directorio del sistema de ficheros del ordenador. Para llevarlo a cabo elegimos **File|Change dir**, nos aparece una ventana que entre otras tiene la opción **drives**; pinchamos sobre ella y aparecen las unidades de la máquina entre las que está la unidad A que corresponde al disco flexible. Pinchamos doble-click sobre ésta y después pulsamos el **OK** a la izquierda de la ventana.

Al arrancar el IDE hemos dicho que una parte importante del mismo es el escritorio. En él de entrada puede aparecer abierta una ventana. En el caso que no fuera así, se debe abrir una nueva. Esto se hace con **File|New**.
IMPORTANTE: Se debe intentar que siempre haya un número pequeño de ventanas abiertas. Como la memoria del ordenador es limitada y en cualquier momento sin previo aviso puede bloquearse el sistema, se pueden causar muchos problemas.

Una vez abierta la ventana, se puede empezar a teclear el texto del programa. Una vez acabado de teclear se debe guardar el texto en un fichero del disco (duro o flexible) con **File|Save**. Si es la primera vez, pide el nombre del fichero en el que se quiere que se guarde. Si estamos guardando modificaciones a un texto ya almacenado no pide nombre y destruye el texto previo.

Dependiendo de la opción, se pueden guardar dos versiones del texto, la última con extensión **.C** y la penúltima con la extensión **.BAK**.

No tiene nada que ver el nombre del programa con el nombre del fichero en el que se guarda. Al dar el nombre del fichero, en las sesiones de prácticas se debe indicar que lo guarde en un disco flexible, que corresponde a la unidad A del ordenador.
IMPORTANTE: Es conveniente que se vaya guardando de vez en cuando y no esperar a completarlo para almacenarlo en ese momento.

Teniendo escrito el texto se pasa a ejecutarlo, con la opción **Run|Run**. Esta opción primero compila el programa y comprueba errores. Es más conveniente compilarlo y después ejecutarlo con **Compile|Compile** y **Run|Run** respectivamente. Hay una opción interesante en el menú **Compile** y es la opción **Make**. Esta compila el programa y si no hay errores crea un archivo ejecutable con extensión **.EXE**, que almacena en el directorio de trabajo actual. Más adelante en esta práctica veremos cómo hacer un ejecutable con más detalle.

¿Qué diferencia hay entre un archivo con la extensión **.EXE** y la extensión

.C? La extensión **.C** indica que el fichero es un fichero de tipo texto, que podemos visualizar con cualquier editor de textos. Si queremos ejecutar el programa, debemos meternos en el IDE, abrir el archivo y ejecutarlo. Sin embargo, la extensión **.EXE** indica que el fichero se puede ejecutar con solo invocarlo desde la línea de comandos de MS-DOS; no es un archivo de texto, no se puede editar y si se intenta visualizar saldrán una serie de símbolos ininteligibles.

Supongamos ahora que se quiere abrir un programa ya existente que se quiere revisar o modificar. Para eso usamos la opción **File|Open**, pide el nombre del fichero, que podemos elegir en la caja con el ratón o tecleando directamente el nombre. Se debe tener en cuenta que los programas, en las prácticas siempre estarán en el disco flexible, por eso o bien cambiamos de directorio de trabajo nada más empezar a trabajar o hay que hacer referencia explícita a la unidad correspondiente en cada ocasión.

2.5. Trabajando con bloques de texto

Un bloque de texto son líneas de texto consecutivas. Hay dos formas principales de seleccionar bloques de texto:

- Con el ratón, se pincha en el inicio del texto a seleccionar y se arrastra hasta el final del texto que se quiera seleccionar.
- Con el teclado: hay que situarse en el inicio y manteniendo pulsada la tecla SHIFT, con las teclas del cursor nos colocamos en el final del mismo.

Una vez seleccionado el texto podemos hacer varias cosas con él:

- Para borrar se pulsa la combinación SHIFT+Del o se elige la opción **Edit|Cut**.
- Para copiar se pulsa la combinación CTRL+Ins o se elige del menú **Edit|Copy**. Cuando hablamos de copiar lo que hace es guardarse el texto seleccionado en un almacén temporal (el clipboard), no lo copia en la ventana activa de un lugar a otro.
- Para situar el texto elegido en otro lugar mediante la opción anterior (sea de la misma ventana o de otra distinta), primero con el cursor o ratón hay que situarse en el lugar de la ventana donde se quiera copiar el texto y después se teclea SHIFT+Ins o bien **Edit|Paste**. Es decir, para realizar una copia de texto, tal y como entendemos nosotros, se

debe usar la secuencia de acciones **Copy + Paste** detalladas en los puntos anteriores.

- Para borrar el texto seleccionado usamos CTRL+Del o con **Edit|Clear** del menú.

2.6. Saliendo del Turbo-C

Hay dos formas de salir del Turbo-C:

- Para salir del IDE completamente se elige la opción **File|Quit**. Si existen cambios que no se han guardado, el IDE muestra un mensaje avisando de ello y preguntando si se quiere guardar.

- Para salir temporalmente y trabajar en el DOS, se elige **File|Dos Shell**. El IDE pemanece en memoria mientras se trabaja en el entorno del sistema operativo, cuando se quiera regresar al IDE se teclea el comando **exit** en la línea de comando del DOS.

2.7. La ayuda del IDE

El sistema de ayuda ofrece fácil acceso a información detallada del lenguaje de programación usado en esta versión del Turbo-C. Con la opción del menú **Help|Contents** aparecen todas las posibilidades de consulta del mismo.

En las distintas pantallas de ayuda hay palabras con un color distinto al resto del texto, es porque son enlaces. Esto significa que guardan más información que se puede consultar de la siguiente manera:

- Pinchando dos veces en el enlace.

- Si la pantalla de ayuda no tiene botones se debe pulsar la tecla Tab varias veces hasta que se ilumine el enlace.

- Si la pantalla de ayuda es una caja de diálogo con botones:

    - Si se tiene ratón se pincha en el correspondiente.
    - Si sólo disponemos de teclado, con la tecla Tab se elige el botón correspondiente y se pulsa enter.

Para volver a pantallas precedentes de ayuda se elige **Help|Previous Topic** o se pulsa Alt+F1.

Se puede acceder a la ayuda de varias maneras:

- Eligiendo **Help** del menú o bien pulsando Alt+H.
- Pulsando Shift+F1 para ver los índices de la ayuda.
- Pulsando F1 se obtiene información de lo que se estaba haciendo en ese momento: editando, depurando errores, compilando, ...
- Eligiendo el botón de ayuda de las cajas de diálogo se obtiene información de la misma.
- Situando el cursor en una ventana de edición sobre un término cualquiera y haciendo una búsqueda por tópico se obtiene información de ese término. La manera de hacer esto puede ser:
    - Pulsa Ctrl+F1.
    - Eligiendo **Help|Topic Search**.

Para cerrar una ventana de ayuda y volver al trabajo con el programa en cuestión se tienen estas posibilidades:

- Pulsando Esc.
- Cerrando el botón de cierre de la ventana.
- Pinchando en cualquier lugar fuera de la ventana de ayuda.

2.8. Empezando a programar

Para acabar la práctica vamos a introducir un par de programas muy sencillos. La idea es que se haga un recorrido por las opciones más relevantes del IDE y para esto nada mejor que seguir el ciclo de vida normal de todo programa: teclearlo, compilarlo para corregir los errores, ejecutarlo y guardarlo.

Queremos resaltar que la detección de errores por parte del compilador sólo sirve para asegurarnos que el programa es sintácticamente correcto de acuerdo a la sintaxis del lenguaje. Puede haber errores lógicos que el compilador no detecta y que sólo se detectan tras pruebas exhaustivas por parte del programador con múltiples casos.

El primer programa tiene como entrada el valor del radio de una circunferencia y calcula el área y el perímetro mostrando los resultados por pantalla. El código es el siguiente:

```
#include <stdio.h>

void main() {
  const float pi=3.1415 ;
  float radio, area, perim ;

  printf("Dame el radio") ;
  scanf("%f",&radio) ;
  area=pi*radio*radio ;
  perim=2*pi*radio ;
  printf("Area=%f\n",area);
  printf("Perimetro=%f\n",perim);
}
```

Una vez tecleado y guardado nuestro programa en un archivo en nuestro disco flexible lo pasamos a compilar y ejecutar según lo expuesto en apartados anteriores.

Al ejecutarlo nos damos cuenta que no nos ha dado tiempo a ver los resultados por pantalla puesto que inmediatamente ha vuelto al IDE. Esto es normal, puesto que no debemos perder de vista que el IDE es un entorno de desarrollo y no un entorno pensado para el producto final.

Para poder ver la salida tenemos dos posibilidades: una está en una opción del menú que abre una pequeña ventana que nos muestra esa salida. Esta posibilidad nos puede servir para esta práctica y quizá para la segunda pero no nos viene demasiado bien para el resto porque es una salida bastante reducida e incómoda de manejar. Trata de averiguar de qué opción del menú se está hablando.

La otra opción es introducir unas líneas de código que nos permitan ver la salida. La única instrucción del C estándar que nos puede venir bien es getch(). Ya sabemos que al llegar a la instrucción getch la ejecución se para hasta que se produce la introducción del dato correspondiente por el teclado; vamos a aprovechar este funcionamiento para pararnos hasta que se pulse la tecla enter. Introducimos las siguientes dos líneas de código al final del programa, justo antes de que éste acabe:

```
printf ("Pulsa enter para terminar .....");
getch();
```

Si vamos realizando ejecuciones vemos que las anteriores aparecen tam-

bién en pantalla, podemos plantearnos que se limpie la pantalla para ver sólo la ejecución en la que estamos. Para poder hacerlo utilizamos la instrucción de borrado:

```
#include <conio.h>

clrscr();
```

que se tiene que situar en el lugar que veáis conveniente para borrar la pantalla; lo normal es colocarlo justo después del inicio del cuerpo del programa principal.

Por último, podemos crear un ejecutable de este fichero en nuestro disco flexible de forma que se pueda ejecutar el programa sin necesidad de arrancar el IDE. La forma de crear el ejecutable varía según la instalación de IDE que tengamos, pero la solución está siempre en la opción **make EXE file**.

Se está diciendo que el resultado va a parar al disco flexible, en realidad va a parar al directorio de trabajo en el que estamos, pero debemos tener en cuenta que lo primero que hacemos al meternos en el IDE es cambiar el directorio de trabajo a nuestro disquette (opción **change dir** de **file**).

El siguiente programa tiene como objetivo la lectura de caracteres por pantalla y cuando se le introduce el caracter $, se detiene e indica el número de caracteres introducidos:

```
#include <stdio.h>

void main()  {
  char ch, fin = '$';
  int cont = 0;

  printf("Dame la cadena: ");
  fflush();
  scanf("%c",&ch);
  while (ch!=fin) {
        cont = cont + 1;
        scanf("%c",&ch);
  }
  printf("El numero de chars es: %d",cont);
}
```

Práctica 3

Ejercicios de programación

Esta práctica complementa la práctica 1, puesto que aquí se van a proponer una serie de ejercicios de programación general, con la idea de que el alumno utilice y practique los mecanismos de programación del lenguaje C explicados en la dicha práctica.

Comenzaremos suponiendo que se conocen los tipos de datos básicos y cómo formar tipos estructurados, así como hacer uso de la entrada y la salida en C.

Por ello, propondremos ejercicios de los mecanismos de programación propuestos en la práctica 1 a partir del apartado *Estructuras de decisión*. Cuando el alumno tenga dudas, debe volver al apartado correspondiente de la práctica 1 y repasarlo.

3.1. Estructuras de decisión

Ejercicio 1:

Realizar un programa que solicite un número entero por teclado. El programa debe ofrecer como salida una frase indicando si el número introducido es mayor o menor que 100 y también si es mayor o menor que 50.

Ejercicio 2:

Modificar el ejercicio anterior de manera que devuelva como resultado una frase indicando si el número introducido es par o impar.

Ejercicio 3:

Realizar un programa (calculadora) que pida por teclado dos números enteros y después pregunte la operación (suma, resta, multiplicación, división y resto) a realizar entre ellos en forma de menú y devuelva el resultado de la operación seleccionada. Debe vigilarse el hecho de que se puedan producir divisiones por cero o que se pueda introducir una operación incorrecta.

Se propone al alumno utilizar la instrucción *switch* para elegir la operación matemática.

Un ejemplo de cómo debe funcionar el programa es el siguiente:

```
Introduce un numero entero: 4
Introduce otro numero entero: 8

Operaciones posibles
====================
1. Suma
2. Resta
3. Multiplicacion
4. Division
5. Resto
Indica la operacion a realizar: 3

Resultado multiplicacion: 32
```

3.2. Estructuras de repetición

Ejercicio 4:

Realizar un programa que vaya pidiendo números enteros por teclado y los vaya sumando hasta que se le introduzca el valor 0, momento en el que nos dará el resultado de la suma. Hacer versiones para la estructura *while* y la *do while*.

Ejercicio 5:

Para generar un número aleatorio se dispone de las dos siguientes funciones en lenguaje C:

```
int rand(void);

void srand(unsigned int seed);
```

La función `rand()` devuelve un número entero aleatorio entre 0 y la constante predefinida `RAND_MAX` (es una constante que ya está definida en C dentro de la librería `stdlib.h`).

La función `srand()` usa el argumento que se le pasa como la semilla para una nueva secuencia de números aleatorios que serán devueltos al usar la función `rand()`. Estas secuencias son repetibles invocando a la función `srand()` con el mismo valor de semilla.

Si no se llama a `srand()` antes de usar la función `rand()`, ésta utiliza automáticamente una semilla con valor 1.

Si se desea generar un número entero aleatorio entre 1 y 10, debe hacerse de la siguiente manera:

```
j=1+(int) (10.0*rand()/(RAND_MAX+1.0));
```

Para poder usar las dos funciones descritas arriba, deberás incluir en el programa la librería `stdlib.h` que implementa las funciones `rand()` y `srand()`.

Se pide realizar un programa que genere un número entero aleatorio entre 1 y 50 y lo guarde en una variable sin mostrarlo al usuario. Después debe preguntar al usuario sucesivamente: ¿cual es el número?, hasta que éste lo acierte. Por cada número que introduzca el usuario, el programa debe contestarle si ha acertado, y en caso contrario si el número a adivinar es mayor o menor que el proporcionado.

Ejercicio 6:

Modificar la calculadora del ejercicio 3 de manera que al finalizar una operación pregunte si queremos realizar otra. Si contestamos con una 's' el programa deberá igualmente preguntar dos números enteros y la operación a realizar con ellos, si contestamos con una 'n', el programa deberá terminar.

Ejercicio 7:

Comprobar de manera experimental cual es el rango de una variable entera. Para ello, se debe inicializar una variable entera a 1, después incrementar

su valor hasta detectar que se desborda, es decir, que pasa de tener un valor positivo a tener un valor negativo.

Una vez ocurra el desbordamiento, se tendrá que el valor anterior antes de desbordarse indicará el máximo valor positivo que puede tener una variable entera y el valor que marca el desbordamiento indicará el menor valor negativo que puede tener una variable entera.

Un esquema del algoritmo a seguir es el siguiente:

1. Inicializar `i` a 1.

2. Mientras `i` sea positivo hacer:

    a) Guardar en `j` el valor de `i`.

    b) Sumar 1 a `i`.

3. Imprimir `i` y `j`.

Ejercicio 8:

Realizar un programa que implemente *la criba de Eratóstenes*.

La criba de Eratóstenes es un algoritmo, creado por el matemático griego Eratóstenes, mediante el cual se obtienen todos los números naturales primos que hay entre el 1 y un cierto número natural especificado. Recordemos que *número primo es aquel que sólo es divisible entre él mismo y la unidad*. Por ejemplo, el resultado de dicho algoritmo si se le da como entrada el número 10 será: 1 2 3 5 7 (los números primos del 1 al 10).

La estrategia básica a seguir para hallar los números primos que hay hasta un número n es sencilla:

- El 1 se considera primo por definición.
- El 2 y el 3 son primos.
- Realizar un bucle $i = 3$ hasta n
    Realizar un bucle j=2 hasta $i - 1$
        Si i es divisible por j entonces i NO es primo
    Si i no ha sido divisible entre ningún j entonces i ES primo

Implementar la criba de Eratóstenes a partir de la estrategia descrita arriba.

Una vez implementada la criba como se ha descrito antes, se proponen al alumno una serie de ideas que deberá implementar modificando el programa anterior para hacerlo mucho más eficiente:

- Una idea es que de todos los números que hay que comprobar entre 3 y n (bucle de la i), la mitad van a ser pares, que sabemos que no son primos, puesto que son divisibles por dos. Para evitar comprobar todos esos números pares perdiendo tiempo de computación, cambiaremos el bucle para la variable i propuesto anteriormente por el siguiente:

  Realizar un bucle $i = 3$ hasta n con incremento de 2

  Con lo que no comprobaremos ningún número par.

- Otra idea combinada con la anterior es que los números que son divisibles por 4, 6, 8, 10, 12, ... van a ser siempre divisibles por 2, es decir, pares. Si hemos realizado la optimización propuesta en el punto anterior de sólo examinar los números impares para comprobar si son primos, estos no van a ser nunca divisibles por 4, 6, 8, 10, 12, ... con lo que podemos abreviar el bucle para la variable j cambiándolo por:

  Realizar un bucle j=3 hasta $i - 1$ con incremento de 2

3.3. Funciones

Ejercicio 9:

Ampliar la calculadora del ejercicio 3 de manera que devuelva el factorial de los dos números introducidos. Para ello debes definir una función:

```
int factorial(int x);
```

Que devuelva el resultado de calcular $x!$. Esta función se invocará cuando sea necesaria.

Se debe realizar primero una implementación iterativa del factorial y probar el programa. Después debe cambiarse la implementación del factorial por una versión recursiva del mismo y volver a probar el programa.

Ejercicio 10:

Modificar el programa que realiza la criba de Eratóstenes como se ha especificado en un ejercicio anterior, pero esta vez, todo el código correspondiente al cálculo de la criba debe incluirse en una función que se invocará desde el programa principal después de preguntar al usuario el número hasta el que se quieren calcular los primos.

Ejercicio 11:

Realizar un programa que imprima el código binario correspondiente a un número entero que el usuario introduzca por teclado en notación decimal.

Escribe el código necesario para obtener el código binario de un número decimal en una función que se invoque desde el programa principal. Esta función que calcula el código binario se puede realizar de manera iterativa (para la cual será aconsejable usar un vector) o recursiva, se deja a la elección del alumno de cual de las dos maneras quiere implementarla.

Ejercicio 12:

Realizar un programa que calcule el área y el perímetro de una circunferencia, en función del radio leído desde teclado. Escribe dos funciones distintas, una para el cálculo del área y otra para el perímetro. Recuerda que:

$$\begin{aligned} A &= \pi r^2 \\ C &= 2\pi r \end{aligned}$$

Ejercicio 13:

Realizar un programa para determinar si un carácter es uno de los dígitos del 0 al 9. El programa debe escribirse de tal forma que sirva para leer varios caracteres diferentes desde teclado.

3.4. Vectores

Ejercicio 14:

Realizar un programa que guarde en un vector 10 números introducidos por teclado, una vez introducidos, debe mostrarlos todos consecutivamente y el resultado de sumar los valores de los 10 números. Las dos últimas operaciones deben implementarse mediante el uso de funciones.

Después modificar el programa de manera que se cree un vector de 10 números aleatorios generados automáticamente y muéstrense al final todos consecutivamente así como el resultado de sumarlos todos.

Ejercicio 15:

Realiza un programa que:

1. Genere un vector de 20 enteros aleatorios, con valores entre 0 y 100.
2. Vaya pidiendo números enteros por teclado (hasta que se le introduzca el valor 1) y diga si el número aparece en el vector generado y, si es así, en qué posición.

La búsqueda de un elemento en un vector la podéis realizar mediante un algoritmo de búsqueda secuencial ascendente. Este algoritmo recorre cada uno de los elementos del vector, deteniéndose cuando encuentra al elemento buscado o bien cuando se llega al final del vector sin haberlo encontrado.

```
funcion BuscaSecAscen(v: Vector; x: Entero) devuelve boolean;
{
  pos:=0;
   mientras ((pos<N-1) y (v[pos] != x)) hacer
     pos:=pos+1;
   devolver(v[pos] = x)
}
```

El alumno debe realizar un programa bien estructurado con las funciones correspondientes.

Ejercicio 16:

Realizar un programa que genere un vector de tamaño máximo 50 elementos, con valores aleatorios entre 0 y 50. El usuario deberá indicar el tamaño del vector a considerar. Una vez generado el vector debe calcularse la media aritmética de los elementos del vector y mostrar por pantalla los elementos del vector y la media calculada. Realizar las funciones que sean necesarias para que el programa quede bien estructurado.

Ejercicio 17:

Realizar un programa que genere un vector de tamaño máximo 50 elementos, con valores aleatorios entre 0 y 100. El usuario deberá indicar el tamaño del vector a considerar.

Ordenar el vector de menor a mayor siguiendo el *algoritmo de la burbuja* que se describe a continuación:

```
funcion OrdenaBurbuja(v: Vector) devuelve Vector_ordenado;
{
   para i:=0 hasta N-1
     para j:=0 hasta N-1
       si (v[j-1]>v[j]) entonces
         aux:=v[j-1];
         v[j-1]:=v[j];
         v[j]:=aux;
}
```

Ejercicio 18:

En una empresa hay 5 secciones diferentes, cada una de las cuales tiene un número indeterminado de trabajadores. Se pide realizar un programa que vaya leyendo desde teclado la siguiente información de cada trabajador: *nombre, apellidos y horas trabajadas* (el máximo número admisible de horas trabajadas será de 50 h., en caso de introducir un número mayor debe ser rechazado e introducido de nuevo indicándose el motivo).

Con esta información se debe calcular la nómina de cada trabajador teniendo en cuenta las siguientes consideraciones:

- Las 30 primeras horas se pagan a 6 euros/hora.
- Las siguientes 10 horas se pagan a 9 euros/hora.
- Las restantes 10 horas se pagan a 12 euros/hora.
- Sobre el sueldo resultante se aplican los siguientes impuestos:
    - Hasta los primeros 180 euros, un 10 %.
    - Al resto del sueldo un 15 %.

Para cada trabajador se deberá imprimir Nombre y Apellidos, el sueldo bruto, los impuestos y el sueldo neto. Además, debe decirse qué sección ha pagado más impuestos, así como la que más horas ha trabajado.

El alumno debe crear las funciones y procedimientos necesarios para que el programa esté convenientemente estructurado.

3.5. Punteros y Memoria dinámica

Ejercicio 19:

Realiza un procedimiento:

```
void intercambio(int *x, int *y)
```

Que intercambie los valores de las variables x e y.

Desde el programa principal, debe preguntarse al usuario dos valores numéricos y guardarlos en dos variables, posteriormente deben intercambiarse los valores de las variables con el procedimiento definido arriba y mostrar los valores de las variables. Un ejemplo de la ejecución del programa sería:

```
Dime un valor para la a: 4
Dime un valor para la b: 7

Intercambiando a y b

Valor de a: 7
Valor de b: 4
```

Ejercicio 20:

Suponiendo que en nuestro programa tenemos las siguientes definiciones de tipos:

```
typedef struct s_punto {
   int x;
   int y;
} punto;

typedef struct s_rectangulo {
   punto p1;
   punto p2;
} rectangulo;
```

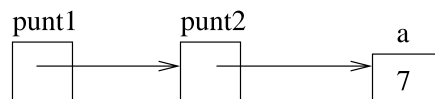
Realizar un programa que pida al usuario las coordenadas de un punto y las almacene en un punto (todo esto dentro de una función) y pida también las coordenadas de la esquina superior izquierda y la esquina inferior derecha

de un rectángulo (dentro de otra función). Después debe comprobar si el punto está dentro del rectángulo (otra función) y por último devuelva un mensaje al usuario diciendo si el punto está dentro o fuera del rectángulo.

Ejercicio 21:

Realizar un programa que pida un número entero al usuario, lo guarde en una variable entera y después actualice un puntero que señale a dicha variable y otro puntero que señale al puntero anterior. Por último debe imprimirse el valor del entero introducido inicialemnte, pero accediendo a dicho valor por medio de la doble indirección de punteros.

Por ejemplo, si tenemos la siguiente situación:



Se imprimirá el valor 7, pero accediendo a él a través de los punteros punt1 y punt2.

Ejercicio 22:

Realizar un programa que cree fichas de alumnos.

Para ello debe definirse un tipo de datos que sea un estructura que contenga el nombre del alumno/a, el número de expediente y la nota correspondiente al alumno (para una sola asignatura).

Debe implementarse una función que reserve memoria para una estructura del tipo descrito, pregunte los datos pertinentes al usuario para rellenar la ficha del alumno y por último devuelva la estructura creada al programa principal, el cual se ocupará de mostrar todos los datos recolectados al usuario y preguntar si son correctos, en caso de que sean correctos, debe liberarse la memoria reservada para la ficha y acabar el programa; si no son correctos, debe liberarse la memoria y volverse a crear una nueva ficha preguntando por los datos otra vez, así hasta que los datos que se introduzcan sean correctos.

Ejercicio 23:

Dados dos vectores de caracteres (strings) S y T, cuyos caracteres se encuentran en orden alfabético, hacer una función en lenguaje C que pasándole como argumento ambas cadenas, almacene en otro vector de caracteres la

cadena resultante al ordenar alfabéticamente todos los caracteres de las dos cadenas S y T.

Un ejemplo de la ejecución del programa sería:

```
Introduce la cadena S: mopt
Introduce la cadena T: ads

La cadena resultado es: admopst
```

Nota: introduce todos los caracteres de las cadenas en minúsculas (si no, puedes aparecer un orden alfabético un poco *raro* en el resultado).

3.6. Strings

Ejercicio 24:

Realizar un programa que utilice: una función para preguntar el nombre del usuario, otra función para preguntar el primer apellido y otra función más para preguntar el segundo apellido. Por último, las tres cadenas anteriores deben concatenarse en una sola cadena y mostrar el nombre completo al usuario.

Ejercicio 25:

Realizar un programa que lea una string y posteriormente calcule su longitud usando una función:

```
int longitud(char *s)
```

Esta función debe devolver el mismo resultado que la función `strlen()`.

Ejercicio 26:

Realizar un programa que solicite una cadena y después devuelva todos los prefijos de dicha cadena, esto es, todas las subcadenas que comienzan desde el principio de la cadena original. No deben considerarse los espacios en blanco como caracteres significativos, es decir la frase "*Hola*" y "*Hola* " se consideran la misma cadena.

Un ejemplo de la ejecución sería:

```
Introduce una cadena: El perro de San Roque

PREFIJOS:
E
El
El p
El pe
El per
El perr
El perro
El perro d
El perro de
El perro de S
El perro de Sa
El perro de San
El perro de San R
El perro de San Ro
El perro de San Roq
El perro de San Roqu
El perro de San Roque
```

Ejercicio 27:

Realizar un programa que pida al usuario 5 cadenas diferentes. Almacenar estas en una estrucutra en memoria, por ejemplo, un vector de cadenas. Por último, aplicar el algoritmo de ordenación propuesta en el ejercicio 17 para números enteros pero esta vez aplicarlo para ordenar cadenas alfabéticamente.

El orden establecido para ordenar cadenas (el >) se implementa en este caso con la función `strcmp`. El alumno debe consultar la ayuda en línea de Turbo-C para conocer el funcionamiento de esta función o releer el apartado de **Strings** de la práctica 2.

3.7. Manejo de ficheros

Ejercicio 28:

Crear un fichero de texto usando un editor de texto, por ejemplo, el editor de texto del Turbo-C. Escribir unas cuantas líneas en él. Este fichero de texto

se podrá usar en éste y los siguientes ejercicios.

Realizar después un programa que lea un fichero y muestre sus líneas por pantalla numerándolas, es decir escribiendo el número de línea al principio de cada línea que se muestre por pantalla.

El usuario deberá indicar el nombre del fichero a utilizar y el número de líneas que quiere leer del fichero.

Por ejemplo, si tenemos el siguiente fichero de texto llamado MUNDIAL.TXT:

```
El mundial de futbol
de 2045, lo ganara la
seleccion de Groenlandia
```

Una posible ejecución de nuestro programa podría ser:

```
cual es el fichero a leer? mundial.txt
cuantas lineas quieres leer? 2

1: El mundial de futbol
2: de 2045, lo ganara la
```

Ejercicio 29:

Realizar un programa que copie las m primeras líneas de un fichero en otro. El usuario deberá indicar los nombres de los dos ficheros (el de origen y el de copia) y el número de líneas a copiar (valor de m).

Ejercicio 30:

Realizar un programa que lea un fichero de texto con varias líneas y genere a partir de él otro fichero de texto con la extensión .INV de manera que este fichero de salida contenga las mismas líneas que el fichero de entrada pero en orden inverso, es decir la primera línea del fichero de entrada será la última del fichero de salida, la segunda línea del fichero de entrada será la penúltima del fichero de salida, etc.

Por ejemplo, si el fichero CUENTO.TXT contuviera las líneas:

```
Erase una vez
en un sitio muy lejano
que existia
un ogro llamado ... SHRECK
```

El programa devolvería como resultado el fichero CUENTO.INV con este contenido:

```
un ogro llamado ... SHRECK
que existia
en un sitio muy lejano
Erase una vez
```

Para conseguir este resultado podemos definir en nuestro programa una estructura de datos que sea un vector de cadenas de texto (strings). Cuando leamos el fichero original, iremos guardando cada línea del fichero de texto en una string, cuando acabemos de leerlo, sólo habrá que escribir en el fichero destino en orden inverso al de lectura.

Práctica 4

Introducción al diseño modular

4.1. Introducción al diseño modular

Cuando un problema adquiere una relativa complejidad, surge la necesidad de dividirlo en distintas tareas. Cada una de estas tareas se implementará en una unidad de compilación separada que denominamos módulo.

El problema original, pues, se divide en diversos subproblemas más pequeños, cada uno de ellos con una misión bien determinada dentro del marco general del proyecto, que interaccionan de manera clara y mínima, de tal forma que todos juntos representan la solución del problema inicial.

Un módulo ha de cumplir ciertos requisitos:

- *Independencia en el desarrollo*: las conexiones del módulo con el resto de módulos que representan la solución del problema deben de ser pocas y simples.
- *Facilidad de modificación del programa*: un cambio en el programa debe afectar mínimamente a los módulos en que queda dividido.
- *Tamaño adecuado*.

Las ventajas que aporta esta técnica de diseño son:

- Utilización del diseño descendente para la resolución de problemas (de lo más general se va descendiendo hasta los detalles mínimos de implementación).
- Protección de las estructuras de información que utilizaremos para la resolución del problema.
- Posibilidad de reutilización de módulos para resolver otros problemas.

4.2. Listas de importación/exportación

En la implementación de los distintos módulos que componen la solución de un problema se requerirá la interactuación de éstos, de tal forma que un módulo puede que necesite tipos y/o procedimientos implementados en otros módulos.

Por otro lado, por razones de seguridad, será conveniente que ciertos elementos propios de un módulo no puedan ser utilizados por otros. Este tipo de requisitos deberá ser expresado de manera explícita mediante las denominadas listas de importación y exportación, que aparecerán en la implementación de cada módulo.

Mediante la lista de exportación se establecerán los elementos (tipos, constantes, funciones y procedimientos) del módulo que son visibles desde el exterior. Mediante la lista de importación se designan los nombres de todos aquellos módulos de los cuales se utilizarán elementos, de los que sólo podrán ser utilizados los que figuren en la lista de exportación del módulo importado.

4.3. Sintaxis de un módulo

La sintaxis de un módulo constará de las siguientes partes:

- *Cabecera*: nombre del módulo.
- *Lista de importación*: nombre de los módulos de los cuales utilizamos tipos, constantes, procedimientos y funciones.
- *Lista de exportación*: lista de todos los elementos que podrán ser utilizados por otros módulos. En la lista aparecerá la cabecera de los procedimientos y/o funciones que se exporten (no su implementación) y los nombres de los tipos y constantes que se exporten (no su declaración).
- *Implementación*: declaración de tipos, constantes y variables propias del módulo. Implementación de funciones y procedimientos, tanto de los que se exportan por el módulo como de otros ocultos para otros módulos.

4.4. Utilización de módulos en C

La implementación y utilización de los módulos en C hace uso de dos tipos de ficheros:

- Fichero .c, con la implementación de las funciones y procedimientos del módulo, tanto de los que se exportan como de los ocultos para otros módulos.
- Fichero .h, denominada librería, con la declaración de los tipos, constantes y variables que se deseen exportar. También incluye la cabecera de los procedimientos y/o funciones que se exporten.

En el fichero .c se deberá incluir, al principio, una sentencia de inclusión de la librería .h con la declaración de tipos, constantes y funciones de nuestro fichero. La sentencia deberá seguir esta estructura:

```
#include "nombre_fichero.h"
```

Si otros ficheros .c desean hacer uso de las funciones proporcionadas por el módulo implementado (las funciones que exportan), deberán incluir la librería que hace referencia al módulo. Por tanto, en la sección de includes se deberá escribir:

```
#include "nombre_fichero.h"
```

siendo `nombre_fichero` el nombre de la librería de módulo creado.

4.5. Compilación de módulos en Turbo C

Una vez que se han generado los ficheros .c y .h que nos interesen utilizar, lo primero que hay que hacer cuando se quieren emplear módulos en Turbo C es crear un proyecto:

1. En **File|Change dir**, indica el directorio en el que guardarás tu proyecto. Normalmente es el directorio en el que se encuentren los ficheros .c y .h a utilizar.
2. En **Project|Open project**, dale un nombre al nuevo proyecto a crear. Se mostrará una ventana con el proyecto y la relación de ficheros que contiene (al crearlo, la lista estará vacía).
3. Con **Project|Add item**, incluir los ficheros .c que vayamos a utilizar.
4. Situándonos sobre cada uno de los ficheros incluidos, en la ventana de proyecto, comprobamos que las librerías que emplean son correctas con: **Project|Include Files**. Deberán aparecer todos los .h que hayamos definido en ese fichero.

5. Seleccionamos **Compile|Make** para compilar todo el proyecto (los ficheros .c con sus librerías). Si se producen errores, deberemos solucionarlos y volver a repetir la compilación con Make, hasta que sea exitosa.

6. Seleccionamos **Run|Run** para ejecutar el programa completo.

4.6. Ejemplo de programa dividido en módulos

En este ejemplo, se tiene un fichero .c, denominado `fich.c`, que implementa algunas funciones de manejo de ficheros. Estas funciones son exportadas gracias a la librería .h (`fich.h`), en la que se definen sus cabeceras.

Otro fichero .c, denominado princip.c, hace uso de las funciones de manejo de ficheros, gracias a la inclusión de la librería fich.h.

Fichero fich.h:

```
/* Libreria que exporta las funciones del modulo fich.c */
#include <stdio.h>

/* FUNCIONES EXPORTADAS */

void copiar_fichero(FILE *in, FILE *out);

FILE *abrir_fichero(char *fich, char *modo);
```

Fichero fich.c:

```
/* Modulo que implementa funciones de uso de ficheros */
#include <stdio.h>
#include <stdlib.h>
#include "fich.h"

void copiar_fichero(FILE *in, FILE *out) {
   while (!feof(in)) putc(getc(in), out);
}

FILE *abrir_fichero(char *fich, char *modo) {
   FILE *aux;

   if ((aux = fopen(fich, modo)) == NULL) {
printf("Cannot open file %s.\n",fich);
        exit(1);
   }
   return(aux);
}
```

Fichero princip.c:

```
/* Programa principal que hace uso del modulo de manejo de ficheros */
#include "fich.h"

int main(void) {
   FILE *in, *out;
   char fich[32];

   printf("Fichero origen:");
   gets(fich);
   in = abrir_fichero(fich,"r");
   printf("Fichero destino:");
   gets(fich);
   out = abrir_fichero(fich,"w");
   copiar_fichero(in,out);
   printf("\n FIN de la copia\n");
   fclose(in);
   fclose(out);
   return(0);
}
```

Práctica 5

Evaluación de expresiones aritméticas

5.1. Introducción

Nuestro objetivo es construir un programa evaluador de expresiones artiméticas simples, para ello veremos que es necesario utilizar el TAD pila, con lo que se espera que el alumno sepa aplicar correctamente los conceptos vistos en clase de teoría.

Algunas de las técnicas utilizadas en el diseño de compiladores pueden utilizarse a pequeña escala en la implementación de una calculadora de bolsillo típica. Las calculadoras evalúan lo que se conoce como **expresiones infijas** (o notación infija), como por ejemplo $1 + 2$, que consisten en un operador binario (la suma, $+$) junto con operandos (argumentos) a su izquierda y su derecha (el 1 y el 2). Este formato, bastante fácil de evaluar en un principio, puede hacerse más complejo. Considérese la expresión:

$$1 + 2 * 3$$

Matemáticamente, la expresión se evalúa a 7, porque el operador de multiplicación tiene mayor precedencia que la suma. Sin embargo, algunas calculadoras con otro esquema de evaluación podrían dar como respuesta 9. Esto ilustra que una simple evaluación de izquierda a derecha no es suficiente, ya que no podemos empezar evaluando $1 + 2$. Considérese ahora la expresión:

$$10 - 4 - 3$$

¿Qué resta debe evaluarse primero? Las restas se evaluarán de izquierda a derecha, dando como resultado 3. Es decir, la resta, a igualdad de precedencia, asocia de izquierda a derecha; supondremos que con el resto de operadores

que vamos a usar ocurrirá lo mismo, es decir, a igualdad de precedencia entre operadores se asocia de izquierda a derecha. Con todo esto, la evaluación de la siguiente expresión se vuelve bastante compleja:

$$1 - 2 - 4/5 * (3 * 6)/7$$

Para eliminar la posible ambigüedad de la expresión anterior podemos introducir paréntesis para ilustrar el orden correcto de los cálculos:

$$(1 - 2) - (((4/5) * (3 * 6))/7)$$

Aunque los paréntesis desambiguan el orden de evaluación, resulta difícil decir que hacen más claro el mecanismo de evaluación. Sin embargo, existe una notación para expresiones diferente, denominada **notación postfija**, que proporciona un mecanismo directo de evaluación. Las siguientes secciones muestran cómo funciona. En la sección 5.2 estudiaremos la notación postfija, mostrando cómo las expresiones escritas con esta notación pueden evaluarse con un simple recorrido de izquierda a derecha. La sección 5.3 muestra cómo las expresiones originales, que utilizan la notación habitual infija, se pueden convertir a notación postfija. De esta manera podremos construir un sistema de evaluación de expresiones aritméticas infijas en dos pasos. En primer lugar transformaremos la expresión infija a la expresión postfija correspondiente y en segundo lugar evaluaremos esta última expresión.

Antes de pasar a la implementación del sistema, se presenta un ejemplo completo al alumno en la sección 5.4. Por ello, finalmente, se presenta un programa a completar por el alumno, que evalúa expresiones infijas que contengan operadores aditivos, multiplicativos y paréntesis. Se utilizará un algoritmo de evaluación de expresiones con precedencia entre operadores.

5.2. Expresiones postfijas

Una expresión postfija está formada por una serie de operadores y operandos, donde un operador va precedido por sus operadores. Por ejemplo, para una de las expresiones infijas mostradas en la sección anterior:

$$1 + 2 * 3$$

la expresión postfija equivalente sería

$$1\ 2\ 3 * +$$

donde el 2 y el 3 preceden al operador *, puesto que son sus operandos, y el 1 y el resultado de 2 3$*$ preceden al operador $+$.

Se evalúa de la siguiente forma: cuando se encuentra un operando, se apila en la pila; cuando se encuentra un operador, el número apropiado de operandos son desapilados de la pila, se evalúa la operación indicada por el operador, y el resultado se apila de nuevo en la pila.

Para operadores binarios, que son los más comunes (y los únicos que vamos a tratar en esta práctica), se desapilan dos operandos. Ejemplos de operadores binarios son la suma (+), la resta (−), la multiplicación (*) y la división (/).

Cuando la expresión postfija ha sido procesada completamente, el único elemento que quede en la pila será el resultado de la evaluación, es decir, el valor equivalente a evaluar la expresión postfija. La notación postfija es una forma natural de evaluar expresiones, pues con ella no son necesarias reglas de precedencia ni existen ambigüedades en la evaluación.

Un pequeño ejemplo de cómo se evalúa una expresión postfija se muestra en la figura 5.1.

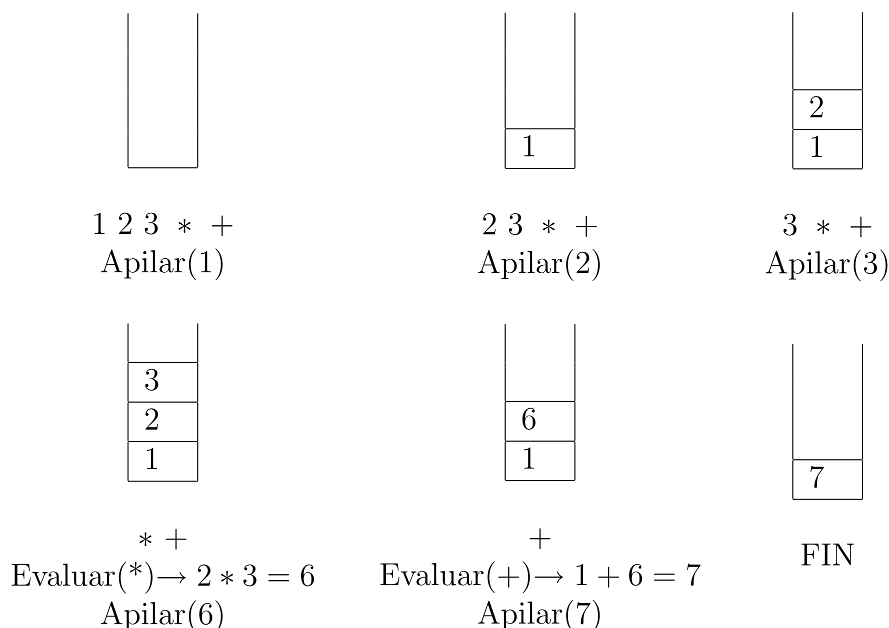


Figura 5.1: Ejemplo de evaluación de la expresión postfija 1 2 3 ∗ +, correspondiente a la expresión infija $1 + 2 * 3$. Se muestra como se va realizando el análisis de la expresión paso a paso. Para cada símbolo que se analiza se muestra la acción a realizar así como el estado de la pila.

La evaluación procede de la siguiente manera: el 1, el 2 y el 3 son apilados, en ese orden en la pila. Para procesar el *, se desapilan los dos elementos superiores de la pila: esto es, el 3 y después el 2. Obsérvese que el primer elemento

desapilado se convierte en el operando derecho del operador, y el segundo elemento desapilado en el operando izquierdo; por tanto, los parámetros se obtienen de la pila en orden inverso al natural. Para la multiplicación, esto no importa, pero para la resta y la división, desde luego que sí. El resultado de la multiplicación es 6, que es apilado en la pila. En este momento, la cima de la pila es un 6, y debajo hay un 1. Para procesar el +, se desapilan el 6 y el 1, y su suma, 7, se apila. En este punto, la expresión se ha leído completamente y la pila sólo tiene un elemento. Por tanto, la respuesta final a la evaluación de la expresión es 7.

Como hay 3 operandos y 2 operadores, habrá 5 pasos y 5 apilamientos para evaluar la expresión. Así pues, la evaluación de una expresión postfija requiere un tiempo lineal.

El punto que falta por estudiar para realizar la evaluación de expresiones infijas es un algoritmo que convierta notación infija en notación postfija. Una vez tengamos uno, tendremos un algoritmo para evaluar expresiones infijas.

5.3. Conversión de notación infija a postfija

El principio básico para convertir una expresión infija a una postfija es el siguiente: cuando se encuentra un operando en la cadena de entrada podemos pasarlo directamente a la cadena de salida; sin embargo, cuando encontramos un operador, no podemos pasarlo todavía a la salida, pues debemos esperar a encontrar su segundo operando. Por consiguiente, debemos guardarlo temporalmente en una estructura adecuada. Si consideramos una expresión como:

$$1 + 2 * 3/4$$

la expresión correspondiente en notación postfija es:

$$1\ 2\ 3 * 4/+$$

observamos que en ocasiones los operadores pueden aparecer en orden inverso a como aparecían en la expresión infija. Por supuesto, esto sólo es cierto si la precedencia de los operadores involucrados crece al ir de izquierda a derecha. Aún así, el hecho comentado sugiere que una pila es la estructura adecuada para almacenar los operadores pendientes. Siguiendo esta lógica, cuando encontremos un operador, debe ser colocado de alguna manera en la pila.

Consideramos ahora como ejemplo otra expresión infija más simple:

$$2 * 5 - 1$$

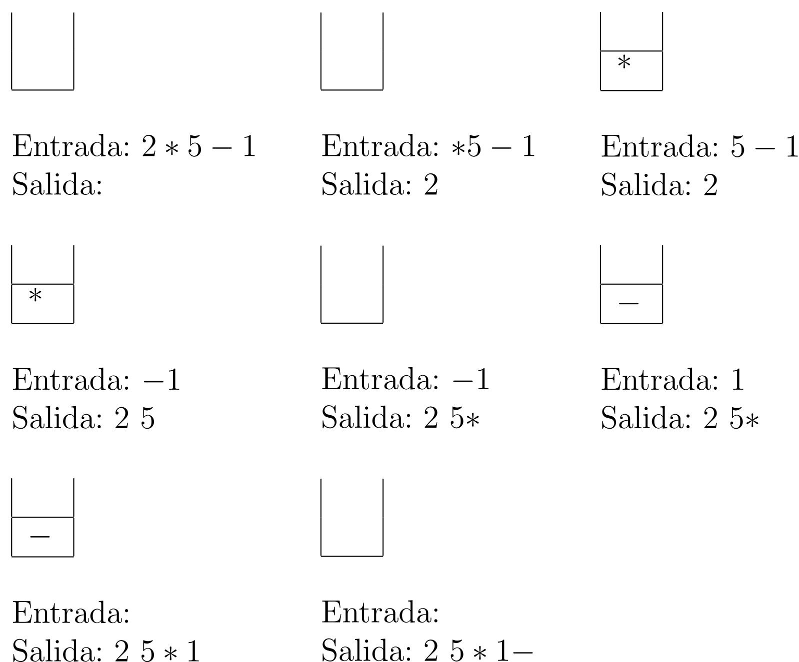


Figura 5.2: Ejemplo de transformación de la expresión infija $2 * 5 - 1$, a la expresión postfija correspondiente $2\ 5 * 1-$. Se muestra como se va realizando la transformación de la expresión paso a paso. Para cada símbolo de la entrada que se procesa se muestra como va quedando la salida y el estado de la pila.

Su transformación de forma infija a postfija se muestra en la figura 5.2.

Cuando alcanzamos el operador $-$, el 2 y el 5 se han incorporado ya a la salida, y * está en la pila. Ya que $-$ tiene menor precedencia que *, * tiene que aplicarse al 2 y al 5. Por lo que debemos desapilar * y, en general, cualquier otro operador que hubiese en la cima de la pila con mayor precedencia que $-$. En consecuencia la expresión resultante postfija es:

$$2\ 5 * 1-$$

En general, cuando procesamos un operador de la entrada, debemos sacar de la pila aquellos operadores que deban ser procesados atendiendo a las reglas de precedencia y asociatividad.

Antes de resumir el algoritmo, deben responderse algunas preguntas. En primer lugar, si el símbolo actual es un + y la cima de la pila es un +, ¿debería desapilarse el + de la pila, o debería quedarse ahí? La respuesta se obtiene decidiendo si el + encontrado en la entrada nos indica que el + de la

pila cuenta ya con sus operandos. Ya que el + asocia de izquierda a derecha, la respuesta es afirmativa.

¿Qué ocurre con los paréntesis? un paréntesis izquierdo puede considerarse como un operador de máxima precedencia cuando está en la entrada, pero de precedencia mínima cuando está en la pila. En consecuencia, el paréntesis izquierdo de la entrada se apilará siempre sin más. Cuando encontremos un paréntesis derecho en la entrada, desapilaremos hasta encontrar el correspondiente paréntesis izquierdo. Por otra parte, en la cadena de salida no aparecen paréntesis. En la sección 5.4 se muestra un ejemplo de transformación que ilustra todos los aspectos expuestos.

Presentamos ahora un resumen de los diferentes casos del algoritmo de transformación de expresiones infijas a expresiones postfijas. En cada paso se supone que estamos consultando el primer elemento de la expresión infija de entrada. Todo lo que se desapila se concatena a la expresión postfija de salida, a excepción de los paréntesis. Los casos son:

- **Operandos**: pasan directamente a la salida.
- **Paréntesis derecho**: desapilar símbolos hasta encontrar un paréntesis izquierdo.
- **Operador**: Desapilar todos los símbolos hasta que encontremos un símbolo de menor precedencia. Apilar entonces el operador encontrado.
- **Fin de la entrada**: desapilar el resto de símbolos en la pila.

5.4. Un ejemplo completo

Ahora ya conocemos el mecanismo de transformación de una expresión infija a forma postfija (visto en la sección 5.3) y el mecanismo de evaluación de una expresión postfija (visto en la sección 5.2). Así pues, ya estamos en disposición de crear un sistema que haciendo uso de las dos técnicas pueda evaluar expresiones aritméticas sencillas. En esta práctica solo trataremos los operadores de suma (+), resta (−), multiplicación (*) y división (/), así como el uso de paréntesis.

Pero antes de pasar a la implementación del sistema ofrecemos a continuación un ejemplo que ilustra los algoritmos mostrados en las secciones anteriores.

El objetivo es evaluar la expresión infija:

$$1 - 2 - 4/5 * (3 * 6)/7$$

Para ello, el primer paso será transformarla a la expresión postfija correspondiente :

$$1\ 2\ -\ 4\ 5\ /\ 3\ 6\ *\ \ *\ 7\ /\ -$$

Para cada símbolo de la entrada analizado se muestra lo que queda por escanear de la cadena de entrada, el estado de la pila de operadores y la cadena que se ha obtenido hasta este momento en la salida.

$-$

Ent: $-2-4/5*(3*6)/7$
Sal: 1

$-$

Ent: $2-4/5*(3*6)/7$
Sal: 1

$-$

Ent: $-4/5*(3*6)/7$
Sal: 1 2

Ent: $-4/5*(3*6)/7$
Sal: 1 2$-$

$-$

Ent: $4/5*(3*6)/7$
Sal: 1 2$-$

$-$

Ent: $/5*(3*6)/7$
Sal: 1 2 $-$ 4

$/$
$-$

Ent: $5*(3*6)/7$
Sal: 1 2 $-$ 4

$/$
$-$

Ent: $*(3*6)/7$
Sal: 1 2 $-$ 4 5

$-$

Ent: $*(3*6)/7$
Sal: 1 2 $-$ 4 5/

$*$
$-$

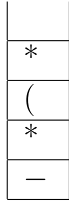
Ent: $(3*6)/7$
Sal: 1 2 $-$ 4 5/

(
$*$
$-$

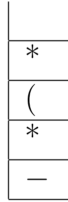
Ent: $3*6)/7$
Sal: 1 2 $-$ 4 5/

(
$*$
$-$

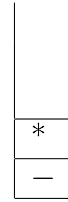
Ent: $*6)/7$
Sal: 1 2 $-$ 4 5/3



Ent: 6)/7
Sal: 1 2 − 4 5/3

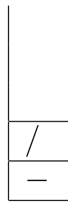


Ent:)/7
Sal: 1 2 − 4 5/3 6

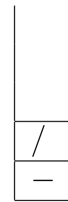


Ent: /7
Sal: 1 2 − 4 5/3 6*

Ent: /7
Sal: 1 2 − 4 5/3 6 * *

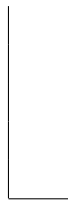


Ent: 7
Sal: 1 2 − 4 5/3 6 * *



Ent:
Sal: 1 2 − 4 5/3 6 * *7

Ent:
Sal: 1 2 − 4 5/3 6 * *7/



Ent:
Sal: 1 2 − 4 5/3 6 * *7/−

Una vez hemos obtenido la expresión postfija

$$1\ 2 - 4\ 5/3\ 6 * *7/-$$

queda evaluarla conforme al algoritmo descrito en la sección 5.2. Los pasos seguidos en este algoritmo se muestran a continuación. Para cada símbolo de la expresión postfija procesado se muestra el estado de la pila. Cuando se haya terminado de procesar toda la expresión, el número que permanezca en la pila será el valor correspondiente a la expresión original.

(empty)	1	2 1
$1\ 2 - 4\ 5/3\ 6 * *7/-$	$2 - 4\ 5/3\ 6 * *7/-$	$-4\ 5/3\ 6 * *7/-$
-1	4 -1	5 4 -1
$4\ 5/3\ 6 * *7/-$	$5/3\ 6 * *7/-$	$/3\ 6 * *7/-$
0.8 -1	3 0.8 -1	6 3 0.8 -1
$3\ 6 * *7/-$	$6 * *7/-$	$* * 7/-$
18 0.8 -1	14.4 -1	7 14.4 -1
$*7/-$	$7/-$	$/-$
2.057 -1	-3.057	
$-$		

5.5. Implementación

El programa a implementar, realizará las tareas de leer la expresión aritmética de entrada, después utilizará el algoritmo descrito en la sección 5.3 para transformar la expresión aritmética infija a postfija y por último utilizará el algoritmo descrito en la sección 5.2 para evaluar la expresión postfija y devolver el valor final al usuario.

Para implementar los algoritmos propuestos en las secciones 5.3 y 5.2 será necesario utilizar dos pilas diferentes:

- Una pila para operadores, utilizada en la transformación de expresión infija a expresión postfija (primera fase) y que realmente se implementará como una pila de enteros, puesto que definiremos un nuevo tipo de datos `Toperador` que se corresponderá con el tipo de datos entero.
- Otra pila para operandos, utilizada en la evaluación de la expresión postfija (segunda fase) y que realmente se implementará como una pila de números reales (`float`), puesto que definiremos un nuevo tipo de datos `Toperando` que se corresponderá con el tipo de datos real.

Para las dos pilas descritas arriba se usará un representación vectorial como la vista en clase, es decir, una estructura (registro) que contiene un vector donde se almacenan los elementos de la pila y una variable `tope` que indica donde está el tope de la pila (para más información, ver apuntes de teoría).

Ejercicio: Completar el código de las funciones `crearp_es()`, `apilar_es()`, `desapilar_es()`, `tope_es()` y `vaciap_es()` para la pila de operadores y las funciones `crearp_os()`, `apilar_os()`, `desapilar_os()`, `tope_os()` y `vaciap_os()` para la pila de operandos. Este código es muy *similar* para los dos tipos de pila. Además el alumno puede basarse en los apuntes de clase.

Otro aspecto importante para la implementación del programa es cómo se va a representar la expresión aritmética a evaluar. Inicialmente el usuario deberá introducir la expresión infija por teclado, con lo que se almacenará en una string (vector de caracteres). Pero esta representación interna no es muy manejable a la hora de conocer cuáles son los operadores y operandos de la expresión para evaluarla, por ello, transformaremos la string inicial a un formato más manejable por el computador. Supongamos que tenemos la siguiente string que representa la expresión infija $(15 * 2) + 1$:

0	1	2	3	4	5	6	7
(	1	5	*	2	)	+	1

Vamos a representarla mediante un vector de enteros donde cada operador y cada operando estará plenamente identificado y será fácilmente manipulable.

Para ello, sustituiremos los números enteros formados por caracteres (tanto por un carácter como por varios) en un valor entero que ocupará sólo una de las posiciones del vector que va a representar la expresión. En esta práctica no vamos a tratar con números negativos, por lo que sólo debemos considerar como números las secuencias de uno o más caracteres consecutivos formadas por dígitos.

Vamos a aprovechar la cualidad de que no existirán números negativos para representar los operadores mediante números menores de 0 (ya que así no se podrán confundir nunca con un operando). Para ello se han definido una serie de constantes como `SUMA`, `RESTA`, etc. en el fichero `aritmet.c` (ver código en la sección 5.5) que en el código del programa se utilizarán para representar a los operadores. Considerando los valores asignados a estas constantes en el código, el vector de enteros que representará la expresión aritmética anterior sería:

0	1	2	3	4	5	6
-5	15	-3	2	-6	-1	1

Esta transformación de string a vector de números enteros la realiza en su totalidad la función `tokenize()` que ya se facilita escrita al alumno.

Otro aspecto a tratar es cómo asignar el valor de precedencia que tiene cada operador, para ello se ha implementado una función `precedencia()` que recibe como argumento un operador y devuelve la precedencia correspondiente como un valor entero. A mayor valor, mayor precedencia. La suma y la resta tienen igual precedencia, la multiplicación y la división también. El paréntesis izquierdo tiene la menor precedencia de todos los operadores para que el tratamiento de la pila de operadores sea como se explicó en la sección 5.3. El paréntesis derecho no necesita ningún valor de precedencia puesto que no se va a apilar nunca. La función que retorna la precedencia de un determinado operador se proporciona ya implementada.

Otra función necesaria para nuestro programa, que es la encargada de transformar la expresión aritmética de forma infija a forma postfija, también se proporciona implementada. Además el alumno puede *basarse* un poco en esta función para implementar la que se le solicitará más adelante. Esta función devuelve un vector de enteros que contiene la expresión en forma postfija con el mismo tipo de representación que se usaba para la infija, si una posición es un entero positivo, entonces representa un operando, sino, es un operador.

El programa principal es muy sencillo, en primer lugar lee de teclado la expresión aritmética y la guarda en una string, en segundo lugar la transforma utilizando la función `tokenize()` explicada antes; después la transforma de forma infija a forma postfija utilizando la función `transforma_postfija()` igualmente explicada antes; luego, evalúa la expresión en forma postfija con la función `evalua_postfija()` que deberá implementar el alumno, y, por último muestra el resultado final al usuario.
Ejercicio: Escribir el código necesario de la función `evalua_postfija()` para evaluar una expresión postfija como se ha explicado en la sección 5.2. Después de completar esta función, el programa `aritmet.c` ya podrá ejecutarse y comprobar si se evaluan correctamente las expresiones aritméticas.

5.6. Códigos fuente para la sección 5.5

Código de aritmet.c:

```
/* PROGRAMA PARA EVALUAR EXPRESIONES ARITMETICAS */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

/*****************************************************/
/****** CONSTANTES ******/

#define MAXCAD 300 /* tamanyo maximo de una cadena */
#define MAXSIM 100 /* numero maximo de simbolos en una expresion */
#define maxP 100   /* tamanyo maximo de una pila */

/* definimos los simbolos artimeticos */
#define SUMA    -1
#define RESTA   -2
#define MULT    -3
#define DIV     -4
#define PAR_IZQ -5
#define PAR_DER -6

/*****************************************************/
/****** TIPOS DE DATOS ******/

typedef int Toperador;
typedef float Toperando;

/* Tipo de datos para la pila de operadores */
typedef struct {
Toperador v[maxP];
int tope;
} TPila_es;

/* Tipo de datos para la pila de operandos */
typedef struct {
Toperando v[maxP];
int tope;
} TPila_os;
```

```
/*********************************************************/
/* FUNCIONES PARA LA PILA DE OPERADORES */

TPila_es *crearp_es() {         /* ..... */
}
/* ----------------------------------------- */
TPila_es *apilar_es(TPila_es *p, Toperador e) {        /* ..... */
}
/* ----------------------------------------- */
TPila_es *desapilar_es(TPila_es *p) {        /* ..... */
}
/* ----------------------------------------- */
Toperador tope_es(TPila_es *p) {        /* ..... */
}
/* ----------------------------------------- */
int vaciap_es(TPila_es *p) {        /* ..... */
}


/*********************************************************/
/* FUNCIONES PARA LA PILA DE OPERANDOS */

TPila_os *crearp_os() {        /* ..... */
}
/* ----------------------------------------- */
TPila_os *apilar_os(TPila_os *p, Toperando e) {        /* ..... */
}
/* ----------------------------------------- */
TPila_os *desapilar_os(TPila_os *p) {        /* ..... */
}
/* ----------------------------------------- */
Toperando tope_os(TPila_os *p) {        /* ..... */
}
/* ----------------------------------------- */
int vaciap_os(TPila_os *p) {        /* ..... */
}
```

```
/*******************************************************/
/* Funcion para transformar la string de entrada a */
/* una secuencia de simbolos. */

void tokenize(char *cadena, int *expresion_infija,
                                       int *num_sim) {
  int i,j;
  char sim[MAXCAD];

  *num_sim = 0;
  i = 0;
  while (i<strlen(cadena)) {
    switch (cadena[i]) {
      case '+': expresion_infija[*num_sim]=SUMA; break;
      case '-': expresion_infija[*num_sim]=RESTA; break;
      case '*': expresion_infija[*num_sim]=MULT; break;
      case '/': expresion_infija[*num_sim]=DIV; break;
      case '(': expresion_infija[*num_sim]=PAR_IZQ; break;
      case ')': expresion_infija[*num_sim]=PAR_DER; break;
      default: j=i;
               strcpy(sim,"");
               while ((j<strlen(cadena))&&(cadena[j]>='0')
                                          &&(cadena[j]<='9')) {
                          sim[j-i]=cadena[j];
                          j++;
               }
               sim[j-i]='\0';
               i=j-1;
               expresion_infija[*num_sim]=atoi(sim);
    }
    (*num_sim)++;
    i++;
  }
}
```

```
/*****************************************************/
/* Funcion para devuelve el grado de precedencia de un */
/* operador. */

int precedencia(Toperador op) {
  int p;
  switch (op) {
     case SUMA: p=1; break;
     case RESTA: p=1; break;
     case MULT: p=2; break;
     case DIV: p=2; break;
     case PAR_IZQ: p=0; break;
  }
  return(p);
}


/*****************************************************/
/* Funcion para transformar la secuencia de entrada en forma */
/* infija a postfija. */

void transforma_postfija(int *exp_infija,
                       int *exp_postfija, int *num_sim) {
  TPila_es *pila_es;    /* Pila de operadores. */
  int i;
  int op;
  int num_sim_out;  /* Numero de simbolos que quedaran en la salida. */

  pila_es = crearp_es();
  num_sim_out = 0;
```

```
for (i=0;i<*num_sim;i++) {
    switch (exp_infija[i]) {
        case SUMA:
        case RESTA:
        case MULT:
        case DIV: op=tope_es(pila_es);
                  while ((precedencia(op)>=precedencia(exp_infija[i])) &&
                         (!vaciap_es(pila_es))) {
                      pila_es=desapilar_es(pila_es);
                      exp_postfija[num_sim_out]=op;
                      num_sim_out++;
                      op=tope_es(pila_es);
                  }
                  pila_es=apilar_es(pila_es, exp_infija[i]);
                  break;
        case PAR_IZQ: pila_es=apilar_es(pila_es, PAR_IZQ);
                      break;
        case PAR_DER: op=tope_es(pila_es);
                      pila_es=desapilar_es(pila_es);
                      if (op!=PAR_IZQ) {
                          exp_postfija[num_sim_out]=op;
                          num_sim_out++;
                      }
                      while (op!=PAR_IZQ) {
                          op=tope_es(pila_es);
                          pila_es=desapilar_es(pila_es);
                          if (op!=PAR_IZQ) {
                              exp_postfija[num_sim_out]=op;
                              num_sim_out++;
                          }
                      }
                      break;
        default: /* Es un operando. */
                 exp_postfija[num_sim_out] = exp_infija[i];
                 num_sim_out++;
    }
}
```

```
  /* Vaciamos los posibles operadores que queden en la pila. */
  while (!vaciap_es(pila_es)) {
      op=tope_es(pila_es);
      pila_es=desapilar_es(pila_es);
      exp_postfija[num_sim_out]=op;
      num_sim_out++;
  }
  *num_sim = num_sim_out;
}


/**********************************************************/
/* Funcion para evaluar una expresion postfija. */

Toperando evalua_postfija(int *exp_postfija, int num_sim) {
  TPila_os *pila_os;     /* Pila de operandos. */
  int i;
  Toperando op1, op2;

  /* INSERTAR CODIGO PARA EVALUAR LA EXPRESION POSTFIJA */

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

```
/*********************************************************/
/******************** PROGRAMA PRINCIPAL ****************/
/*********************************************************/
int main(){
  char cadena[MAXCAD];  /* Para leer la expresion. */
  int num_sim;    /* Numero de simbolos en la expresion. */
  int expresion_infija[MAXSIM];  /* Guarda los simbolos de la */
                                 /* expresion infija. */
  int expresion_postfija[MAXSIM]; /* Guarda los simbolos de la */
                                  /* expresion postfija. */
  Toperando resultado;

  clrscr();
  printf("Introduce la expresion aritmetica: ");
  gets(cadena);

  tokenize(cadena, expresion_infija, &num_sim);

  transforma_postfija(expresion_infija, expresion_postfija, &num_sim);

  resultado = evalua_postfija(expresion_postfija, num_sim);
  printf("\n\nRESULTADO: %f\n",resultado);

  printf("Pulsa Intro para acabar\n");
  gets(cadena);
  return(0);
}
```

Práctica 6

Estructurando programas

En esta práctica recordaremos como construir un programa estructurado en diversos módulos. De ahora en adelante se espera que el alumno sea capaz de estructurar correctamente los ejercicios de las prácticas en módulos para conseguir una mejor organización y limpieza del código.

Aparte de la estructuración en varios módulos veremos como utilizar ficheros para ahorrar tiempo en la lectura de datos por parte de los programas y conseguir mayor comodidad para repetir experimentos y testear programas o algoritmos utilizando los mismos datos de entrada en cada ejecución.

Veremos primero un ejemplo basado en una lista de números enteros y en segundo lugar un ejemplo idéntico al primero pero utilizando fichas de alumnos en vez de números enteros simples. El alumno deberá completar parte este último ejemplo.

6.1. Listas de números enteros

El objetivo de este ejercicio va a ser conseguir que nuestro programa lea una serie de números de un fichero y cree una lista con ellos; después debe imprimir por pantalla dicha lista de números y terminar.

También vamos a realizar dos versiones del programa, una versión utilizando la representación vectorial de listas vista en clase y otra versión utilizando la representación enlazada con variables dinámicas vista en clase igualmente. El código correspondiente a cada una de las dos representaciones estará en un módulo independiente con lo que el programa principal será completamente independiente de la implementación que se haya realizado para la lista. Los módulos construidos para las dos representaciones de listas constituyen lo que se conoce como librerías. En el programa principal se puede

optar por utilizar (incluir) la librería correspondiente a la representación vectorial o la librería correspondiente a la representación enlazada con variable dinámica.

El programa principal se denomina `prog1.c` y puede verse su código en la sección 6.3. También puede verse el código del módulo `lista_v.c` que implementa la representación vectorial de listas junto con su fichero de cabecera `lista_v.h` donde se especifican los tipos de datos, variables y funciones que se exportan al programa que incluya dicho modulo.

La inclusión de esta librería por parte del programa principal se realiza con la línea:

```
#include "lista_v.h"
```

al principio de `prog1.c`. Además como estamos trabajando con Turbo C, si queremos utilizar varios módulos para un programa deberemos crear lo que se conoce como un proyecto, en el cual se especificarán los ficheros de código C que forman el programa. En nuestro ejemplo los pasos a seguir son, una vez estemos dentro del entorno de Turbo C:

1. En `File` → `Change dir`, cambiar el directorio por defecto al directorio donde se encuentran los fuentes del programa y los módulos a utilizar.

2. En `Project` → `Open project`, especificar el nombre del fichero `.prj` que va a guardar el proyecto que vamos a crear. Por ejemplo `prog1.prj`. Esta acción abre una ventana donde se encuentra la lista de módulos que componen el proyecto, inicialmente vacía.

3. Ahora añadiremos todos los ficheros `.c` que se correspondan con los módulos utilizados por el programa principal así como el fichero que contiene el programa principal. Para ello, en `Project` → `Add item` añadir primero `lista_v.c` y después `prog1.c`, utilizando el boton de `Add` para cada uno de ellos. Para acabar pulsar `Done`.

4. Ahora ya podemos construir el ejecutable para nuestro programa con `Compile` → `Build All` o bien podemos ejecutarlo directamente con `Run` → `Run`. En el caso en que el programa contenga errores sintácticos, estos se nos mostrarán en una ventana de mensajes.

Para los ejemplos hay disponible un fichero `numeros` que contiene una serie de números enteros cada uno en una línea, con lo que podremos leerlos utilizando la función `fgets()` y crear la lista de números a partir de este fichero. Esta tarea es la que realiza la función `lee_lista()` que podemos ver en `prog1.c`. Como el fichero `numeros` es un fichero de texto, se puede modificar manualmente con un editor de texto, como el propio de Turbo C, para modificar o crear nuevos datos.

Ejercicio: ejecutar el programa `prog1.c` y ver el resultado que muestra por pantalla (acuerdate que tienes que crear un proyecto).

Ejercicio: cerrar el proyecto anterior y cambiar la línea de `#include` necesaria para que el programa utilice la librería `lista_ed.c` que implementa la representación de listas mediante variable dinámica, crea un nuevo proyecto y ejecuta el programa:

- ¿existe alguna diferencia en el resultado?
- ¿existe alguna diferencia en el programa principal?

Si te fijas hemos conseguido que el programa principal sea independiente de la implementación utilizada para las listas. Además el programa está mucho mejor organizado y el código es más legible. Ahora por ejemplo, si sabemos que nuestra librería para listas esta libre de errores, en el caso en que nuestro programa funcionara mal, podríamos centrarnos en la búsqueda del error en otra parte del código que no corresponda al código de las listas.

Por otro lado, también hemos conseguido que repetir experimentos en igualdad de condiciones, esto es, con los mismos datos de entrada, sea más cómodo con el uso de ficheros de texto donde se almacenan los datos del programa.

6.2. Listas de fichas de datos

Este ejercicio es idéntico al anterior exceptuando que aquí utilizaremos fichas de datos de alumnos en vez de simples números enteros. El objetivo de este ejercicio va a ser conseguir que nuestro programa lea una serie de fichas de alumnos de un fichero y cree una lista con ellas; después debe imprimir por pantalla dicha lista de fichas y terminar.

Para ello, vamos a especificar un formato muy simple para el fichero que va a contener los datos de las fichas de los alumnos. Cada ficha estará compuesta de tres datos: nombre del alumno, una de las asignaturas de las que se ha examinado y nota de dicha asignatura. Cada uno de los datos estará en una linea del fichero de texto, ademas, para cada ficha, los datos deben estar consecutivos y en este orden:

```
nombre
nota de la asignatura
asignatura
```

En el fichero, después de una ficha completa, puede venir otra ficha completa o el final del fichero. Podemos ver un ejemplo de fichero de texto con fichas de alumnos en el fichero `fichas` proporcionado para la práctica. La forma en cómo se deben leer los datos de este fichero y crear la lista puede verse en la función `lee_lista()` de `prog2.c`. Todos los códigos fuente para este sección se muestran en la sección 6.4.

Ejercicio: completar el código necesario de las funciones `insertar()`, `borrar()`, `vacial()`, `fin()`, `principio()` y `siguiente()` del módulo `lista_v.c` para fichas de alumnos con representación vectorial.

Ejercicio: ejecutar el nuevo programa `prog2.c` y ver el resultado que muestra por pantalla (acuérdate que tienes que crear un proyecto).

Ejercicio: completar el código necesario de las funciones `insertar()`, `borrar()`, `vacial()`, `fin()`, `principio()` y `siguiente()` del módulo `lista_ed.c` para fichas de alumnos con representación enlazada mediante variable dinámica.

Ejercicio: cerrar el proyecto anterior y cambiar la línea de `#include` necesaria para que el programa utilice la librería `lista_ed.c` que implementa la representación de listas mediante variable dinámica, crea un nuevo proyecto y ejecuta el programa.

Puedes probar a modificar los datos del fichero `fichas`.

6.3. Códigos fuente para la sección 1

Codigo de lista_v.h:

```
/* Cabecera para listas con representacion vectorial vista en clase. */
#include <stdio.h>
#include <stdlib.h>
/*****************************************/
/* Definicion de tipos de datos: */
#define maxL 1000

typedef int posicion;
typedef struct {
     int v[maxL];
     posicion ultimo;
} lista;
/*****************************************/
/* Operaciones definidas: */

lista *crearl();
lista *insertar(lista *l, int e, posicion p);
lista *borrar(lista *l, posicion p);
int recuperar(lista *l, posicion p);
int vacial(lista *l);
posicion fin(lista *l);
posicion principio(lista *l);
posicion siguiente(lista *l, posicion p);
```

Codigo de lista_v.c:

```
/* Libreria para listas con representacion vectorial vista en clase. */
#include "lista_v.h"
/****************************************/
/* Operaciones definidas: */
/****************************************/
lista *crearl() {
  lista *l;

  l = (lista *) malloc(sizeof(lista));
  l->ultimo = -1;
  return(l);
}
/****************************************/
lista *insertar(lista *l, int e, posicion p) {
  posicion i;

  if (l->ultimo == maxL-1)
       printf("ERROR: Lista llena. Imposible insertar\n");
  else {
       for (i=l->ultimo; i>=p; i--)
            l->v[i+1] = l->v[i];
       l->v[p] = e;
       l->ultimo = l->ultimo + 1;
  }
  return(l);
}
/****************************************/
lista *borrar(lista *l, posicion p) {
  posicion i;

  for (i=p; i<l->ultimo; i++)
         l->v[i] = l->v[i+1];
  l->ultimo = l->ultimo -1;
  return(l);
}
/****************************************/
int recuperar(lista *l, posicion p) {
  return(l->v[p]);
}
```

```
/****************************************/
int vacial(lista *l) {
  return(l->ultimo < 0);
}
/****************************************/
posicion fin(lista *l) {
  return(l->ultimo + 1);
}
/****************************************/
posicion principio(lista *l) {
  return(0);
}
/****************************************/
posicion siguiente(lista *l, posicion p) {
  return(p + 1);
}
```

Codigo de lista_ed.h:

```
/* Cabecera para listas con representacion enlazada de variable dinamica. */
#include <stdio.h>
#include <stdlib.h>
/****************************************/
/* Definicion de tipos de datos: */
typedef struct _lnodo {
     int e;
     struct _lnodo *sig;
} lnodo;
typedef lnodo *posicion;
typedef struct {
     posicion primero, ultimo;
} lista;
/****************************************/
/* Operaciones definidas: */

lista *crearl();
lista *insertar(lista *l, int e, posicion p);
lista *borrar(lista *l, posicion p);
int recuperar(lista *l, posicion p);
int vacial(lista *l);
posicion fin(lista *l);
posicion principio(lista *l);
posicion siguiente(lista *l, posicion p);
```

Codigo de lista_ed.c:

```
/* Libreria para listas con representacion enlazada de variable dinamica */
#include "lista_ed.h"
/****************************************/
/* Operaciones definidas: */
/****************************************/
lista *crearl() {
  lista *l;

  l = (lista *) malloc(sizeof(lista));
  l->primero = (lnodo *) malloc(sizeof(lnodo));
  l->primero->sig = NULL;
  l->ultimo = l->primero;
  return(l);
}
/****************************************/
lista *insertar(lista *l, int e, posicion p) {
  posicion q;

  q = p->sig;
  p->sig = (lnodo *) malloc(sizeof(lnodo));
  p->sig->e = e;
  p->sig->sig = q;

  if (p == l->ultimo)
       l->ultimo = p->sig;
  return(l);
}
/****************************************/
lista *borrar(lista *l, posicion p) {
  posicion q;

  if (p->sig == l->ultimo)
       l->ultimo = p;
  q = p->sig;
  p->sig = p->sig->sig;
  free(q);
  return(l);
}
```

```
/****************************************/
int recuperar(lista *l, posicion p) {
  return(p->sig->e);
}
/****************************************/
int vacial(lista *l) {
  return(l->primero->sig == NULL);
}
/****************************************/
posicion fin(lista *l) {
  return(l->ultimo);
}
/****************************************/
posicion principio(lista *l) {
  return(l->primero);
}
/****************************************/
posicion siguiente(lista *l, posicion p) {
  return(p->sig);
}
```

Codigo de prog1.c:

```
/* Programa que lee una lista de numeros de */
/* un fichero y la imprime por pantalla. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include "lista_v.h"
/* #include "lista_ed.h" */


/*************************************/
/* Funcion que crea una lista de numeros */
/* leidos de una fichero de texto. */

lista *lee_lista(char *nombre_fichero) {
  FILE *f;
  lista *l;
  char cadena[100];
  int elemento;
  posicion pos;

  f=fopen(nombre_fichero,"r");
  if (f==NULL) {
         printf("ERROR: No se encuentra fichero %s\n",nombre_fichero);
         exit(1);
  }
  l = crearl();
  pos = fin(l);
  while (fgets(cadena, 100, f) != NULL) {
        cadena[strlen(cadena)-1] = '\0';
        elemento = atoi(cadena);
        insertar(l, elemento, pos);
        pos = siguiente(l, pos);
  }
  fclose(f);
  return(l);
}
```

```
/************************************/
/* Procedimiento que imprime los elementos */
/* de una lista de numeros. */

void imprime_lista(lista *l) {
  int e;
  posicion pos;

  printf("\n\nImprimiendo lista:\n");
  pos = principio(l);
  while (pos != fin(l)) {
        e = recuperar(l, pos);
        printf("Elemento: %d\n",e);
        pos = siguiente(l, pos);
}
printf("\n\n");
}


/*******************************************/
/******     PROGRAMA PRINCIPAL     ******/
/*******************************************/

void main(){
  lista *l;
  char aux[5];

  clrscr();
  /* creamos la lista a partir del fichero de numeros */
  l = lee_lista("numeros");

  /* imprimimos la lista por pantalla */
  imprime_lista(l);

  printf("Pulse intro para acabar");
  gets(aux);
}
```

6.4. Códigos fuente para la sección 2

Código de lista_v.h:

```
/* Cabecera para listas con representacion vectorial vista en clase. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/****************************************/
/* Definicion de tipos de datos: */
#define maxL 100
#define maxcad 100

typedef struct {
     char nombre[maxcad];
     float nota;
     char asignatura[maxcad];
} ficha;
typedef int posicion;
typedef struct {
     ficha v[maxL];
     posicion ultimo;
} lista;
/****************************************/
/* Operaciones definidas: */

lista *crearl();
lista *insertar(lista *l, ficha *e, posicion p);
lista *borrar(lista *l, posicion p);
ficha *recuperar(lista *l, posicion p);
int vacial(lista *l);
posicion fin(lista *l);
posicion principio(lista *l);
posicion siguiente(lista *l, posicion p);
```

Código de lista_v.c:

```
/* Libreria para listas con representacion vectorial vista en clase. */

#include "lista_v.h"

/*****************************************/
/* Operaciones definidas: */

/*****************************************/
lista *crearl() {
  lista *l;

  l = (lista *) malloc(sizeof(lista));
  l->ultimo = -1;
  return(l);
}

/*****************************************/
lista *insertar(lista *l, ficha *e, posicion p) {
  posicion i;

  /* INCLUIR CODIGO PARA INSERTAR EN UNA LISTA VECTORIAL */
  /* .... */
  /* .... */
  /* .... */
}

/*****************************************/
lista *borrar(lista *l, posicion p) {
  posicion i;

  /* INCLUIR CODIGO PARA BORRAR DE UNA LISTA VECTORIAL */
  /* .... */
  /* .... */
  /* .... */
}
```

```
/****************************************/
ficha *recuperar(lista *l, posicion p) {
  return(&(l->v[p]));
}


/****************************************/
int vacial(lista *l) {
  /* INCLUIR CODIGO PARA vacial */
  /* .... */
}


/****************************************/
posicion fin(lista *l) {
  /* INCLUIR CODIGO PARA fin */
  /* .... */
}


/****************************************/
posicion principio(lista *l) {
  /* INCLUIR CODIGO PARA principio */
  /* .... */
}


/****************************************/
posicion siguiente(lista *l, posicion p) {
  /* INCLUIR CODIGO PARA siguiente */
  /* .... */
}
```

Código de lista_ed.h:

```
/* Cabecera para listas con representacion enlazada de variable dinamica. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/****************************************/
/* Definicion de tipos de datos: */
#define maxcad 100

typedef struct {
     char nombre[maxcad];
     float nota;
     char asignatura[maxcad];
} ficha;
typedef struct _lnodo {
     ficha e;
     struct _lnodo *sig;
} lnodo;
typedef lnodo *posicion;
typedef struct {
     posicion primero, ultimo;
} lista;
/****************************************/
/* Operaciones definidas: */

lista *crearl();
lista *insertar(lista *l, ficha *e, posicion p);
lista *borrar(lista *l, posicion p);
ficha *recuperar(lista *l, posicion p);
int vacial(lista *l);
posicion fin(lista *l);
posicion principio(lista *l);
posicion siguiente(lista *l, posicion p);
```

Código de lista_ed.c:

```
/* Libreria para listas con representacion enlazada de variable dinamica */

#include "lista_ed.h"

/****************************************/
/* Operaciones definidas: */

/****************************************/
lista *crearl() {
  lista *l;

  l = (lista *) malloc(sizeof(lista));
  l->primero = (lnodo *) malloc(sizeof(lnodo));
  l->primero->sig = NULL;
  l->ultimo = l->primero;
  return(l);
}

/****************************************/
lista *insertar(lista *l, ficha *e, posicion p) {
  posicion q;

  /* INCLUIR CODIGO PARA INSERTAR EN UNA LISTA ENLAZADA */
  /* .... */
  /* .... */
  /* .... */
}

/****************************************/
lista *borrar(lista *l, posicion p) {
  posicion q;

  /* INCLUIR CODIGO PARA BORRAR DE UNA LISTA ENLAZADA */
  /* .... */
  /* .... */
  /* .... */
}
```

```
/****************************************/
ficha *recuperar(lista *l, posicion p) {
  return(&(p->sig->e));
}

/****************************************/
int vacial(lista *l) {
  /* INCLUIR CODIGO PARA vacial */
  /* .... */
}

/****************************************/
posicion fin(lista *l) {
  /* INCLUIR CODIGO PARA fin */
  /* .... */
}

/****************************************/
posicion principio(lista *l) {
  /* INCLUIR CODIGO PARA principio */
  /* .... */
}

/****************************************/
posicion siguiente(lista *l, posicion p) {
  /* INCLUIR CODIGO PARA siguiente */
  /* .... */
}
```

Código de prog2.c:

```c
/* Programa que lee una lista de numeros de fichero y la imprime. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include "lista_v.h"
/* #include "lista_ed.h" */
/*************************************/
/* Funcion que crea una lista de fichas leidas de un fichero. */
lista *lee_lista(char *nombre_fichero) {
  FILE *f;
  lista *l;
  char cadena[100];
  ficha *elemento;
  posicion pos;

  elemento = (ficha *) malloc(sizeof(ficha));
  f=fopen(nombre_fichero,"r");
  if  (f==NULL) {
         printf("ERROR: No se encuentra fichero %s\n",nombre_fichero);
         exit(1);
  }
  l = crearl();
  pos = fin(l);
  while (fgets(cadena, 100, f) != NULL) {
        cadena[strlen(cadena)-1] = '\0';
        strcpy(elemento->nombre, cadena);
        if  (fgets(cadena, 100, f) != NULL) {
               cadena[strlen(cadena)-1] = '\0';
               elemento->nota = atof(cadena);
        }
        if  (fgets(cadena, 100, f) != NULL) {
               cadena[strlen(cadena)-1] = '\0';
               strcpy(elemento->asignatura, cadena);
        }
        insertar(l, elemento, pos);
        pos = siguiente(l, pos);
  }
  fclose(f);
  return(l);
}
```

```
/*************************************/
/* Procedimiento que imprime los elementos */
/* de una lista de numeros. */

void imprime_lista(lista *l) {
  ficha *e;
  posicion pos;

  printf("\n\nImprimiendo lista:\n");
  pos = principio(l);
  while (pos != fin(l)) {
        e = recuperar(l, pos);
        printf("Nombre: %s\n",e->nombre);
        printf("Asignatura: %s    Nota: %f\n\n", e->asignatura, e->nota);
        pos = siguiente(l, pos);
}
printf("\n\n");
}


/*******************************************/
/******    PROGRAMA PRINCIPAL    ******/
/*******************************************/

void main(){
  lista *l;
  char aux[5];

  clrscr();
  /* creamos la lista a partir del fichero de fichas */
  l = lee_lista("fichas");

  /* imprimimos la lista por pantalla */
  imprime_lista(l);

  printf("Pulse intro para acabar");
  gets(aux);
}
```

Práctica 7

Utilizando otros recursos para programar

En esta práctica pretendemos introducir al alumno en la búsqueda de otros recursos a la hora de programar, aparte de la inteligencia y predisposición del programador.

Como otros recursos entenderemos todos aquellos mecanismos que pueden facilitarnos, agilizarnos o aclararnos determinadas tareas de la programación. Por ejemplo, supongamos que queremos implementar una cola de prioridades para la utilización de la impresora de una empresa; si resulta que ya existe otra persona que ha realizado un gran trabajo programando colas de prioridades para otro dispositivo, ¿por qué no utilizar su trabajo (código programado), así como sus conocimientos sobre el tema? Debemos considerar que aparte del ahorro de tiempo escribiendo código es probable que las líneas de código que haya escrito contendrán menos errores en promedio que las que podamos escribir nosotros, debido a su experiencia en el tema así como el tiempo que haya destinado a comprobar la bondad de su código.

Una primera ayuda para programar que ya conocemos es la propia ayuda que nos ofrezca el entorno de programación en el que estemos desarrollando software. Por ejemplo, en el caso de Turbo C se tratará de las ventanas de ayuda que puede facilitar el entorno de Turbo C, mientras que si usamos el `gcc`[1] de Linux podemos obtener ayuda utilizando el comando `man`.

Otra ayuda bastante útil es la que nos puede ofrecer Internet. En Internet se puede encontrar gran cantidad de información referente a programación. Desde manuales de programación a librerías que pueden prestarnos una solución directa al problema que intentamos resolver, así como mucho software de desarrollo que puede hacernos más cómoda y efectiva nuestra programación.

[1] gnu c compiler.

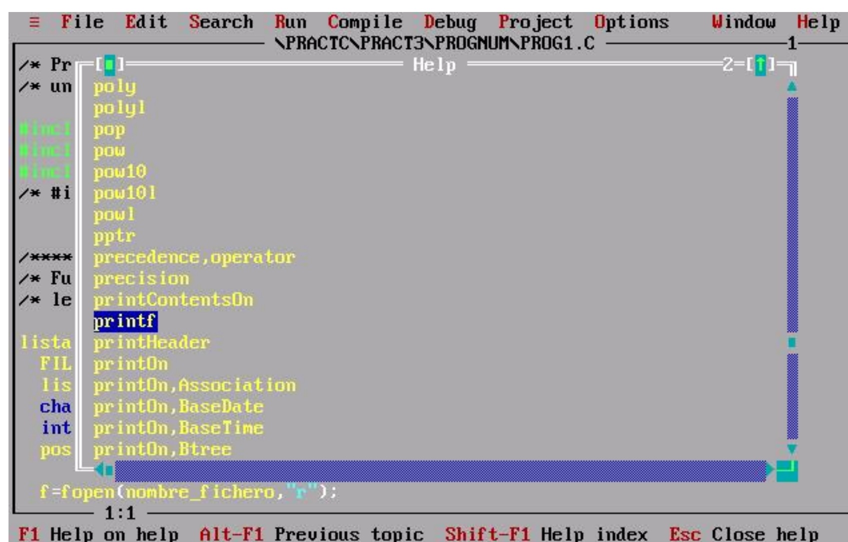


Figura 7.1: Ejemplo de ayuda por tópico de Turbo C a partir de un comando printf().

Veremos en primer lugar un recordatorio de como usar la ayuda de Turbo C y después veremos más en profundidad varias direcciones de Internet que nos proporcionarán una serie de recursos importantes para la programación en lenguaje C.

Ayuda en línea de Turbo C

Durante las prácticas del curso con Turbo C, el alumno ya ha utilizado repetidas veces esta utilidad, por lo que aquí solo reseñaremos dos formas de uso:

- La ayuda por tópico: para ello, posicionamos el cursor sobre la palabra (identificador reservado) sobre el que queremos obtener ayuda y pulsamos Ctrl+F1. Se abre una ventana como la de la figura 7.1

- El índice de la ayuda, donde se muestran todos las funciones e identificadores reservados de Turbo C por orden alfabético. Se puede acceder a él con Shift+F1. Se abre una ventana como la de la figura 7.2.

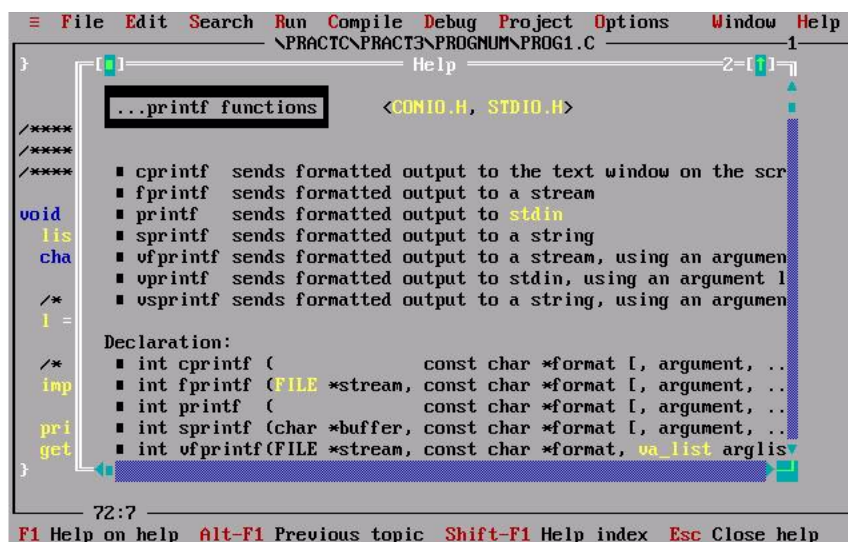


Figura 7.2: Ejemplo de índice de la ayuda de Turbo C.

Buscando en la Web

Internet pone a nuestro alcance gran cantidad de información. Mucha de esta información está destinada a desarrolladores de software y programadores. A continuación vamos a revisar algunos enlaces con información referente al lenguaje C activos a fecha 11-02-2002 que pueden ser de gran utilidad para el alumno:

Manuales y libros de C

Cuando tengamos dudas de sintaxis del lenguaje C o queramos aprender nuevas funcionalidades del lenguaje más avanzadas que la programación básica, estos enlaces pueden ser de gran ayuda:

- *"Programming in C: A tutorial"*. Un libro que sirve como iniciación al C:
  http://www.lysator.lin.se/c/bwk-tutor.html

- *"Programming in C"*. Un libro electrónico muy completo. Prácticamente fundamental. Con pequeños ejemplos de código y algunos programas-ejemplo:
  http://www.cm.cf.ac.uk/Dave/C/CE.html

- *"Thinking in C++"*. Libro electrónico:
  http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

- *"C++ Programming HOWTO"*. Documentación en línea del lenguaje C++:
  http://www.linuxdoc.org/HOWTO/C++Programming-HOWTO.html

- *"C++ Annotations"*. Libro electrónico:
  http://www.icce.rug.nl/documents/cpp.shtml

Librerías

Éste es uno de los apartados más importantes, ya que nos puede permitir encontrar librerías de estructuras de datos y soluciones algorítmicas ya implementadas y comprobadas que nos ahorren mucho tiempo de desarrollo:

- Graph Template Library. Librería para grafos en C++:
  http://www.infosun.fmi.uni-passau.de/GTL/

- DISLIN - Scientific Data Plotting Software. Librerías para dibujar gráficos científicos:
  http://www.linmpi.mpg.de/dislin/

- SGI Standard Template Library Programmer's Guide. Guía de librerías para programadores de SGI:
  http://www.sgi.com/tech/stl/

Compiladores

Es interesante tener disponibilidad de compiladores de uso libre que además dispongan de un entorno de desarrollo bastante agradable:

- Compiladores de lenguaje C (y otros lenguajes) gratuitos:
  http://program.solobox.com/compilers/c.html

- Compilador de C de 32 bits para Windows gratuito:
  http://www.mingw.org

Generales

- Enlaces que permiten acceder a mucha información sobre el lenguaje de programación C:
  http://home.att.net/~jackklein/c/c_links.html

- Guías de referencia rápida, en formato pdf o ps (postscript):
  http://www.refcard.net

Conclusiones

Un aspecto a tener en cuenta es que de la misma manera que tenemos toda la información anterior disponible para el lenguaje C, existe también gran cantidad de información disponible para otros lenguajes de programación como Java, Pascal, Python, Perl, etc.

Por último, se invita al alumno a investigar más en Internet, haciendo uso de los enlaces propuestos anteriormente, así como de los buscadores disponibles en la red.

Práctica 8

Evaluando el coste temporal empírico de un programa

Los objetivos de esta práctica son: en primer lugar, aprender a medir empíricamente el coste computacional de un programa y en segundo lugar estudiar experimentalmente el algoritmo rápido de ordenación conocido como `Quicksort`. Este algoritmo fue propuesto por Hoare en 1962 y es uno de los más eficientes algoritmos de ordenación conocidos. Para el estudio teórico, referirse al material visto en clase de teoría.

8.1. Medición empírica del coste temporal

Una forma de medir el coste computacional asociado a la ejecución de un programa o segmento del mismo, consiste en utilizar alguna rutina del sistema que mida tiempos de ejecución. Es importante señalar que si se trabaja en un sistema de tiempo compartido, las medidas que se realicen deberán ser independientes de la carga del sistema.

Normalmente, los sistemas operativos llevan una cuenta para cada proceso y en cada momento del número de *"ticks"* de CPU consumidos. Estos ticks equivalen generalmente a una pequeña cantidad de tiempo que suele ser del orden de microsegundos. Podemos saber los segundos que ha tardado la ejecución de un programa dividiendo el número de ticks que ha costado ejecutarlo por una constante que indique el número de ticks por segundo que tiene cada sistema. En sistemas Windows y MSDOS esta constante suele llamarse CLK_TCK mientras que en sistemas Linux suele denominarse CLOCKS_PER_SEC. Otro aspecto a tener en cuenta es la resolución de la medida de tiempos, puesto que si transformamos de ticks a segundos es probable que no podamos diferenciar el tiempo de ejecución de varios programas

debido a la pérdida de resolución, por eso usaremos el tick como unidad temporal en vez de usar segundos, con esto conseguiremos obtener una mejor resolución temporal.

En lenguaje C es posible determinar el número de ticks haciendo uso de la rutina `clock()` (de la librería `time.h` del lenguaje C). Una llamada a esta función (sin argumentos), devuelve un entero que es proporcional a la cantidad de tiempo de procesador utilizado por el proceso en curso. Así, sea por ejemplo el siguiente fragmento de programa en C:

```
c1 = clock();     /* instruccion 1 */
procedure();
c2 = clock();     /* instruccion 2 */
c3 = c2 - c1;
```

donde `procedure();` es cualquier instrucción, un procedimiento o una función. Después de la ejecución de este fragmento de código tendremos:

c1	=	tiempo de proceso consumido **hasta la instrucción 1** (en ticks).
c2	=	tiempo de proceso consumido **hasta la instrucción 2** (en ticks).
c3	=	tiempo de proceso consumido **para el procedimiento** `procedure()` (en ticks).

Hay que tener cuidado en la colocación de las instrucciones de temporización, ya que se debe evitar incluir entre ellas toda aquella parte del código que no se desee cronometrar.

El programa `ejemplo1.c`, que puede consultarse en la sección 8.3, muestra cómo debe hacerse una medida de tiempos para la función $f(x)$ que calcula la siguiente expresión:

$$f(x) = \sum_{i=1}^{\lfloor x \rfloor} \prod_{j=1}^{2i} \sum_{k=1}^{3j} \frac{(k+x)}{x}$$

Todos los códigos fuente incluídos en las secciones finales de esta práctica serán proporcionados al alumno.

Ejercicio: ejecuta el programa `ejemplo1.c` y apunta el número de ticks que tarda en ejecutarse la función $f(x)$ para cada valor de x en una tabla como la siguiente. Cambia el valor inicial de la x, el incremento que se usa en el bucle y el límite para el valor de la x y crea una nueva tabla.

VALOR DE X	TICKS

8.2. Quicksort: otro algoritmo de Partición.

Si consideramos el algoritmo de partición estudiado en clase (algoritmo `partition`), y suponemos como entrada un vector completamente aleatorio, entonces, el coste esperado por término medio es $\Theta(kn \log n)$, puesto que `partition` elige un valor aleatorio al tomar como pivote el primer elemento del vector.

Pueden conseguirse mejoras en el tiempo de ejecución del algoritmo `Quicksort` que se ha estudiado reduciendo k, la constante asociada a `partition`. Modificando para ello la selección del pivote, de forma que la subdivisión en dos subvectores efectuada por el algoritmo de partición sea más equilibrada, evitando en lo posible la contribución de casos desequilibrados al coste promedio.

Weiss propuso en [Weiss95] un algoritmo de partición que mejoraba en dos aspectos al algoritmo `partition` estudiado. Uno de estos dos aspectos es la estrategia utilizada para seleccionar el pivote, que vamos a estudiar a continuación

8.2.1. Elección del pivote

Naturalmente hay unas elecciones del pivote mejores que otras. Del estudio de costes del algoritmo `Quicksort` parece razonable deducir que, cuanto más equilibrada sea la subdivisión realizada por `partition`, más próximo será el comportamiento temporal de `Quicksort` al de su caso mejor.

Cuando la entrada al algoritmo es aleatoria, cualquier elemento del vector serviría como pivote, ya que todos ellos tienen la misma probabilidad de

quedar centrados en la partición -o en un extremo de ella-, en ese caso, la elección del primer elemento es aceptable. Pero si el vector está parcialmente ordenado -algo bastante frecuente en la práctica- entonces la elección del primer elemento hará más probables particiones desequilibradas.

Una estrategia mejor consistiría en usar como pivote un elemento cualquiera del vector elegido al azar, evitando así elecciones deficientes debido a entradas no aleatorias. El problema de esta última elección es que la generación de un número aleatorio puede ser temporalmente costosa, con lo que se penalizaría el algoritmo de partición.

Otra estrategia posible consiste en elegir el elemento central del vector en lugar del primero, de modo que `partition` dará, por lo general, subdivisiones más equilibradas cuando la entrada ya esté parcialmente ordenada, manteniendo la aleatoriedad de la elección para entradas también aleatorias.

La elección óptima del pivote consistiría en seleccionar la mediana del vector, esto es: aquel elemento que tiene un número igual de elementos mayores y menores que él mismo. Esta elección garantizaría siempre la subdivisión más equilibrada; sin embargo determinar la mediana de un vector tiene como mínimo un coste lineal, lo que supone en la práctica una penalización temporal excesiva.

Otra estrategia distinta, la considerada por Weiss, consiste en seleccionar tres elementos del vector, el primero, el central y el último, eligiendo como pivote la mediana de los tres, esto es, el elemento intermedio. Este método es, frecuentemente, mejor que una elección puramente aleatoria, ya que escoge como pivote un elemento que estará con más probabilidad cerca de la mediana que de haberse elegido al azar; además, el método elimina el caso "malo" que se produce ante una entrada ordenada.

El algoritmo que mostramos a continuación, debido a Weiss, determina el pivote que utilizará el algoritmo de partición, para el subvector de `A` comprendido entre `l` y `r`.

```
tipobase mediana3(vector A, int l, int r) {
  int centro;  tipobase pivote;

  centro=(l+r) / 2;
  if (A[l] > A[centro]) intercambiar(A[l],A[centro]);
  if (A[l] > A[r]) intercambiar(A[l],A[r]);
  if (A[centro] > A[r]) intercambiar(A[centro],A[r]);
  pivote=A[centro];
  intercambiar(A[centro],A[r-1]);   /* oculta el pivote */
  return(pivote);
}
```

Como hemos indicado, además de determinar el pivote, el algoritmo intercambia entre sí los elementos inicial, central y final del subvector de forma que estén ordenados entre sí, lo que, además de situar ya dichos elementos, garantiza la existencia de un elemento inferior o igual al pivote en el extremo izquierdo, de forma que pueda utilizarse como centinela en búsquedas posteriores.

8.2.2. Evaluando algoritmos de ordenación

Inserción directa Vs Quicksort

La tarea a realizar es comparar experimentalmente el algoritmo Quicksort (de metodología Divide y Vencerás) con un algoritmo de ordenación de los llamados "directos", como es Inserción directa. Estos algoritmos ya se proporcionan implementados en una librería compuesta por el fichero de cabecera `sorting.h` y el fichero `sorting.c` (ver códigos fuente en la sección 8.4). Para utilizar estos módulos en Turbo C, debe crearse un proyecto.

El programa principal se encuentra en el fichero `timing.c` (ver código fuente en la sección 8.4) y para realizar la práctica, el alumno debe acabar la implementación del programa principal que realiza la toma de tiempos de las diversas ejecuciones de los algoritmos de ordenación.

El tiempo de ejecución de un determinado algoritmo puede calcularse con la rutina `clock` del lenguaje C, como se ha visto en la sección anterior. Como la idea es ver el comportamiento de los algoritmos de ordenación dependiendo de la talla del vector a ordenar, la salida del programa debe facilitarnos los ticks de reloj que consumen Inserción directa y Quicksort para diferentes tallas de vectores. Si el alumno lo prefiere, puede generar un fichero de salida con los datos de talla y tiempos asociados para dibujar cómodamente una gráfica de la evolución del coste temporal utilizando algún programa de dibujo de gráficas.

Se deberán medir los tiempos de ejecución de cada uno de los algoritmos en función de la talla, para un rango de talla entre 5000 y 15000, a intervalos de 1000.

Para cada talla, se debería generar un vector aleatorio, pero en ese caso, la medición de tiempo que realicemos para una sola talla estaría totalmente condicionada a este vector, con lo que si la "aleatoriedad" del sistema ha generado una mala instancia para quicksort, pero buena para Inserción directa, podemos estar obteniendo medidas de tiempo "no del todo correctas" para una determinada talla. Para solucionar este problema, para cada talla, generaremos un conjunto de vectores aleatorios de dicha talla (un conjunto de instancias para cada talla), por ejemplo 100 vectores. Calcularemos el

tiempo que cuesta ordenar cada vector y acumularemos todos los tiempos para obtener finalmente la media dividiendo por el número de vectores de esa talla. Así obtendremos el tiempo promedio de cada algoritmo para una talla determinada.
Ejercicio: Completar el código del programa principal de `timing.c` y realizar la comparativa entre Quicksort e Inserción directa.

Finalmente, el alumno debe poder rellenar una tabla como la siguiente u obtener un fichero con estos resultados.

TALLA VECTOR	INSERCION DIRECTA	QUICKSORT
500		
1000		
1500		
2000		
2500		
⋮		

Influencia del comportamiento de partición en Quicksort

El trabajo ahora consistirá en implementar la modificación propuesta en la sección 8.2.1 para el algoritmo de partición (`partition`) en una nueva función.

Respecto a la modificación de Weiss del algoritmo partición, esta partición solo es aplicable a vectores de tres o más elementos, por lo que en esta implementación de Quicksort, el caso general se dará cuando $l+1 < r$. Debes fijarte que el nuevo procedimiento `quicksort_Weiss()` tendrá un código ligeramente diferente al procedimiento `quicksort()`.
Ejercicio: completar el código de la función `partition_Weiss()` del módulo `sorting.c` así como el código del algoritmo `quicksort_Weiss()`. Puedes ver el código de `sorting.c` en la sección 8.4.
Ejercicio: comparar el comportamiento temporal de las dos versiones de Quicksort de igual manera que anteriormente comparamos Inserción directa y Quicksort.

Para realizar la comparativa, deberás cambiar el programa principal en `timing.c` para tomar tiempos de `quicksort()` y `quicksort_Weiss()`. Si resulta necesario para poder obtener una comparativa interesante, se deberá aumentar el rango de tallas de los vectores a estudiar.

8.3. Códigos fuente para la sección 1

Código de ejemplo1.c:

```
/* Programa que calcula la funcion f(x) para varios valores de x */
/* y mide el tiempo de ejecucion de cada llamada a f(x) en ticks */
/* de reloj de CPU */
#include <stdio.h>
#include <conio.h>
#include <time.h>
/****************************************/
/* DEFINICION DE CONSTANTES */
#define x_inicial  102.1
#define incremento 44.1
#define limite_x   600.0
/****************************************/
double f(double x) {
  int i, j, k;
  double a1, a2, a3;

  a1 = 0.0;
  for(i=1; i<=(int)x; i++) {
     a2 = 1.0;
     for(j=1;j<=2*i;j++) {
         a3 = 0.0;
         for(k=1;k<=3*j;k++) a3 = (a3 + x)/x;
         a2 = a2 * a3;
         }
     a1 = a1 + a2;
     }
  return(a1);
}
```

```c
/***********  PROGRAMA PRINCIPAL  ***********/
void main(){
  double resultado, x;
  int t1, t2, t3;
  char aux[3];

  clrscr();
  x = x_inicial;
  while (x < limite_x) {
      t1 = clock();
      resultado = f(x);
      t2 = clock();
      t3 = t2 - t1;
      printf("x = %lf --> Numero de ticks: %d\n",x,t3);
      x += incremento;
  }
  printf("\n\nPulsa Intro para acabar\n");
  gets(aux);
}
```

8.4. Códigos fuente para la sección 2

Código de sorting.h:

```
/* Fichero cabecera para algoritmos de ordenacion. */

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


/****************************************/
/* Algoritmos definidos: */


void insercion_directa(int *A, int l, int r);
void quicksort(int *A, int l, int r);
void quicksort_Weiss(int *A, int l, int r);
```

Código de sorting.c:

```
/* Libreria para algoritmos de ordenacion. */

#include "sorting.h"

/****************************************/
/* Funciones internas a la libreria: */

/****************************************/
int partition(int *A, int l, int r) {
  int i, j, pivote, aux;

  i = l-1;
  j = r+1;
  pivote = A[l];

  while (i<j) {
     do {
        j--;
     } while(A[j]>pivote);

     do {
        i++;
     } while(A[i]<pivote);

     if (i<j) {
        aux = A[i];
        A[i] = A[j];
        A[j] = aux;
     }
  }
  return(j);
}
```

```
/*****************************************/
int partition_Weiss(int *A, int l, int r) {
  int i, j, pivote, aux;
  int centro;

  /* INCLUIR CODIGO PARA CALCULAR EL PIVOTE */
  /* COMO LA PSEUDO-MEDIANA PROPUESTA POR */
  /* WEISS */

  /* .... */
  /* .... */

  /* .... */
  /* .... */

  i = l-1;
  j = r+1;

  while (i<j) {
     do {
        j--;
     } while(A[j]>pivote);

     do {
        i++;
     } while(A[i]<pivote);

     if (i<j) {
        aux = A[i];
        A[i] = A[j];
        A[j] = aux;
     }
  }
  return(j);
}
```

```
/* Algoritmos definidos: */
/****************************************/
void insercion_directa(int *A, int l, int r) {
  int i, j, aux;

  for (i=l+1;i<=r;i++) {
      aux = A[i];
      j = i;
      while ((j>l) && (A[j-1]>aux)) {
         A[j] = A[j-1];
         j--;
      }
      A[j] = aux;
  }
}
/****************************************/
void quicksort(int *A, int l, int r) {
  int q;

  if (l<r) {
     q = partition(A,l,r);
     quicksort(A,l,q);
     quicksort(A,q+1,r);
  }
}
/****************************************/
void quicksort_Weiss(int *A, int l, int r) {
  int q, aux;

  /* INSERTAR CODIGO CORRESPONDIENTE */
  /* AL QUICKSORT DE WEISS */
  /* .... */
  /* .... */

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

Código de timing.c:

```c
/* Programa para estudiar experimentalmente el */
/* coste de algoritmos de ordenacion. */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <time.h>
#include "sorting.h"

/****************************************/
/* Definicion de constantes: */
#define max_vector 15000
#define talla_ini 5000
#define talla_fin 15000
#define talla_inc 1000
#define num_vect 100

/**************************************/
/* funcion que genera un vector aleatorio */
/* de la talla especificada */
int *genera_vector_aleatorio(int *V, int talla, int semilla) {
  time_t t;
  int i;

  srand((unsigned) time(&t)+semilla);
  for(i=0; i<talla; i++) V[i] = 1+(int) (100.0*rand()/(RAND_MAX+1.0));
  return(V);
}
```

```
/*********************************************/
/******    PROGRAMA PRINCIPAL    ******/
/*********************************************/
void main(){
  int *A1, *A2;
  int talla, n_vect;
  int idmedia, qmedia, i, semilla=1;
  double idmedia_r, qmedia_r;
  int t1, t2, t;
  char aux[3];

  clrscr();
  A1 = (int *) malloc(max_vector * sizeof(int));
  A2 = (int *) malloc(max_vector * sizeof(int));
  for (talla=talla_ini; talla<=talla_fin; talla+=talla_inc) {
      idmedia = 0;
      qmedia = 0;
      for (n_vect=1; n_vect<=num_vect; n_vect++) {
          A1 = genera_vector_aleatorio(A1,talla,semilla);
          for(i=0;i<talla;i++) A2[i] = A1[i];
          semilla++;
          /* UTILIZANDO t1, t2 Y t, ESCRIBIR EL */
          /* CODIGO NECESARIO PARA MEDIR EL TIEMPO */
          /* DE INSERCION DIRECTA PARA ORDENAR EL */
          /* VECTOR A1 */
          /* .............. */
          idmedia += t;
          /* UTILIZANDO t1, t2 Y t, ESCRIBIR EL */
          /* CODIGO NECESARIO PARA MEDIR EL TIEMPO */
          /* DE QUICKSORT PARA ORDENAR EL VECTOR A2 */
          /* .............. */
          qmedia += t;
      }
      idmedia_r = idmedia / num_vect;
      qmedia_r = qmedia / num_vect;
      printf("Talla: %d   Ins. Directa:  %lf    Quicksort:  %lf\n",
                                            talla,idmedia_r,qmedia_r);
  }
  printf("Pulse intro para acabar");
  gets(aux);
}
```

Práctica 9

Estudio de tablas de dispersión

9.1. Introducción

En esta práctica se pretende estudiar diferentes aspectos computacionales de las tablas de dispersión., así como una pequeña aplicación de las mismas. Los objetivos concretos del trabajo son los siguientes:

- realizar un estudio comparativo de diferentes funciones de dispersión,
- realizar un estudio experimental para determinar el tamaño de la tabla y de la función de dispersión más adecuada,
- aplicar las tablas de dispersión a una tarea sencilla.

Este documento se estructura de la siguiente manera: en la segunda sección se comentan brevemente las características más importantes de las tablas de dispersión y en la tercera, cuarta y quinta sección se explica cuáles son las actividades a realizar en la práctica. En las secciones finales se encuentran los códigos fuente de los algoritmos que serán proporcionados al alumno.

9.2. Tablas de dispersión

Una tabla de dispersión es una estructura de datos apropiada para representar un conjunto de elementos cuando las operaciones a optimizar son *insertar* (INSERT), *eliminar* (DELETE) y comprobar si un elemento *pertenece* (SEARCH) al conjunto, esto es, para representar un *diccionario*.

En una tabla de dispersión, los elementos se distribuyen en un conjunto de m cubetas utilizando para ello una clave y una *función de dispersión*. Cuando varios elementos se dispersan en la misma cubeta decimos que se

produce una colisión. A la hora de implementar una tabla de dispersión existen diversas propuestas para tratar con el problema de las colisiones: una posibilidad es utilizar una representación con *direccionamiento abierto* (ver referencias), mientras que otra posibilidad es utilizar una representación con *encadenamiento separado*. Esta última consiste básicamente en un vector donde cada componente apunta a una lista enlazada que está asociada a una determinada cubeta. La representación con encadenamiento es la propuesta que nosotros vamos a estudiar en esta práctica.

Un aspecto importante cuando se implementa una tabla de dispersión es la elección de una *función de dispersión* adecuada que permita distribuir los datos entre las diferentes cubetas equivalentemente. Otro punto interesante es la elección del número de cubetas. Para esto puede estudiarse el parámetro $\alpha = \frac{n}{m}$, llamado *factor de carga*, donde n es el número de elementos de la tabla y m es el número de cubetas. Estos serán dos de los puntos que estudiaremos con detenimiento en la práctica.

9.3. Actividades en el laboratorio

Esta práctica está organizada en varias sesiones dedicadas a los aspectos comentados anteriormente: la elección del tamaño de la tabla, la elección de una función de dispersión y la aplicación de las tablas de dispersión a un problema, que en este caso consistirá en un sencillo sistema *criptográfico* que por un lado *encriptará* un texto y generará también la clave para desencriptarlo, por otro lado se leerá la clave de desencriptado y el fichero encriptado y se restaurará el texto original.

9.4. Eligiendo el número de cubetas y la función hash

A la hora de elegir apropiadamente el tamaño de la tabla hash hay que llegar a un compromiso en el tamaño promedio que deseamos que tengan las cubetas (aparte de otras consideraciones, cómo la conveniencia de que m sea primo si la función de dispersión se basa en el método de la división). Por ejemplo, si deseamos que $\alpha \approx 2$ y el conjunto a dispersar tiene aproximadamente 300 elementos, el número de cubetas tendrá que ser un número primo próximo a 150 si utilizamos el método de la división (por ejemplo 151). Para facilitar al alumno la elección de un número primo cercano al necesario para conseguir el factor de carga deseado, se proporciona junto con los códigos fuente un fichero de texto llamado **PRIMOS** que contiene los números

primos entre 1 y 2000.

Para probar las tablas de dispersión sobre unos textos reales, se proporcionan al alumno dos ficheros de texto llamados **DOGVcas.txt** y **ENGLISH.txt** que servirán como ejemplo para aplicar todo los estudios que queremos realizar sobre las tablas de dispersión:

- **DOGVcas.txt** es un texto extraído del Diari Oficial de la Generalitat Valenciana y está escrito en castellano. El lenguaje utilizado es de tipo administrativo.
- **ENGLISH.txt** es un texto científico extraído de un artículo que trata sobre métodos de evaluación en traducción automática.

En la siguiente tabla se detalla un pequeño resumen de las características de cada fichero de texto:

	Núm. de líneas	Núm. de palabras	Núm. de palabras sin repetir (Vocabulario)
DOGVcas.txt	571	5877	1378
ENGLISH.txt	492	2689	918

Ejercicio: queremos saber para el fichero **DOGVcas.txt** y el fichero **ENGLISH.txt** cuáles pueden ser unos buenos tamaños para las tablas de dispersión suponiendo que deseamos que en cada cubeta haya aproximadamente 2 elementos, esto es, $\alpha = 2$.

Ejercicio: En los módulos **hash.h** y **hash.c** que se muestran en la sección 9.7 puedes encontrar la implementación para tablas hash que vamos a utilizar en esta práctica. Completa las funciones `crearTabla()`, `insertarTabla()`, `perteneceTabla()` y `eliminarTabla()` del módulo **hash.c** con el código necesario para que dichas funciones cumplan con su cometido.

Vamos a proponer ahora la problemática que supone la elección de una función de dispersión. Para ello, hemos propuesto cuatro funciones de dispersión diferentes (vistas en clase de teoría) que se deben estudiar con detenimiento: tres de ellas están basadas en el método de la división (se llamarán `func1`, `func2` y `func3`) y una en el método de la multiplicación (se llamará `func4`).

Una breve descripción de cada función es la siguiente:

Función 1 : Esta primera función (`func1`) se basa en el método de la división. Primeramente transforma una clave alfabética en una clave numérica sumando los códigos ASCII de cada carácter. Se caracteriza porque aprovecha toda la clave alfanumérica para realizar esta transformación.

Función 2 : La segunda función (`func2`) del módulo, también se basa en el método de la división. Una clave alfabética se transforma en una clave numérica, aprovechando para ello los tres primeros caracteres de la clave, pero considerando que son un valor numérico en una determinada base (256 en este caso).

Función 3 : La tercera función (`func3`) es similar a la segunda pero considerando toda la cadena. El hecho de que la operación módulo esté dentro del bucle es para evitar que se produzca un desbordamiento.

Función 4 : La cuarta función (`func4`) se basa en el método de la multiplicación. Considera que la clave es un valor numérico en base 2. En las referencias bibliográficas puede encontrarse documentación sobre la conveniencia de utilizar la constante que aparece en la función.

Ejercicio: En el módulo **hash.c**, completa el código necesario para la función `func1` que calcula la función de dispersión para una cadena con la propuesta de la Función 1. Recuerda que lo que calcula es:

$$h(x) = (\sum_{i=0}^{n-1} x_i) \bmod m$$

Ejercicio: Supondremos que ya se ha elegido el número de cubetas óptimo a utilizar. Prueba ahora cada función de dispersión sobre los ficheros de texto de prueba, generando los histogramas oportunos para ver como se dispersan los elementos en la cubetas de la tabla. A partir de los resultados. Intenta decidir qué función tiene un mejor comportamiento para los datos utilizados. Para generar los histogramas debes usar el programa **estadist.c**, que deberás incluir en un proyecto (project) junto con el módulo para las tablas hash. En el programa **estadist.c** tienes que rellenar el código necesario para llamar a la función `estadisticaTabla` del módulo hash.c, que generará un fichero de salida con el mismo nombre que el fichero de texto de entrada, pero con la extensión **.es?**, donde ? toma el valor 1, 2, 3 o 4 según se halla usado la función de dispersión 1, 2, 3 o 4. Este fichero de salida contiene varias lineas con 2 columnas, donde la primera columna es el número de cubeta de la tabla hash y la segunda columna es un número que indica la cantidad de elementos que contiene esa cubeta. Además al final del fichero, existen dos lineas que dan la media de elementos para cada cubeta de la tabla hash así como la desviación típica. Este fichero puede ser visualizado en forma de gráfica (o sea, como un histograma) con el programa **grafica.exe** que se proporciona al alumno, en el caso en que se esté trabajando con Turbo C y MS-DOS

y con el programa **gnuplot**[1] en el caso en que se esté trabajando con gcc y Linux. El programa **grafica.exe** pregunta por el fichero que contiene los datos para generar el histograma y muestra por pantalla una gráfica donde el eje X se corresponde con cada una de las cubetas y el eje Y indica el número de elementos en cada cubeta. Con esta gráfica podremos observar como se distribuyen los elementos en la cubetas de la tabla hash.

Ahora que tienes todas las funciones de dispersión implementadas y has elegido la que consideras la mejor para los textos ejemplo, podrías utilizar otro método para hallar cuál es el número de cubetas (m) óptimo para cada tarea. Antes lo hemos hecho con una simple división y aproximando a un número primo cercano, sin embargo, ahora podríamos hacerlo mediante un *barrido* del valor de m. Esto es, obteniendo el histograma para varios valores de m, y quedarnos con el valor de m que obtuviera una mejor distribución de los datos o más cercana a la deseada. Esto tan solo es una propuesta. Pruébalo si te apetece.

9.5. Encriptando un fichero de texto

Vamos a aplicar ahora las tablas de dispersión para resolver un problema determinado: la encriptación de texto.

Realmente el sistema de encriptado que vamos a proponer es extremadamente simple y sencillo. Además la protección que ofrece frente a ataques para su desencriptado es muy débil, por lo que no se recomienda su uso, sin embargo, es más que suficiente para ver las técnicas básicas utilizadas a la hora de trabajar con textos, por ejemplo, para realizar compiladores, criptografía o procesamiento del Lenguaje Natural.

En esta sección trataremos el sistema de encriptado de textos y en la siguiente sección trataremos la construcción del sistema de desencriptado de textos.

La idea para encriptar textos va a ser muy sencilla: supongamos que los textos con los que podemos trabajar tienen un vocabulario limitado. Entonces podríamos asignar un número diferente a cada palabra posible. Encriptar un texto determinado consistiría en ir leyendo cada una de las palabras que forman ese texto e ir generando otro texto de salida donde cada palabra de entrada es sustituida por su número identificador correspondiente. El texto de salida sería una secuencia de números que para un intruso resultarían un mensaje ininteligible. Con los algoritmos propuestos en el módulo **encripta.c** (ver código en la sección 9.9) obtendremos un fichero de salida con el

[1]preguntar al profesor sobre el manejo de esta herramienta en el caso en que se opte por ella durante el transcurso de la práctica

mismo nombre que el fichero de texto de entrada, pero con extensión **.out** y que será el texto encriptado. Ahora bien, supongamos que mandamos el texto encriptado a un amigo. Éste necesitaría ayuda para desencriptar el mensaje, puesto que el seguiría leyendo una secuencia de números sin sentido. Esta ayuda viene dada por lo que se conoce como **clave**, que en este caso esta formada por el vocabulario de encriptado, es decir por las palabras del vocabulario junto con sus respectivos identificadores. Este vocabulario de encriptado debe transmitirse al destinatario de nuestros mensajes encriptados por un canal seguro, esto es, libre de espías, puesto que si un intruso consigue este fichero ya tendrá la clave para desencriptar nuestros mensajes. Por todo esto, el sistema de encriptado también debe generar un fichero de salida que sea la **clave**, esto es, el vocabulario de encriptado. Los algoritmos proporcionados en el código de **encripta.c** generan otro fichero de salida que tiene el nombre del fichero de texto de entrada pero con extensión **.voc** y que será el vocabulario de traducción.

El vocabulario de traducción se implementa internamente como una tabla hash donde para cada cadena se guarda como información asociada su número identificador.

Ejercicio: Construir un proyecto con los módulos hash.c y encripta.c y completar las partes de código que faltan en encripta.c para construir un programa que encripte un fichero de texto como los proporcionados como ejemplo. Falta código en el programa principal así como en la función que introduce las cadenas en la tabla hash para construir el vocabulario de traducción.

Cuando hayas acabado el código podrás encriptar los ficheros de texto que se proporcionan como ejemplo.

9.6. Desencriptando un fichero de texto

Como ya se ha anticipado en la sección anterior el sistema de desencriptado recibirá por un lado un fichero con el texto encriptado consistente en una secuencia de números y por otro lado recibirá otro fichero que será la clave de desencriptado, esto es, el vocabulario de traducción.

La idea para el desencriptado es muy sencilla: bastará seguir el proceso inverso al encriptado; para cada número del fichero de entrada, devolver la palabra correspondiente y regenerar el texto original.

En el módulo **desencri.c** (ver código en la sección 9.10) se presentan los algoritmos para desencriptar. El vocabulario de traducción se implementa internamente como una tabla hash donde para cada cadena se guarda como información asociada su número identificador. A partir de la tabla hash se construye un vector de cadenas donde en cada posición del vector está al-

macenada una cadena del vocabulario de encriptado. La posición de cada cadena en el vector se corresponde con el identificador de dicha cadena (ya que es único). A partir de este vector de cadenas, la tarea de desencriptar es automática puesto que para cada número leido, obtenemos la cadena correspondiente a la posición de ese número y la ponemos en el fichero de salida que acabará conteniendo el texto desencriptado.
Ejercicio: Construir un proyecto con los módulos hash.c y desencri.c y completar las partes de código que faltan en desencri.c para construir un programa que desencripte un fichero de números. Falta código sólo en el programa principal.

Cuando hayas acabado el código podrás desencriptar los ficheros que anteriormente habías encriptado.

9.7. Librería de tablas hash

A continuación se presentan los módulos necesarios para implementar el tipo abstracto de datos **tabla de dispersión** en una librería. Estos ficheros se proporcionan al alumno.

En el código que aparece a continuación se implementan algunas de las operaciones propias de las tablas de dispersión. Es interesante destacar en este módulo cómo se define el tipo tabla. Ésta se define como un vector (de talla máxima `MAX_CUBETAS`) de punteros a nodos. En cada nodo se guardan la cadena ASCII (el elemento), un identificador numérico único para esa cadena y un puntero al siguiente nodo de la lista correspondiente a la cubeta donde se encuentre el nodo actual.

La talla del vector que representa la verdadera tabla hash (esto es, el número de cubetas) vendrá especificado por el usuario, y normalmente se guarda en una variable de nombre m.

Código de hash.h:

```
/* MODULO PARA LA MANIPULACION DE TABLAS HASH */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
/********************************************/
/* Definicion de constantes: */
#define MAX_TAM_PAL 25
#define MAX_TAM_LIN 100
#define MAX_CUBETAS 2000
/********************************************/
/* Definicion de tipos: */
/* Definicion del tipo tnodo */
typedef struct snodo {
  char palabra[MAX_TAM_PAL];   /* La cadena de la palabra almacenada. */
  int identificador;          /* El identificador de la palabra. */
  struct snodo *sig;          /* Puntero al siguiente nodo. */
} tnodo;
/* Definicion de ttabla = tabla hash: es un vector de tamanyo maximo */
/* MAX_CUBETAS de punteros a listas de nodos. */
typedef tnodo *ttabla[MAX_CUBETAS];
/********************************************/
/* Definicion de funciones: */
/* Inicializa los punteros de la tabla. */
void crearTabla(ttabla tabla);
/* Inserta la string cadena en la tabla hash asociandole el */
/* identificador que se especifica. */
void insertarTabla(ttabla tabla, char *cadena,
                        int identificador, int m, int funcion_hash);
/* Devuelve un puntero al nodo donde se encuentra la string cadena */
/* si esta en la tabla hash, si no, devuelve NULL. */
tnodo *perteneceTabla(ttabla tabla, char *cadena, int m, int funcion_hash);
/* Elimina la string cadena de la tabla hash. */
void eliminarTabla(ttabla tabla, char *cadena, int m, int funcion_hash);
/* Genera un fichero con la informacion del numero de elementos */
/* por cubeta. */
void estadisticaTabla(ttabla tabla, int m, char *fich_estad);
```

Código de hash.c:

```
/* MODULO PARA LA MANIPULACION DE TABLAS HASH */
#include "hash.h"
/***************************************************/
/* Definimos las 4 funciones hash vistas en clase: */
/*-----------------------------------------------*/
int func1(char *cadena, int m){
  unsigned int valor;
  int i;

  /* INSERTAR CODIGO PARA CALCULAR EL VALOR DE LA FUNCION 1 */
  /* ..... */
  /* ..... */
}
/*-----------------------------------------------*/
int func2(char *cadena, int m){
  unsigned int valor;
  int lon;

  valor=0;
  lon=strlen(cadena);
  if (lon == 1) strcat(cadena,"  ");
  else if (lon == 2) strcat(cadena," ");
       else if (lon == 0) {
               fprintf(stderr,"?Cadena de longitud 0?\n");
               exit(-1);
            }
  valor = (unsigned int) cadena[0]
       + 256 * (unsigned int) cadena[1]
       + 65025 * (unsigned int) cadena[2];
  return(valor % m);
}
/*-----------------------------------------------*/
int func3(char *cadena, int m){
  unsigned int valor;
  int i;

  valor=0;
  for(i=0;i<strlen(cadena);i++)
            valor = (valor * 256 + (unsigned int)cadena[i]) % m;
  return(valor);
}
```

```c
/*---------------------------------------------------*/
int func4(char *cadena, int m){
  #define A 0.6180339887498948482O      /* = (sqrt(5)-1)/2 */
  double valor, base;
  int i;

  valor=0.0;
  base=1;
  for(i=0; i<strlen(cadena); i++) {
        valor = valor + base * (unsigned int)cadena[i];
        base = base * 2;
  }
  return((int) floor(m * modf(valor * A, &base)));
}
/**********************************************************/
/* Calcula el valor hash para una determinada cadena segun la */
/* funcion hash que se haya elegido. */

int hashing(char *cadena, int m, int funcion_hash){
   switch(funcion_hash) {
      case 1: return(func1(cadena,m)); break;
      case 2: return(func2(cadena,m)); break;
      case 3: return(func3(cadena,m)); break;
      case 4: return(func4(cadena,m)); break;
      default: fprintf(stderr,"Error: Funcion hashing incorrecta\n");
               exit(-1);
               break;
   }
}
/**********************************************************/
/****************** FUNCIONES EXPORTABLES ******************/
/**********************************************************/
/* Inicializa los punteros de la tabla. */

void crearTabla(ttabla tabla){
  int i;

  /* INSERTAR CODIGO PARA CREAR LA TABLA HASH */
  /* .... */
}
```

```
/*********************************************************/
/* Inserta la string cadena en la tabla hash asociandole el */
/* identificador que se especifica. */

void insertarTabla(ttabla tabla, char *cadena,
                        int identificador, int m, int funcion_hash){
  tnodo *aux;
  int cubeta;

  /* INSERTAR CODIGO PARA ANYADIR cadena */
  /* A LA TABLA HASH */

  /* ...... */
  /* ...... */
  /* ...... */
}
/*********************************************************/
/* Devuelve un puntero al nodo donde se encuentra la string cadena */
/* si esta en la tabla hash, si no, devuelve NULL. */

tnodo *perteneceTabla(ttabla tabla, char *cadena, int m, int funcion_hash){
  tnodo *aux;
  int cubeta;

  /* INSERTAR CODIGO PARA COMPROBAR SI cadena */
  /* PERTENECE A LA TABLA HASH */

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

```
/*****************************************************/
/* Elimina la string cadena de la tabla hash. */

void eliminarTabla(ttabla tabla, char *cadena, int m, int funcion_hash){
  tnodo *aux,*aux1;
  int cubeta;

  /* INSERTAR CODIGO PARA ELIMINAR cadena */
  /* DE LA TABLA HASH */
  /* .... */
  /* .... */

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

```c
/*******************************************************/
/* Genera un fichero con la informacion del numero de elementos */
/* por cubeta. */

void estadisticaTabla(ttabla tabla, int m, char *fich_estad){
  FILE *f;
  double n, n_aux, n_des;
  int i;
  tnodo *aux;

  if ((f=fopen(fich_estad,"w"))==NULL) {
       fprintf(stderr,"No se puede abrir fichero %s\n",fich_estad);
       exit(-1);
  }
  n=0.0;
  n_des=0.0;
  for(i=0;i<m;i++){
    aux = tabla[i];
    n_aux=0.0;
    while (aux != NULL){
      n_aux+=1.0;
      aux = aux->sig;
    }
    n+=n_aux;
    n_des+=(n_aux*n_aux);
    fprintf(f,"%d %d\n",i,(int) n_aux);
  }

  fprintf(f,"# Media %e\n",n/m);
  fprintf(f,"# Desv. Tip. %e\n",sqrt((n_des - 2*n*n/m + n*n/m)/(m - 1)));
  fclose(f);
}
```

9.8. Códigos fuente para la Sección 9.4

Código de estadist.c:

```
/* PROGRAMA PRINCIPAL PARA CALCULAR ESTADISTICAS DE TABLAS HASH */
#include <stdio.h>
#include <conio.h>
#include "hash.h"
/*********************************************************************/
/* Lee un fichero de texto e introduce en la tabla hash todas las */
/* palabras del texto. */
void leer_fichero(char *fich_entrada, ttabla t, int m, int funcion_hash){
  FILE *f;
  char linea[MAX_TAM_LIN], cadena[MAX_TAM_PAL];
  int i,j,k;
  if ((f=fopen(fich_entrada,"r"))==NULL) {
        fprintf(stderr,"No se pudo abrir fichero %s\n", fich_entrada);
        exit(-1);
  }
  while (fgets(linea,MAX_TAM_LIN,f)!=NULL) {
      linea[strlen(linea)-1]='\0';
      i=0;
      while (i<strlen(linea)) {
          j=i;
          strcpy(cadena,"");
          while ((j<strlen(linea))&&((linea[j]==' ')||(linea[j]==''))) j++;
          k=0;
          while ((j<strlen(linea)) &&((linea[j]!=' ') && (linea[j]!=''))) {
                              cadena[k]=linea[j];
                              j++; k++;
          }
          cadena[k]='\0';
          if (k!=0) {
             if (perteneceTabla(t,cadena,m,funcion_hash)==NULL)
                 insertarTabla(t,cadena,0,m,funcion_hash);
          }
          i=j;
      }
  }
  fclose(f);
}
```

```
/*******************************************************************/
/**************** PROGRAMA PRINCIPAL ********************************/
/*******************************************************************/
int main() {
  ttabla t;
  char fich_entrada[MAX_TAM_PAL];
  char fich_estad[MAX_TAM_PAL];
  int m;
  int funcion_hash;
  int i;
  char respuesta[MAX_TAM_PAL];

  clrscr();
  printf("Cuantas cubetas quieres para la tabla hash? ");
  gets(respuesta);
  m = atoi(respuesta);
  printf("Que funcion hash quieres usar (1,2,3,4)? ");
  gets(respuesta);
  funcion_hash = atoi(respuesta);
  printf("Cual es el nombre del fichero de entrada? ");
  gets(fich_entrada);

  crearTabla(t);
  leer_fichero(fich_entrada, t, m, funcion_hash);

  strcpy(fich_estad,fich_entrada);
  i=0;
  while ((i<strlen(fich_estad)) && (fich_estad[i]!='.')) i++;
  fich_estad[i+1]='e'; fich_estad[i+2]='s'; fich_estad[i+3]=funcion_hash+48;
  fich_estad[i+4]='\0';

  /* INSERTAR CODIGO PARA LLAMAR A LA FUNCION */
  /* estadisticaTabla QUE GENERA UN FICHERO */
  /* DE ESTADISTICAS */
  /* ...... */

  return(0);
}
```

9.9. Códigos fuente para la Sección 9.5

Código de encripta.c:

```
/* PROGRAMA PRINCIPAL PARA ENCRIPTAR */
#include <stdio.h>
#include <conio.h>
#include "hash.h"
/*****************************************************************/
/* Lee un fichero de texto e introduce en la tabla hash todas las */
/* palabras del texto asignadoles un identificador numerico diferente */
/* a cada una de ellas. */
void leer_fichero(char *fich_entrada, ttabla t, int m, int funcion_hash){
  FILE *f;
  char linea[MAX_TAM_LIN], cadena[MAX_TAM_PAL];
  int i,j,k,nuevo_identificador;
  if ((f=fopen(fich_entrada,"r"))==NULL) {
        fprintf(stderr,"No se pudo abrir fichero %s\n", fich_entrada);
        exit(-1);
  }
  nuevo_identificador = 0;
  while (fgets(linea,MAX_TAM_LIN,f)!=NULL) {
      linea[strlen(linea)-1]='\0'; i=0;
      while (i<strlen(linea)) {
        j=i; strcpy(cadena,"");
        while((j<strlen(linea))&&((linea[j]==' ')||(linea[j]==''))) j++;
        k=0;
        while((j<strlen(linea))&&((linea[j]!=' ')&&(linea[j]!=''))){
                                cadena[k]=linea[j];
                                j++; k++;
        }
        cadena[k]='\0';
        if (k!=0) {
           /* INSERTAR CODIGO PARA VER SI LA CADENA YA ESTA */
           /* EN LA TABLA HASH; SI NO, ANYADIRLA CON EL */
           /* IDENTIFICADOR ADECUADO Y ACTUALIZAR nuevo_identificador */
           /* ....... */
        }
        i=j;
      }
  }
  fclose(f);
}
```

```
/************************************************************************/
/* Lee un fichero de texto y partir de los identificadores que le */
/* proporciona la tabla hash para cada una de las palabras, genera */
/* un fichero de salida con los identificadores de las palabras */
/* y otro fichero de salida que relaciona cada palabra con su */
/* identificador. */
void encripta_fichero(char *fich_entrada, ttabla t,
                                          int m, int funcion_hash){
  FILE *f_input, *f_output, *f_vocab;
  char fich_salida[MAX_TAM_PAL], fich_vocab[MAX_TAM_PAL];
  char linea[MAX_TAM_LIN], cadena[MAX_TAM_PAL];
  int i,j,k;
  tnodo *aux;

  strcpy(fich_salida,fich_entrada); i=0;
  while ((i<strlen(fich_salida)) && (fich_salida[i]!='.')) i++;
  fich_salida[i+1]='o'; fich_salida[i+2]='u'; fich_salida[i+3]='t';
  fich_salida[i+4]='\0';
  strcpy(fich_vocab,fich_entrada); i=0;
  while ((i<strlen(fich_vocab)) && (fich_vocab[i]!='.')) i++;
  fich_vocab[i+1]='v'; fich_vocab[i+2]='o'; fich_vocab[i+3]='c';
  fich_vocab[i+4]='\0';
  if  ((f_input=fopen(fich_entrada,"r"))==NULL) {
        fprintf(stderr,"No se pudo abrir fichero %s\n", fich_entrada);
        exit(-1);
  }
  if  ((f_output=fopen(fich_salida,"w"))==NULL) {
        fprintf(stderr,"No se pudo abrir fichero %s\n", fich_salida);
        exit(-1);
  }
```

```
  while (fgets(linea,MAX_TAM_LIN,f_input)!=NULL) {
      linea[strlen(linea)-1]='\0';
      i=0;
      while (i<strlen(linea)) {
        j=i;
        strcpy(cadena,"");
        while((j<strlen(linea))&&((linea[j]==' ')||(linea[j]==''))) j++;
        k=0;
        while((j<strlen(linea))&&((linea[j]!=' ')&&(linea[j]!=''))){
                              cadena[k]=linea[j];
                              j++; k++;
        }
        cadena[k]='\0';
        if (k!=0) {
             aux = perteneceTabla(t,cadena,m,funcion_hash);
             if (aux != NULL) fprintf(f_output,"%d ",aux->identificador);
             else {
                  fprintf(stderr,"Palabra %s desconocida\n",cadena);
                  exit(-1);
             }
        }
        i=j;
      }
      fprintf(f_output,"\n");
  }
  fclose(f_input);  fclose(f_output);
  if ((f_vocab=fopen(fich_vocab,"w"))==NULL) {
     fprintf(stderr,"No se pudo abrir fichero %s\n", fich_vocab); exit(-1);
  }
  fprintf(f_vocab,"m= %d\n",m);
  fprintf(f_vocab,"funcion_hash= %d\n",funcion_hash);
  fprintf(f_vocab,"============= ============\n");
  for(i=0;i<m;i++) {
         aux = t[i];
         while (aux!=NULL) {
              fprintf(f_vocab,"%s %d\n",aux->palabra,aux->identificador);
              aux = aux->sig;
         }
  }
  fclose(f_vocab);
}
```

```
/*********************************************************************/
/**************** PROGRAMA PRINCIPAL ********************************/
/*********************************************************************/
int main() {
  ttabla t;
  char fich_entrada[MAX_TAM_PAL];
  int m;
  int funcion_hash;
  char respuesta[MAX_TAM_PAL];

  clrscr();
  printf("Cuantas cubetas quieres para la tabla hash? ");
  gets(respuesta);
  m = atoi(respuesta);

  printf("Que funcion hash quieres usar (1,2,3,4)? ");
  gets(respuesta);
  funcion_hash = atoi(respuesta);

  printf("Cual es el nombre del fichero de entrada? ");
  gets(fich_entrada);

  /* INSERTAR CODIGO PARA CREAR LA TABLA HASH, */
  /* LLAMAR A LA FUNCION leer_fichero PARA CREAR */
  /* EL VOCABULARIO, Y LLAMAR A encripta_fichero */
  /* PARA ENCRIPTAR EL FICHERO DE ENTRADA */
  /* ..... */
  /* ..... */
  /* ..... */

  return(0);
}
```

9.10. Códigos fuente para la Sección 9.6

Código de desencri.c:

```
/* PROGRAMA PRINCIPAL PARA DESENCRIPTAR */
#include <stdio.h>
#include <conio.h>
#include "hash.h"
/*******************************************************************/
/* Lee un fichero que contiene las palabras del vocabulario y el */
/* identificador correspondiente a cada palabra y rellena la */
/* tabla hash con estas palabras y sus identificadores. */

void leer_vocabulario(char *fich_vocab, ttabla t,
                                          int *m, int *funcion_hash){
  FILE *f_vocab;
  char linea[MAX_TAM_LIN];
  char cadena1[MAX_TAM_PAL], cadena2[MAX_TAM_PAL];
  int aux;

  if ((f_vocab=fopen(fich_vocab,"r"))==NULL) {
         fprintf(stderr,"No se pudo abrir fichero %s\n", fich_vocab);
         exit(-1);
  }
  while (fgets(linea,MAX_TAM_LIN,f_vocab)!=NULL) {
         linea[strlen(linea)-1]='\0';
         sscanf(linea,"%s %s",cadena1,cadena2);
         if (strcmp(cadena1,"=============")!=0) {
            aux = atoi(cadena2);
            if (strcmp(cadena1,"m=")==0) *m = aux;
            else if (strcmp(cadena1,"funcion_hash=")==0)
                        *funcion_hash = aux;
                 else insertarTabla(t,cadena1,aux,*m,*funcion_hash);
         }
  }
  fclose(f_vocab);
}
```

```
/*****************************************************************/
/* Lee un fichero de identificadores de palabras y por medio de la */
/* tabla hash genera un fichero de salida con las palabras */
/* correspondientes a las palabras */
void desencripta_fichero(char *fich_entrada, ttabla t,
                                                  int m, int funcion_hash){
  FILE *f_input, *f_output;
  char fich_salida[MAX_TAM_PAL], linea[MAX_TAM_LIN], cadena[MAX_TAM_PAL];
  char *palabras[6000];
  tnodo *aux;
  int id,i,j,k;

  strcpy(fich_salida,fich_entrada); i=0;
  while ((i<strlen(fich_salida)) && (fich_salida[i]!='.')) i++;
  fich_salida[i+1]='d'; fich_salida[i+2]='e'; fich_salida[i+3]='s';
  fich_salida[i+4]='\0';
  for (i=0;i<m;i++) {
      aux = t[i];
      while (aux != NULL) {
           palabras[aux->identificador] =
                             (char *) malloc(strlen(aux->palabra)+1);
           strcpy(palabras[aux->identificador],aux->palabra);
           aux = aux->sig;
      }
  }
  if ((f_input=fopen(fich_entrada,"r"))==NULL) {
         fprintf(stderr,"No se pudo abrir fichero %s\n", fich_entrada);
         exit(-1);
  }
  if ((f_output=fopen(fich_salida,"w"))==NULL) {
         fprintf(stderr,"No se pudo abrir fichero %s\n", fich_salida);
         exit(-1);
  }
```

```
  while (fgets(linea,MAX_TAM_LIN,f_input)!=NULL) {
      linea[strlen(linea)-1]='\0';
      i=0;
      while (i<strlen(linea)) {
        j=i;
        strcpy(cadena,"");
        while((j<strlen(linea))&&((linea[j]==' ')||(linea[j]==''))) j++;
        k=0;
        while((j<strlen(linea))&&((linea[j]!=' ')&&(linea[j]!=''))){
                              cadena[k]=linea[j];
                              j++; k++;
        }
        cadena[k]='\0';
        if (k!=0) {
               id = atoi(cadena);
               fprintf(f_output,"%s ",palabras[id]);
        }
        i=j;
      }
      fprintf(f_output,"\n");
  }
  fclose(f_input);  fclose(f_output);
}
```

```
/*****************************************************************/
/*************** PROGRAMA PRINCIPAL *******************************/
/*****************************************************************/
int main() {
  ttabla t;
  char fich_entrada[MAX_TAM_PAL];
  char fich_vocab[MAX_TAM_PAL];
  int m;
  int funcion_hash;

  clrscr();
  printf("Cual es el nombre del fichero de vocabulario? ");
  gets(fich_vocab);

  printf("Cual es el nombre del fichero encriptado? ");
  gets(fich_entrada);

  /* INSERTAR CODIGO PARA CREAR LA TABLA HASH, */
  /* LLAMAR A LA FUNCION leer_vocabulario PARA */
  /* CREAR EL VOCABULARIO, Y LLAMAR A */
  /* desencripta_fichero PARA DESENCRIPTAR EL */
  /* FICHERO DE ENTRADA */
  /* ..... */
  /* ..... */
  /* ..... */

  return(0);
}
```

Práctica 10

Estudio de Montículos (Heaps)

10.1. Introducción

En esta práctica se pretende estudiar la estructura de datos Montículo. Los objetivos concretos del trabajo son los siguientes:

- Conocer una variante de la estructura de datos heap vista en clase de teoría donde cambia la condición de montículo.
- Conocer el algoritmo de ordenación Heapsort.
- Aplicar la estructura de datos montículo a un problema concreto como es implementar y gestionar una cola de prioridad.

La práctica está organizada en varias sesiones dedicadas a los aspectos comentados anteriormente y está formada por tres partes claramente diferenciadas: una primera parte (sección 10.2) que introducirá la variante de los heaps que se utilizará durante el resto de la práctica, una segunda parte (sección 10.3) que consiste en la comprobación experimental del algoritmo de ordenación de vectores *Heapsort* y una última parte (sección 10.4) que consiste en implementar una cola de prioridad mediante un heap para controlar la ejecución de procesos en una CPU.

Todos los códigos fuente mostrados en las secciones finales para las diferentes partes de la práctica serán proporcionados al alumno.

10.2. Montículos (Heaps)

La estructura de datos **Montículo (Heap)** vista en clase de teoría se corresponde realmente con lo que se conoce como **MaxHeap**. Esto es debido

a que la propiedad de montículo especifica que un nodo interno del árbol tiene que ser mayor o igual que sus hijos, con lo que conforme subamos de nivel encontraremos valores más altos en los nodos del árbol, además podremos decir que el elemento máximo estará almacenado en la raíz.

En esta práctica vamos a trabajar con **MinHeaps**. Un MinHeap es un montículo (por tanto árbol binario completo) donde la propiedad de montículo especifica que la clave de un nodo interno del árbol tiene que ser menor o igual a las claves de sus hijos, con lo que conforme subamos de nivel encontraremos valores más pequeños y podemos asegurar que el elemento mínimo estará en la raíz. La estructura MinHeap se implementará igual que el MaxHeap; con un vector para el árbol binario completo y con una variable `talla_Heap` que indica el número de nodos del montículo. Podemos ver un ejemplo de MinHeap en la figura 10.1.

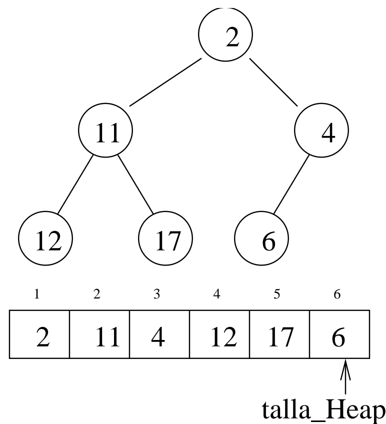


Figura 10.1: Ejemplo de MinHeap

A partir de ahora y durante el transcurso de esta práctica, cuando hablemos de un Heap o Montículo, nos estaremos refiriendo a un MinHeap.

10.3. El algoritmo *Heapsort*

Se ha implementado una estructura de datos MinHeap cuyos elementos son números enteros. La implementación se encuentra en los módulos `heap.h` y `heap.c` (ver la sección 10.5).

En estos módulos se implementan algunas de las operaciones propias de los montículos. Es interesante destacar en este módulo cómo se define el tipo heap. Es un vector (de talla máxima `MAX_TAM_HEAP`) de enteros.

La talla del heap, esto es, el número de elementos, se guardará en una variable `talla_Heap` que nos indica hasta qué posición hay elementos válidos en el vector. La posición 0 del vector no se usa para nada.

En el módulo `sortheap.c` (ver código en sección 10.6) se hace uso de las funciones implementadas en `heap.h` y `heap.c` para manejo de montículos y existe un programa principal que pregunta de qué talla se desea el vector a ordenar, después genera un vector aleatorio de ese tamaño llamando a la función `random_vector()` y lo muestra llamando a la función `show_vector()`. Por último, lo ordena usando el método Heapsort y muestra el vector ordenado.

La estrategia de Heapsort es la misma que la vista en clase de teoría, pero en este caso como usamos un MinHeap obtendremos un vector ordenado de mayor a menor en vez de ordenado de menor a mayor que es como normalmente hemos trabajado con métodos de ordenación. Esto será así puesto que el algoritmo Heapsort intercambiará la raíz del montículo (el elemento mínimo) con la última posición del vector (el último nodo del heap), reducirá en uno la talla del montículo y aplicará heapify sobre la nueva raíz para que cumpla la propiedad de montículo. Si aplicamos heapsort al vector (montículo) presentado como ejemplo antes obtendremos:

1	2	3	4	5	6
17	12	11	6	4	2

Si nos fijamos en las tres funciones necesarias a implementar para poder aplicar el método de ordenación Heapsort:

- La función `heapsort()` seguirá el mismo esquema que el visto en clase de teoría para MaxHeaps, puesto que esta función no comprueba directamente la condición de montículo, que es lo único que cambia respecto a un MaxHeap. Por tanto, no será necesario modificar `heapsort()`.

- La función `buildheap()` es usada por `heapsort()`, sin embargo, tampoco será necesario un cambio respecto a la versión vista para MaxHeaps en teoría, puesto que tampoco comprueba directamente la condición de montículo.

- La función `heapify()` también es usada por `heapsort()`, pero en este caso sí que habrá que cambiar la función, puesto que ahora la condición de montículo que debe comprobar `heapify()` es justo la contraria que para MaxHeaps. El resto del esquema de esta función es idéntico al visto en teoría.

Ejercicio: Incluir el código necesario en la función `heapify()` del módulo `heap.c` para que Heapsort pueda funcionar correctamente. Ejecutar el programa `sortheap.c` (tendrás que crear un proyecto en Turbo C) y ver como se ordenan los vectores de mayor a menor.

10.4. Implementación de una cola de prioridad: Simulación de una CPU

Vamos a simular una CPU manejada mediante un sistema operativo UNIX, que atiende (ejecuta) los procesos que se van lanzando en un computador. Los ejecuta conforme se van lanzando pero teniendo en cuenta el tiempo en que son lanzados a ejecución y su prioridad de ejecución. Para hacer la simulación se mostrará por pantalla el nombre del proceso que se esté ejecutando en un determinado momento.

En UNIX, cuanto menor sea el número asignado a un proceso, mayor será su prioridad.

Existirá una cola de prioridad para gestionar el orden de ejecución de los procesos. Esta cola de prioridad estará implementada mediante un MinHeap, donde la clave de cada elemento es la prioridad y las claves más pequeñas indican mayor prioridad. Podrá haber claves repetidas (procesos con la misma prioridad). Cuando en la cola haya dos procesos con el mismo valor de prioridad, se considerará más prioritario (estará más cerca de la raíz del montículo) el más antiguo en el tiempo (el que haya entrado antes en la cola de prioridad).

La cola de prioridad va a estar implementada mediante un MinHeap que en este caso va a ser un registro que contendrá un vector `prioridad` donde se guardaran las claves (prioridades) de los procesos formando un montículo, otro vector `ident`, donde en la posición `i` estará el identificador del proceso cuya prioridad esté en la posición `i` del vector de prioridades (más adelante veremos cual es el identificador que vamos a asignar a cada proceso); además habrá una variable `talla_Heap` que indicará la talla del heap (número de procesos en la cola). La implementación de esta estructura de datos y sus operaciones puede verse en los módulos de código `colapri.h` y `colapri.c` que se muestran en el la sección 10.7.

La entrada al programa es un fichero donde se especifican los nombres de los procesos, su prioridad, el instante de tiempo en que se lanzan a ejecución y el tiempo que tardan en ejecutarse.

Las normas para especificar la planificación de tareas en este fichero son:

- Las líneas que comiencen por el caracter # son comentarios.
- No se pueden lanzar dos procesos al mismo tiempo (en la misma unidad de tiempo).
- No se admiten tiempos de lanzamiento, prioridades y tiempos de ejecución negativos.
- Cada línea que especifica un proceso tiene (por este orden):
  - Un número indicando el tiempo en que se lanza a ejecución el proceso.
  - Nombre del proceso.
  - Un número que indica la prioridad del proceso.
  - Un número que indica el número de unidades de tiempo que el proceso requiere para su ejecución.

Un fichero de entrada ejemplo podría contener esta planificación de procesos:

```
# TIEMPO   PROCESO    PRIORIDAD  UNIDADES_DE_TIEMPO_DE_EJECUCION
  2        guindous   10         3
  4        linups     5          1
  5        guord      3          3
  6        notescapes 4          2
```

Inicialmente, el programa leerá este fichero y lo representará en un vector en memoria guardando los datos de cada proceso en la posición del vector que indique su tiempo de lanzamiento a ejecución de esta manera:

0	1	2	3	4	5	6	7
		guindous 10 3		linups 5 1	guord 3 3	notescapes 4 2	

Esto servirá para que el programa tenga una representación manejable de los procesos. Aquí podemos decir que como cada proceso estará en una posición del vector determinada que indica su tiempo de lanzamiento, entonces podremos tomar esa posición como identificador del proceso, es decir, el identificador de un proceso será su tiempo de lanzamiento y por ello también se guardará en el campo de identificador correspondiente para cada uno de los elementos (procesos) que haya en la cola de prioridad.

Después de crear este vector, simularemos un bucle de tiempo, en el cual aplicaremos el algoritmo de gestión y ejecución de procesos. Este algoritmo consiste en:

1. Si llega un proceso nuevo, entonces se introduce en la cola de prioridad.
2. Si no hay ningún proceso ejecutándose, entonces se extrae de la cola el más prioritario y pasa a ejecutarse.
3. Se ejecuta una unidad de tiempo del proceso correspondiente que esté en ejecución..
4. Si el proceso acaba en esta unidad de tiempo, entonces deja libre la CPU para el siguiente ciclo de reloj.

Este algoritmo y la lectura del fichero de planificación de procesos se encuentran implementados en el módulo `cpu.c`, cuyo código está disponible en la sección 10.8. Además, este programa hace uso de las librerías `screen.h` y `screen.c` (cuyo código también se muestra en la sección 10.8), que implementan funciones para poder imprimir en pantalla los sucesos que emulan la CPU de nuestro computador ficticio, permitiendo una visualización agradable al usuario de la emulación de procesos.

La salida por pantalla correspondiente a la simulación del programa para el fichero de planificación de procesos mostrado más arriba sería:

```
              Pulse Ctrl-c para parar

TIEMPO   PROCESO        SUCESO
======   ========       =================================
0
1
2        guindous       LLEGA guindous   PRI:10   TIEMPO:3
3        guindous
4        guindous       LLEGA linups   PRI:5   TIEMPO:1
5        guord          LLEGA guord   PRI:3   TIEMPO:3
6        guord          LLEGA notescapes   PRI:4   TIEMPO:2
7        guord
8        notescapes
9        notescapes
10       linups
11
12
13
.....
```

Ejercicio: Se deja al alumno la implementación del código correspondiente a las funciones de la cola de prioridad: `crea_heap()`, `vacio()`, `insertar()`, `extrae_min()` y `heapify()` que se encuentran en el módulo `colapri.c`. Estas funciones permiten a la cola de prioridad comportarse como un MinHeap, por ello, la implementación es muy parecida a la vista en clase de teoría para MaxHeaps. Sólo hay que tener en cuenta dos aspectos:

- La condición de montículo es la inversa a la de los MaxHeaps.
- En la cola de prioridad, un elemento está formado por su clave (prioridad del proceso) y el identificador del proceso correspondiente, así pues, cuando se realice alguna operación que cambie el vector de claves, también deberá cambiarse el vector de identificadores.

Despues el alumno debe ejecutar el programa `cpu.c`, para lo cual será necesario crear un proyecto en Turbo C y probarlo con varios ficheros de planificación de proyectos. Se proporcionan dos ficheros de planificación como ejemplo llamados `proyectos` y `proyectos.2`.

10.5. Librería para Montículos (Heap)

A continuación se presentan los módulos necesarios para implementar el tipo abstracto de datos **Montículo (MinHeap)** en una librería.

Código de heap.h:

```
/* MODULO PARA LA DEFINICION DE MINHEAPS DE ENTEROS */
#include <stdio.h>
#include <stdlib.h>

/*******************************************/
/* Definicion de constantes: */

#define MAX_TAM_HEAP 2000

/*******************************************/
/* Definicion de tipos: */

/* Definicion del tipo theap */
typedef int theap[MAX_TAM_HEAP];

/*******************************************/
/* Definicion de funciones: */

/* Heapify: mantiene la propiedad de monticulo para */
/* el subarbol que parte del nodo especificado. */
void heapify(theap heap, int i, int talla_Heap);

/* build_heap: dado un vector con n elementos, */
/* lo transforma en un monticulo. */
void build_heap(theap heap, int talla_Heap);

/* heapsort: dado un vector, lo ordena de mayor a menor. */
void heapsort(theap heap, int talla_Heap);
```

Código de heap.c:

```
/* MODULO PARA LA MANIPULACION DE MINHEAPS DE ENTEROS */
#include "heap.h"

/*******************************************************/
/***************** FUNCIONES EXPORTABLES *****************/
/*******************************************************/

/*******************************************************/
/* Heapify: mantiene la propiedad de monticulo para */
/* el subarbol que parte del nodo especificado. */

void heapify(theap heap, int i, int talla_Heap){
  int aux;
  int hizq, hder, menor;

  /* INSERTAR CODIGO DE HEAPIFY. */
  /* ES MUY PARECIDO AL HEAPIFY DE UN */
  /* MAXHEAP, PERO VARIA LA CONDICION */
  /* DE MONTICULO. */
  /* ..... */
  /* ..... */
  /* ..... */
}


/*******************************************************/
/* build_heap: dado un vector con n elementos, */
/* lo transforma en un monticulo. */

void build_heap(theap heap, int talla_Heap){
  int i;

  for (i=talla_Heap/2; i>0; i--) heapify(heap, i, talla_Heap);
}
```

```
/**********************************************************/
/* heapsort: dado un vector, lo ordena de mayor a menor. */

void heapsort(theap heap, int talla_Heap){
  int i;
  int aux;

  build_heap(heap, talla_Heap);

  for (i=talla_Heap; i>1; i--) {
      aux = heap[i];
      heap[i] = heap[1];
      heap[1] = aux;
      talla_Heap--;
      heapify(heap, 1, talla_Heap);
  }
}
```

10.6. Códigos fuente para la sección 10.3

Código de sortheap.c:

```
/* PROGRAMA PRINCIPAL PARA ORDENAR VECTORES MEDIANTE */
/* HEAPSORT */
#include <stdio.h>
#include <time.h>
#include <conio.h>
#include "heap.h"
/*************** FUNCIONES PARA VECTORES DE ENTEROS *************/
/* OJO! : CONSIDERAMOS QUE LOS VECTORES EMPIEZAN EN LA */
/* POSICION 1, COMO LOS MONTICULOS VISTOS EN CLASE DE */
/* TEORIA */
/**********************************************/
/* Muestra un vector y dice si esta ordenado. */
void show_vector(int *v, int size){
  int i;
  int ordered;

  ordered = 0;
  printf("\n");
  for (i=1; i<=size; i++) {
      printf("%d ",v[i]);
      if (i>1) if (v[i]>v[i-1]) ordered = 1;
  }
  if (ordered == 0) printf("\n  --> ORDENADO\n");
  else printf("\n  --> DESORDENADO\n");
}
/**********************************************/
/* Genera un vector aleatorio de talla size. */
void random_vector(int *v, int size){
  time_t t;
  int i;
  float aux;

  if (size < MAX_TAM_HEAP) {
     aux = (float) MAX_TAM_HEAP/2;
     srand((unsigned) time(&t));
     for (i=1; i<=size; i++)
                v[i] = 1+(int) (aux*rand()/(RAND_MAX+1.0));
  }
}
```

```
/*****************************************************************/
/*************** PROGRAMA PRINCIPAL ****************************/
/*****************************************************************/
int main() {
  theap heap;
  int talla_Heap;

  clrscr();
  /* Creamos un vector de la talla especificada por el usuario. */
  printf("De que talla quieres el vector? ");
  scanf("%d",&talla_Heap);
  random_vector(heap, talla_Heap);
  show_vector(heap, talla_Heap);

  /* Lo ordenamos usando el metodo heapsort() */
  heapsort(heap, talla_Heap);
  show_vector(heap, talla_Heap);

  printf("Pulse una tecla"); getch();
  return(0);
}
```

10.7. Librería para Colas de Prioridad

Código de colapri.h:

```
/* MODULO PARA LA DEFINICION DE MINHEAPS COMO COLA DE */
/* PRIORIDAD DE PROCESOS */
#include <stdio.h>
#include <stdlib.h>
/*********************************************/
/****** Definicion de constantes: ******/
#define MAX_TAM_HEAP 500
/*********************************************/
/****** Definicion de tipos: ******/
/* Definicion del tipo theap. */
typedef struct {
        /* Vector para representar el verdadero heap */
        /* donde las claves son las prioridades. */
        int prioridad[MAX_TAM_HEAP];
        /* Vector para asociar cada identificador de */
        /* proceso con su prioridad. */
        int ident[MAX_TAM_HEAP];
        int talla_Heap;
} theap;
/*********************************************/
/****** Definicion de funciones: ******/
/* Crea una cola de prioridad vacia. */
theap *crea_heap();

/* Devuelve 0 si el heap no esta vacio y 1 si el heap esta vacio. */
int vacio(theap *heap);

/* Inserta un nuevo elemento siendo su prioridad la clave. */
void insertar(theap *heap, int prioridad, int identificador);

/* Extrae el elemento de minima prioridad del heap, */
/* sera el elemento de la raiz. Devuelve el identificador del proceso. */
int extrae_min(theap *heap);

/* Heapify: mantiene la propiedad de monticulo para */
/* el subarbol que parte del nodo especificado. */
void heapify(theap *heap, int i);
```

Código de colapri.c:

```
/* MODULO PARA LA DEFINICION DE MINHEAPS COMO COLA DE */
/* PRIORIDAD DE PROCESOS */
#include "colapri.h"

/*****************************************************/
/***************** FUNCIONES EXPORTABLES *****************/
/*****************************************************/

/*****************************************************/
/* Crea una cola de prioridad vacia. */

theap *crea_heap(){
  theap *aux;
  int i;

  /* INSERTAR CODIGO DE crea_heap. */
  /* RESERVAR MEMORIA E INICIALIZAR. */
  /* ..... */
  /* ..... */
  /* ..... */
}


/*****************************************************/
/* Devuelve 0 si el heap no esta vacio y 1 si el heap */
/* esta vacio. */

int vacio(theap *heap){
  /* INSERTAR CODIGO DE vacio. */
  /* ..... */
}
```

```
/*******************************************************/
/* Inserta un nuevo elemento siendo su prioridad la clave. */
void insertar(theap *heap, int prioridad, int identificador){
  int i;

  /* INSERTAR CODIGO DE insertar. */
  /* ES MUY PARECIDO AL insertar DE UN */
  /* MAXHEAP, PERO VARIA LA CONDICION */
  /* DE MONTICULO. */
  /* NOTA: AQUI EL HEAP GUARDA MAS INFORMACION */
  /* QUE LA CLAVE. */
  /* ..... */
  /* ..... */
}
/*******************************************************/
/* Extrae el elemento de minima prioridad del heap, */
/* sera el elemento de la raiz. */
/* Devuelve el identificador del proceso. */
int extrae_min(theap *heap){
  int ident_min;

  /* INSERTAR CODIGO DE extrae_min. */
  /* ES MUY PARECIDO AL extract_max DE UN */
  /* MAXHEAP. */
  /* NOTA: AQUI EL HEAP GUARDA MAS INFORMACION */
  /* QUE LA CLAVE. */
  /* ..... */
  /* ..... */
}
/*******************************************************/
/* Heapify: mantiene la propiedad de monticulo para */
/* el subarbol que parte del nodo especificado. */
void heapify(theap *heap, int i){
  int aux_prio, aux_ident;
  int hizq, hder, menor;

  /* INSERTAR CODIGO DE HEAPIFY. */
  /* ES MUY PARECIDO AL HEAPIFY DE UN */
  /* MAXHEAP, PERO VARIA LA CONDICION */
  /* DE MONTICULO. */
  /* ..... */
  /* ..... */
}
```

10.8. Códigos fuente para la sección 10.4

Código de screen.h:

```
/* MODULO PARA LA MANIPULACION DE LA PANTALLA */
/* EMULANDO UNA CPU */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/*****************************************************/
/* CONSTANTES y VARIABLES GLOBALES PARA IMPRIMIR EN PANTALLA */
#define T 0
#define P 9
#define S 24

/* Linea para imprimir en la pantalla. */
static char pantalla[80];
/* Linea para el suceso (si llega un proceso). */
static char suceso[56];

/*******************************************************/
/* FUNCIONES EXPORTABLES */
/********************************************/
/* Limpia la cadena suceso. */
void limpia_suceso();
/********************************************/
/* Crea la cadena suceso con los datos correspondientes. */
void crea_suceso(char *nombre, int prioridad, int tiempo);
/********************************************/
/* Limpia la cadena a imprimir por pantalla. */
void limpia_pantalla();
/********************************************/
/* Copia una substring cad en la posicion indicada de pantalla. */
void put_substring(int pos, char *cad);
/********************************************/
/* Imprime la cabecera de la ejecucion de procesos. */
void print_head();
/********************************************/
/* Imprime una linea con el tiempo, proceso y suceso */
/* especificados. */
void print_pantalla(int t, char *p);
```

Código de screen.c:

```
/* MODULO PARA LA MANIPULACION DE LA PANTALLA */
/* EMULANDO UNA CPU */
#include "screen.h"
/**************************************************/
/**************  FUNCIONES EXPORTABLES *************/
/*******************************************/
/* Limpia la cadena suceso. */

void limpia_suceso(){
  int i;
  for(i=0;i<53;i++) suceso[i]=' ';
  suceso[53]='\0';
}

/*******************************************/
/* Crea la cadena suceso con los datos correspondientes. */

void crea_suceso(char *nombre, int prioridad, int tiempo){
  limpia_suceso();
  sprintf(suceso,"LLEGA %s   PRI:%d   TIEMPO:%d",nombre,prioridad,tiempo);
}

/*******************************************/
/* Limpia la cadena a imprimir por pantalla. */

void limpia_pantalla(){
  int i;
  for(i=0;i<79;i++) pantalla[i]=' ';
  pantalla[79]='\0';
}

/*******************************************/
/* Copia una substring en la posicion indicada de pantalla */

void put_substring(int pos, char *cad){
  int i;
  for(i=pos; i<(pos+strlen(cad)); i++)
      pantalla[i] = cad[i-pos];
}
```

```
/*********************************************/
/* Imprime la cabecera de la ejecucion de procesos. */

void print_head(){
  printf("\n                  Pulse Ctrl-c para parar\n\n");
  limpia_pantalla();
  put_substring(T,"TIEMPO");
  put_substring(P,"PROCESO");
  put_substring(S,"SUCESO");
  printf("%s\n",pantalla);
  limpia_pantalla();
  put_substring(T,"======");
  put_substring(P,"========");
  put_substring(S,"=================================");
  printf("%s\n",pantalla);
}


/*********************************************/
/* Imprime una linea con el tiempo, proceso y suceso */
/* especificados. */

void print_pantalla(int t, char *p){
  char tt[5];

  limpia_pantalla();
  sprintf(tt,"%d",t);
  put_substring(T,tt);
  put_substring(P,p);
  put_substring(S,suceso);
  printf("%s\n",pantalla);
  limpia_suceso();
}
```

Código de cpu.c:

```c
/* PROGRAMA PRINCIPAL PARA SIMULAR UNA CPU QUE ATIENDE */
/* PROCESOS BASANDOSE EN SU PRIORIDAD */
#include <stdio.h>
#include <conio.h>
#include "colapri.h"
#include "screen.h"

/****************************************************/
/****************** TIPOS DE DATOS ******************/

/* Estructura de datos para guardar la informacion de */
/* un proceso. */

typedef struct {
  char nombre[20];  /* Nombre del proceso. */
  int prioridad;    /* Prioridad del proceso. */
  int tiempo;       /* Unidades de tiempo que tarda */
                    /* en ejecutarse. */
} tproceso;

/****************************************************/
/******************** FUNCIONES *******************/

/*******************************************/
/* Genera una parada de ejecucion durante un */
/* tiempo breve. */

void sleep_1(){
  int i, j, k;   double x, a1, a2, a3;
  x = 200.0;     a1 = 0.0;
  for(i=1; i<=(int)x; i++) {
     a2 = 1.0;
     for(j=1;j<=2*i;j++) {
         a3 = 0.0;
         for(k=1;k<=3*j;k++) a3 = (a3 + x)/x;
         a2 = a2 * a3;
     }
     a1 = a1 + a2;
  }
}
```

```c
/******************************************/
/* Lee el fichero de procesos y lo representa en el */
/* vector procesos */

void lee_fichero_proc(char *fichero_procesos,
                                        tproceso *procesos){
  FILE *f;
  char l[100];    char cadena[100];
  int i,j;
  int t_lanzado;

  /* Inicializamos el vector de procesos */
  for(i=0;i<MAX_TAM_HEAP;i++) {
     strcpy(procesos[i].nombre,"");
     procesos[i].prioridad = -1;
     procesos[i].tiempo = -1;
  }

  if ((f=fopen(fichero_procesos,"r"))==NULL) {
     fprintf(stderr,"ERROR: no se puede abrir fichero %s\n",fichero_procesos);
     exit(1);
  }

  while (fgets(l,100,f)!=NULL) {
    l[strlen(l)-1]='\0';
    if (l[0] != '#') {
      /* Leemos tiempo lanzamiento */
      i=0;
      strcpy(cadena,"");
      while ((i<strlen(l))&&((l[i]==' ')||(l[i]==''))) i++;
      j=0;
      while ((j+i<strlen(l))&&((l[j+i]!=' ')&&(l[j+i]!=''))) {
                        cadena[j]=l[j+i];    j++;
      }
      cadena[j]='\0';
      if (j!=0) t_lanzado = atoi(cadena);
```

```
      /* Leemos nombre proceso */
      i=i+j;
      strcpy(cadena,"");
      while ((i<strlen(l))&&((l[i]==' ')||(l[i]==''))) i++;
      j=0;
      while ((j+i<strlen(l))&&((l[j+i]!=' ')&&(l[j+i]!=''))) {
                      cadena[j]=l[j+i];    j++;
      }
      cadena[j]='\0';
      if  (j!=0) strcpy(procesos[t_lanzado].nombre,cadena);

      /* Leemos prioridad proceso */
      i=i+j;
      strcpy(cadena,"");
      while ((i<strlen(l))&&((l[i]==' ')||(l[i]==''))) i++;
      j=0;
      while ((j+i<strlen(l))&&((l[j+i]!=' ')&&(l[j+i]!=''))) {
                      cadena[j]=l[j+i];    j++;
      }
      cadena[j]='\0';
      if  (j!=0) procesos[t_lanzado].prioridad = atoi(cadena);

      /* Leemos tiempo ejecucion proceso */
      i=i+j;
      strcpy(cadena,"");
      while ((i<strlen(l))&&((l[i]==' ')||(l[i]==''))) i++;
      j=0;
      while ((j+i<strlen(l))&&((l[j+i]!=' ')&&(l[j+i]!=''))) {
                      cadena[j]=l[j+i];    j++;
      }
      cadena[j]='\0';
      if  (j!=0) procesos[t_lanzado].tiempo = atoi(cadena);
    }
  }
  fclose(f);
}
```

```
/*****************************************************************/
/**************** PROGRAMA PRINCIPAL *****************************/
/*****************************************************************/
int main() {
  char fichero_procesos[100];
  int i;
  tproceso vprocesos[MAX_TAM_HEAP];
  /* Vector que guarda la informacion proporcionada por */
  /* el fichero de procesos referente a los procesos */
  /* lanzados. */
  /* NOTA: el identificador de cada proceso es la posicion */
  /* donde se guarda en este vector (que sera el tiempo */
  /* en que se lanzo). Este identificador se guardara en */
  /* el heap junto con la prioridad. */

  theap *cola;      /* La cola de prioridad. */

  int proceso_ejec;   /* Identificador del proceso en ejecucion */
  int t_proceso_ejec; /* Tiempo del proceso en ejecucion */


  clrscr();
  printf("Cual es el fichero con la lista de procesos? ");
  gets(fichero_procesos);

  clrscr();
  /* Representamos la secuencia de procesos en memoria */
  proceso_ejec = -1;
  lee_fichero_proc(fichero_procesos, vprocesos);
```

```
  /* creamos la cola de prioridad vacia */
  cola = crea_heap();

  /* Imprimimos cabecera de la ejecucion de procesos */
  print_head();

  /* Simulamos un bucle de tiempo donde van viniendo los */
  /* procesos y se van ejecutando */
  for (i=0;i<1000;i++) {
     /* Si llega nuevo proceso */
     /*    --> encolar y generar mensaje de suceso */
     if (vprocesos[i].tiempo!=-1) {
        insertar(cola, vprocesos[i].prioridad, i);
        crea_suceso(vprocesos[i].nombre,
                     vprocesos[i].prioridad,vprocesos[i].tiempo);
     }
     /* Si no hay proceso ejecutandose */
     /*    --> extraer de la cola el mas prioritario */
     if ((proceso_ejec == -1) && (vacio(cola)==0)) {
         proceso_ejec = extrae_min(cola);
         t_proceso_ejec = 1;
     }
     /* ejecutar una unidad de tiempo con el proceso actual */
     if (proceso_ejec != -1)
           print_pantalla(i, vprocesos[proceso_ejec].nombre);
     else print_pantalla(i, "");
     /* Actualizar proceso_ejec y t_proceso_ejec */
     if (vprocesos[proceso_ejec].tiempo <= t_proceso_ejec) {
        proceso_ejec = -1;
     }
     else t_proceso_ejec++;
     sleep_1();
  }
  return(0);
}
```

Práctica 11

Aciclidad de Grafos

11.1. Introducción

En esta práctica se pretende ampliar el estudio que sobre la estructura de datos Grafo se ha realizado en clase de teoría. Los objetivos del trabajo son:

- Conocer detalles de la implementación de grafos en lenguaje C.
- Conocer e implementar el algoritmo que comprueba la existencia de ciclos en un grafo.

En la sección 11.2 se recordará la estructura de datos grafo y qué son los ciclos. En la sección 11.3 veremos cómo se puede almacenar un grafo en un fichero para poder usarlo en nuestros programas, y por último, en la sección 11.4 veremos el algoritmo que comprueba la aciclidad en un grafo, que será el algoritmo que el alumno tendrá que implementar.

Todos los códigos fuente mostrados en las secciones finales para las diferentes partes de la práctica serán proporcionados al alumno.

11.2. Grafos acíclicos

En clase de teoría se vió que los grafos son modelos que permiten representar relaciones entre elementos de un conjunto. Formalmente, un **grafo** es un par (V, E) donde V es un conjunto de elementos denominados **vértices o nodos**, sobre los que se ha definido una relación, y E es un conjunto de pares (u, v), $u, v \in V$, denominados **aristas o arcos**, donde (u, v) indica que u está relacionado con v de acuerdo a la relación definida sobre V. Durante el resto de la práctica supondremos que estamos trabajando con grafos dirigidos.

También se definió un **camino** desde un vértice $u \in V$ a un vértice $v \in V$ como una secuencia de vértices $v_1, v_2, \ldots, v_k$ tal que $u = v_1$, $v = v_k$, y $(v_{i-1}, v_i) \in E$, para $i = 2, \ldots, k$.

Definimos también un **camino simple** como un camino en que todos sus vértices, excepto, tal vez, el primero y el último, son distintos. Por último decimos que un **ciclo** es un camino simple $v_1, v_2, \ldots, v_k$ tal que $v_1 = v_k$.

A un grafo sin ciclos se le conoce como **grafo acíclico**. En las figuras 11.1 y 11.3 podemos ver ejemplos de grafos con ciclos y en la figura 11.2 podemos ver un ejemplo de grafo sin ciclos (acíclico).

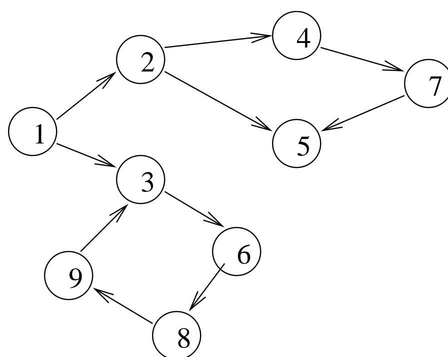


Figura 11.1: Ejemplo de grafo con un ciclo formado por el camino $< 3, 6, 8, 9, 3 >$.

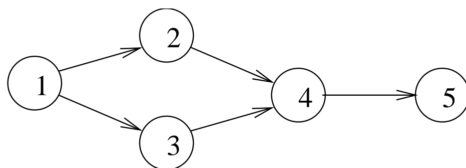


Figura 11.2: Ejemplo de grafo acíclico.

11.3. Representación de grafos

Utilizaremos la representación para grafos propuesta en clase mediante matrices de adyacencia. Para ello, supondremos que los vértices de cualquier grafo que definamos se identificarán con números del 1 al número de vértices que haya correlativamente. Así, a la hora de construir la matriz de adyacencia, tendremos que `A[u][v]` valdrá 0 si no existe ninguna arista que vaya de `u` a

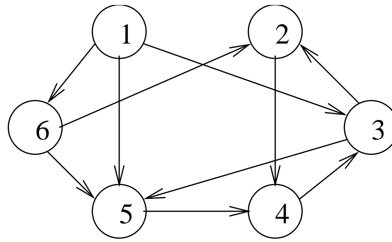


Figura 11.3: Otro ejemplo de grafo con ciclos. Por ejemplo $< 2, 4, 3, 2 >$ o $< 3, 5, 4, 3 >$

`v`, y `A[u][v]` valdrá 1 si existe una arista $(u, v) \in E$ (notar que trabajamos con grafos dirigidos y la matriz no será simétrica).

El tipo de datos `tgrafo` que utilizaremos en lenguaje C se define de esta forma:

```
#define MAXVERT 100

typedef struct {
    int talla;    /* Numero de vertices. */
    int A[MAXVERT][MAXVERT];
                  /* Matriz de adyacencia. */
} tgrafo;
```

que se puede observar también en el fichero `grafo.h` mostrado en la sección 11.5.

Para poder introducir un grafo de manera cómoda en nuestros programas y poder especificar su conjunto de vértices así como su conjunto de aristas, vamos a realizarlo mediante una definición del grafo en un fichero que se leerá de manera automática mediante una función en nuestros programas. Con esto permitiremos que probar nuestro algoritmo sobre nuevos grafos sea tan fácil como cambiar el fichero de entrada.

El formato de fichero que vamos a utilizar para definir grafos es el siguiente:

- Todas las líneas del fichero cuyo primer caracter sea una `#` serán consideradas como comentarios.

- Las líneas que definen el grafo tienen dos campos por línea:

  - Si el primer campo es la palabra reservada `Numvert`, entonces el segundo campo es numérico e indica el número de vértices del grafo (se puede entender como el conjunto V).

- Si el primer campo es un número, entonces el segundo campo es también numérico y los dos campos se refieren a los vértices que componen la arista que va del vértice del primer campo al del segundo campo.

Por ejemplo, en el fichero `ej_graf` que se proporciona junto con los códigos fuente puede verse la definición para el grafo de la figura 11.1, que se muestra a continuación:

```
Numvert 9
1 2
1 3
2 4
2 5
3 6
4 7
7 5
6 8
8 9
9 3
```

Otro ejemplo de definición de grafos en un fichero puede ser el siguiente, que define el grafo de la figura 11.3:

```
Numvert 6
1 3
1 5
1 6
2 4
3 2
3 5
4 3
5 4
6 2
6 5
```

11.4. El algoritmo *acíclico*

Nuestro principal objetivo ahora va a consistir en detectar si un grafo tiene ciclos o no. Una posible aplicación de este algoritmo sería detectar si

se producen ciclos de llamadas en las lineas telefónicas. Por ejemplo, supongamos que existen dos despachos A y B, cada uno con una extensión telefónica diferente, el teléfono A y el teléfono B. Supongamos que el trabajador del despacho A va a reunirse con el otro trabajador en el despacho B, y, para estar localizable, redirige el telefóno A hacia el teléfono B, haciendo uso del servicio de desvío de llamada ofertado por la operadora de telefonía. Supongamos ahora que mientras el trabajador A está de camino hacia el despacho B, el trabajador B decide llamar al trabajador A. Lo que ocurriría es que el telefono A redirigiría la llamada al teléfono B, que es el mismo que le está llamando, se produciría un ciclo que las redes telefónicas suelen solventar indicando que el teléfono destino está descolgado (comunicando). Con un algoritmo que compruebe la aciclidad de un grafo podría haberse detectado esta situación y tratarla de una manera más adecuada.

Supongamos que tenemos un grafo dirigido $G = (V, E)$, para determinar si G es acíclico, esto es, si G no contiene ciclos, puede utilizarse el recorrido en profundidad. La idea es que si durante el recorrido en profundidad se encuentra una arista de retroceso (que vuelve a un vértice ya visitado en el camino actual) entonces el grafo G tiene un ciclo.

Para ver este hecho, supongamos que G es cíclico. Si se efectúa un recorrido en profundidad de G, habrá un vertice v que en algún camino establecido desde el vértice origen aparecerá dos veces, con lo que se habrá formado un ciclo. Como se puede observar en la figura 11.4.

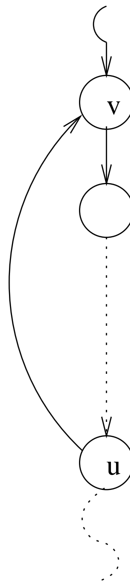


Figura 11.4: Todo ciclo contiene una arista de retroceso.

Para poder detectar un ciclo usando el recorrido en profundidad será necesario especificar para cada nodo su *nivel de profundidad* cuando vayamos recorriendo un determinado camino, y borrarlo cuando deshagamos el camino. Vamos a ver esto con más detalle.

Es sencillo ver que el algoritmo de recorrido en profundidad por sí solo (visto en clase de teoría) no es suficiente para detectar ciclos, puesto que si bien dicho algoritmo iba coloreando los nodos ya visitados, en nuestro problema, el llegar durante el recorrido a un nodo ya coloreado no implica necesariamente que se haya formado un ciclo, sino que puede indicar que a ese nodo se puede llegar por distintos caminos. Esta última situación es la que se muestra en la figura 11.5.

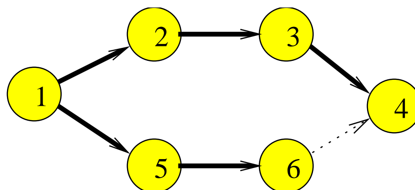


Figura 11.5: Cuando vamos a visitar el nodo 4 desde el nodo 6, se detecta que 4 ya está coloreado, pero ello no necesariamente implica que exista un ciclo.

Para solucionar esto, vamos a introducir un nuevo vector llamado `profundidad` en el algoritmo de recorrido en profundidad y que almacenará en la posición correspondiente a cada nodo un número que indicará la *profundidad* en el camino desde el vértice desde el cual se comenzó el recorrido. Como profundidad entenderemos la distancia al vértice de comienzo del recorrido (más concretamente al vértice de inicio de la componente fuertemente conexa a la que estamos aplicando el recorrido en profundidad en ese momento). Por ejemplo, en el grafo de la figura 11.5, si suponemos que comenzamos el recorrido desde el vértice 1, entonces el vértice 1 obtendrá una profundidad 1, cuando se visite el nodo 2, éste obtendrá una profundidad 2 y cuando se visite el nodo 3, éste obtendrá una profundidad 3, etc. Cuando se haya llegado al nodo 4 por ese camino y deshagamos el camino hasta el nodo 1, debe actualizarse la profundidad de los nodos 4, 3 y 2 a un valor 0.

Para poder asignar la profundidad correspondiente a un nodo cuando vayamos a visitarlo en el recorrido y para poder *borrar* esa profundidad al volver atrás en el camino, debemos aprovechar el esquema recursivo de la función `visita_nodo()`. A partir de estas guías, el alumno tendrá que modificar el algoritmo `visita_nodo()` visto en clase para que realice la asignación correspondiente del valor de profundidad.

Aclaramos ahora cómo vamos a detectar ciclos. Supongamos que un grafo G es cíclico. Si se realiza el recorrido en profundidad de G, coloreando los nodos visitados y manteniendo adecuadamente el vector `profundidad` como se ha explicado en el párrafo anterior, entonces, por ser G cíclico habrá un vértice v que tenga el nivel de profundidad menor en un ciclo. Para ver esto más claramente, considérese una arista (u, v) en algún ciclo que contenga a v (ver figura 11.4). Ya que u está en el ciclo, debe ser un *descendiente* de v en el recorrido en profundidad, pero entonces ocurrirá que cuando intentemos visitar v desde u, nos encontraremos con que v ya está coloreado (visitado) y además tiene un nivel de profundidad menor que el de u y diferente de 0, con lo que necesariamente, la arista (u, v) es una arista de retroceso y por tanto existe un ciclo.

La detección de ciclos también se debe realizar en la función `visita_nodo()`. Para saber que un grafo es cíclico, usaremos una variable que se incremente cuando haya detectado un ciclo, esta variable se inicializará a 0 antes de realizar el recorrido en profundidad del grafo, y, si al acabar éste, su valor es superior a 0, entonces es que el grafo tiene al menos un ciclo y es cíclico.

La ejecución de nuestro programa para el grafo de la figura 11.1 debe devolver *algo parecido* a:

```
Cual es el fichero del grafo? ej_graf

Recorriendo grafo: Vertice inicio = 1
Paso de 1 a 2
Paso de 2 a 4
Paso de 4 a 7
Paso de 7 a 5
Paso de 1 a 3
Paso de 3 a 6
Paso de 6 a 8
Paso de 8 a 9
Ciclo desde 9 a 3

EL GRAFO ES CICLICO

Pulsa Intro para acabar
```

No es necesario que el alumno imprima todas las líneas que muestran el recorrido del grafo y detección de ciclos como se ha propuesto en el ejemplo.

Pero es fundamental que el alumno sea capaz de definir el algoritmo de aciclidad de un grafo.
Ejercicio: Incluir el código necesario en las funciones `visita_nodo()` y `aciclico()` del módulo `grafo.c` para que el algoritmo pueda funcionar correctamente y testear la aciclidad de los grafos. Ejecutar el programa `acyclic.c` (tendrás que crear un proyecto en Turbo C) y ver si se cumple la aciclidad en los diferentes grafos ejemplo que se proporcionan.

11.5. Librería para Grafos

A continuación se presentan los módulos necesarios para implementar el tipo abstracto de datos **Grafo** en una librería.

Código de grafo.h:

```
/* MODULO PARA LA DEFINICION DE GRAFOS DIRIGIDOS */
/* CON MATRICES DE ADYACENCIA */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*********************************************/
/****** Definicion de constantes: ******/

#define MAXVERT 100
#define BLANCO 0   /* Para colorear los vertices. */
#define AMARILLO 1

/*********************************************/
/****** Definicion de tipos: ******/

/* Definicion del tipo tgrafo. */
typedef struct {
   int talla;   /* Numero de vertices. */
   int A[MAXVERT][MAXVERT];
                /* Matriz de adyacencia. */
} tgrafo;

/*********************************************/
/* Definicion de funciones: */

/* Lee un grafo de un fichero y lo carga en */
/* una estructura tgrafo en memoria. */
tgrafo *lee_grafo(char *graph_file_name);

/* Devuelve cierto (1) si el grafo es aciclico, */
/* falso (0) si el grafo tiene algun ciclo. */
int aciclico(tgrafo *g);
```

Código de grafo.c:

```
/* MODULO PARA LA DEFINICION DE GRAFOS DIRIGIDOS */
/* CON MATRICES DE ADYACENCIA */
#include "grafo.h"

/*******************************************************/
/****************** FUNCIONES LOCALES ******************/
/*******************************************************/

/*******************************************************/
/* Funcion de ayuda a aciclico() para realizar el recorrido */
/* de un grafo comprobando si tiene ciclos. */

void visita_nodo(int u, tgrafo *g, int *color,
                               int *num_ciclos, int *profundidad){
  int v;   /* Para recorrer vertices. */

  /* INSERTAR CODIGO DE VISITA NODO. */
  /* ES MUY SIMILAR AL VISTO EN CLASE PERO */
  /* DETECTANDO LOS POSIBLES CICLOS QUE SE FORMEN */
  /* ..... */
  /* ..... */
  /* ..... */
}
```

```
/**************************************************************/
/****************** FUNCIONES EXPORTABLES ******************/

/**************************************************************/
/* Lee un grafo de un fichero y lo carga en */
/* una estructura tgrafo en memoria. */

tgrafo *lee_grafo(char *graph_file_name){
  tgrafo *g;   /* Para crear el grafo. */
  FILE *f;     /* Para abrir el fichero. */
  int u,v;     /* Para vertices. */
  char l[100]; /* Para leer de fichero. */
  char sim1[50], sim2[50];

  /* Creamos el grafo y lo inicializamos */
  if ((g = (tgrafo *) malloc(sizeof(tgrafo))) == NULL) {
     fprintf(stderr,"ERROR: no se puede reservar memoria para el grafo\n");
     exit(1);
  }
  g->talla = 0;
  for (u=0;u<MAXVERT;u++) for (v=0;v<MAXVERT;v++) g->A[u][v] = 0;

  /* Leemos el grafo del fichero. */
  if ((f = fopen(graph_file_name,"r")) == NULL) {
     fprintf(stderr,"ERROR: no se puede abrir el fichero %s\n",graph_file_name)
     exit(1);
  }
  while (fgets(l,100,f)!=NULL) {
    l[strlen(l)-1]='\0';
    if (l[0] != '#') {
        sscanf(l,"%s %s",sim1,sim2);
        if (strcmp(sim1,"Numvert") == 0)
            /* Hemos leido el numero de vertices del grafo. */
            g->talla = atoi(sim2);
        else /* Hemos leido una arista del grafo. */
            g->A[atoi(sim1)][atoi(sim2)] = 1;
    }
  }
  fclose(f);
  return(g);
}
```

```
/*******************************************************/
/* Devuelve cierto (1) si el grafo es aciclico, */
/* falso (0) si el grafo tiene algun ciclo. */

int aciclico(tgrafo *g){
  int u;   /* Para recorrer vertices. */
  int color[MAXVERT];
             /* Para colorear los vertices. */
  int profundidad[MAXVERT];
             /* Para mantener la profundidad de los vertices. */
  int num_ciclos;
             /* Para contar posibles ciclos. */

  /* INSERTAR CODIGO DE LA FUNCION ACICLICO. */
  /* ES MUY SIMILAR AL RECORRIDO EN PROFUNDIDAD */
  /* VISTO EN CLASE. */
  /* ..... */
  /* ..... */
  /* ..... */
}
```

11.6. Código fuente del programa principal

Código de acyclic.c:

```
/* PROGRAMA PRINCIPAL PARA TESTEAR LA ACICLIDAD EN GRAFOS */
/* DIRIGIDOS NO PONDERADOS */
#include <stdio.h>
#include <conio.h>
#include "grafo.h"

/**************************************************************/
/*************** PROGRAMA PRINCIPAL ****************************/
/**************************************************************/
int main() {
  char aux[3];
  char fichero_grafo[100];
  tgrafo *g;     /* El grafo. */

  clrscr();

  printf("Cual es el fichero del grafo? ");
  gets(fichero_grafo);

  /* Leemos el grafo del fichero. */
  g = lee_grafo(fichero_grafo);

  /* Comprobamos si el grafo tiene ciclos. */
  if  (aciclico(g)) printf("\n\nEL GRAFO ES ACICLICO\n\n");
  else  printf("\n\nEL GRAFO ES CICLICO\n\n");

  printf("Pulsa Intro para acabar\n");
  gets(aux);

  return(0);
}
```

Bibliografía

[Aho88] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *"Estructuras de datos y algoritmos"*, Addison-Wesley, 1988.

[Aho86] A. V. Aho, R. Sethi, J. D. Ullman, *"Compiler design: principles, techniques and tools"*, Addison-Wesley, 1986.

[Cormen90] T. Cormen, Ch. Leiserson, R. Rivest, *"Introduction to Algorithms"*, MIT, 1990.

[Gonnet91] G.H. Gonnet, R. Baeza-Yates, *"Handbook of algorithms and data structures in Pascal and C"*, Addison-Wesley, 2nd ed., 1991.

[Hoare62] C. A. R. Hoare, *"Quicksort"*, Computer Journal, Vol. 5(4), 1962.

[Sedgewick78] R. Sedgewick, *"Implementing Quicksort Programs"*, Communications of the ACM, Vol. 21(10), 1978.

[Weiss95] M. A. Weiss, *"Estructuras de datos y Algoritmos"*, Addison-Wesley Iberoamericana, 1995.

[Weiss00] M. A. Weiss, *"Estructuras de datos en Java"*, Addison-Wesley, 2000.