Algoritmos y Estructuras de Datos
2º de Ingeniería Informática, Curso 2008/2009

SEMINARIO "C para programadores java"
(este seminario forma parte de las actividades del *proyecto piloto*)

Sesión 4

Contenidos:

1. Manejo de ficheros.
2. Otras funciones de interés
3. El preprocesador de C
4. Programación modular
5. El programa make

---

1. Manejo de ficheros

- Los tipos y funciones necesarios para el manejo de ficheros están definidos en la librería **`stdio.h`**.

- La librería **`stdio.h`** define el tipo **`FILE`** (ojo, en mayúsculas). Para usar ficheros debemos usar punteros al tipo `FILE`.

```
#include <stdio.h>
FILE *f, *g;
```

- **Operaciones con ficheros:**
  - Abrir un fichero: `fopen`
  - Leer/escribir en un fichero:
    - modo texto: `fprintf, fscanf, getc, putc, fgets, fputs`
    - modo bineario/crudo: `fread, fwrite`
  - Moverse a un punto: `fseek`
  - Comprobar si se ha llegado al final: `feof`
  - Cerrar un fichero: `fclose`
  - NO EXISTE la serialización.

- **Abrir un fichero**: **`fopen`**
  `FILE * fopen ( char *nombre , char *modo )`
  - **Valor devuelto:** devuelve un puntero al fichero abierto. Si no se puede abrir devuelve `NULL`.
  - **Modo de acceso:** de tipo cadena

	Fichero de texto	Fichero binario
Abrir sólo lectura	`"r"`	`"rb"`
Crear y abrir para escritura	`"w"`	`"wb"`
Abrir para añadir al final	`"a"`	`"ab"`
Abrir para lectura/escritura	`"r+"`	`"r+b"`
Crear y abrir para lectura/escritura	`"w+"`	`"w+b"`
Abrir o crear para añadir	`"a+"`	`"a+b"`

- **Cerrar un fichero**: **`fclose`**
  `int fclose ( FILE *fichero )`

- **Leer/escribir en un fichero.**
- **`fprintf`**, **`fscanf`**. Igual que `printf` y `scanf` pero en un fichero, que se pasa como primer parámetro.

```
FILE *f;
int n;
f= fopen("prueba", "r+");
if (!f) {printf("Error. No puedo..."); return;}
fprintf(f, "Hola número %d", n);
fscanf(f, "%d", &n);
```

- **`fgetc(FILE *f)`**, **`fputc(int c, FILE *f)`**. Leer o escribir un carácter en un fichero. Devuelven la constante **`EOF`** si hay un error.

```
char c;
while ((c= fgetc(f))!= EOF)
printf("Leído: %c\n", c);
```

- **`fgets`**, **`fputs`**. Leer o escribir una línea de texto de un fichero. Recordar que `scanf("%s", ...)` lee una cadena pero acabando en el primer espacio en blanco (tabulador, intro o espacio).

`int fputs (char *cadena, FILE *fichero)`
Escribe la cadena `cadena` hasta el final de la misma (carácter `'\0'`). Devuelve `EOF` si ha ocurrido un error.

`char *fgets (char *cadena, int max, FILE *fichero)`
Lee la siguiente línea del fichero en `cadena`, acabando en un final de línea (carácter `'\n'`). Devuelve `NULL` si ha ocurrido un error.

- **`fread`**, **`fwrite`**. Leer o escribir en un fichero en **modo binario**.

`unsigned fread(void *ptr, unsigned size, unsigned n, FILE *fichero)`
Lee de fichero bloque de `size*n` bytes y los almacena a partir de posición apuntada por `ptr`. `size` es tamaño de tipo de datos a leer y `n` número de datos. Devuelve el número de datos leídos (si no hay error, debería ser igual a `n`).

`unsigned fwrite(void *ptr, unsigned size, unsigned n, FILE *fichero)`
Escribe en fichero bloque de `size*n` bytes, tomados a partir de posición apuntada por `ptr`. `size` es el tamaño del tipo de datos a escribir y `n` el número de datos. Devuelve número de datos escritos (si no hay error, igual a `n`).

- **`fseek`**. Moverse a un punto concreto de un fichero.

`int fseek(FILE *fichero, long offset, int modo)`

Permite moverse a una posición cualquiera dentro de un fichero. La posición nueva está dada por `offset` (desplazamiento, en bytes) a partir de la posición dada por `modo`: `SEEK_SET` (=0) desde el comienzo del fichero; `SEEK_CUR` (=1) desde la posición actual; `SEEK_END` (=2) desde el final del fichero.
Función de consulta (dice la posición actual en el fichero):

`long int` **`ftell`** `(FILE *fichero).`

- **`feof`**. Comprobar si se ha llegado al final de un fichero.

`int feof(FILE *fichero)`

Devuelve **true** (valor distinto de 0) si hemos llegado al final del fichero o **false** (valor 0) en caso contrario.

- **Ejemplo.**

```
#include <stdio.h>
#include <stdlib.h>    /* Funciones rand(), srand() */

int main() {
   FILE *e, *s;
   int n, m, total= 0;
   if (!(e= fopen("ejemplo.txt", "w"))) {
      printf("Error. No se puede crear ejemplo.txt\n");
      return 1;
   }
   srand(10);
   fprintf(e, "%d %d\n", (int) (rand()*10.0/RAND_MAX),
                         (int) (rand()*10.0/RAND_MAX));
   if (fclose(e)==EOF) {
      printf("Error. No se puede cerrar ejemplo.txt\n");
      return 2;
   }
   if (!(s= fopen("ejemplo.txt", "r"))) {
      printf("Error. No se puede abrir ejemplo.txt\n");
      return 3;
   }
   while (!feof(s)) {
      total++;
      if (fscanf(s, "%d %d\n", &n, &m)==2)
         printf("R. Murcia: %d - At. Madrid: %d\n", n, m);
      else
         printf("El fichero no tiene el formato esperado\n");
   }
   printf("Realizados %d pronósticos.\n", total);
   if (fclose(s)==EOF) {
      printf("Error. No se cerrar el fichero ejemplo.txt\n");
      return 4;
   }
   return 0;
}
```

- Probar distintas posibilidades en la línea señalada por: ➡ , en lugar de `"w"` poner `"a"`, `"r"`, `"r+"`, `"w+"`, ... ¿Cuál es el resultado?

2. Otras funciones de interés

NOTA: referencia completa de funciones librería en la página web (libc.pdf).

- **Con cadenas**: `string.h  stdlib.h`

Nombre	Sintaxis	Explicación
strlen	`unsigned strlen(char *s)`	Devuelve la longitud de la cadena.
strcpy	`char *strcpy(char *des, char *src)`	Copia la cadena `src` en `des` y devuelve `des`.
strcat	`char *strcat(char *des, char *src)`	Concatena a `des` la cadena `src` y devuelve `des`.
strcmp	`int strcmp(char *s1, char *s2)`	Compara las dos cadenas. El resultado es: 0 si son iguales; <0 si `s1<s2` ; y >0 si `s1>s2`.
atoi	`int atoi(char *s)`	Convierte una cadena a un entero. Otras funciones relacionadas: `atol, atof, itoa, itol, ...`

Ejemplo de uso:

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

void imprime_resultado_strcmp(int resultado_strcmp) {
   if(!resultado_strcmp)  {printf("cadenas iguales\n"); return;}
   if(resultado_strcmp>0) {printf("cadena 1 mayor\n"); return;}
   printf("cadena 1 menor \n");
}

main(){
   int longitud = 20;
   char cadenaA[longitud]="Soy la cadena A";
   char cadenaB[longitud]="Soy la cadena B";

   imprime_resultado_strcmp(strcmp(cadenaA,cadenaB);
   imprime_resultado_strcmp(strcmp(cadenaB,cadenaA);

   strcpy(cadenaA,cadenaB);
   printf("%s; %s\n",cadenaA,cadenaB);

   imprime_resultado_strcmp(strcmp(cadenaA,cadenaB);
}
```

- **Con memoria**: `mem.h  string.h`

Nombre	Sintaxis	Explicación
memcpy	`void *memcpy(void *dest, void *src , unsigned n)`	Copia `n` bytes desde la posición apuntada por `src` hasta la apuntada por `dest`. Ver también `memmove`.
memcmp	`void *memcmp(void *s1, void *s2 , unsigned n)`	Compara `n` bytes de `s1` y de `s2`. El resultado es como en `strcmp`.
memset	`void *memset(void *s, int c, unsigned n)`	Escribe en los `n` bytes de `s` el valor (byte) `c`.

- **Con caracteres**: `ctype.h`

Nombre	Sintaxis	Explicación
isalpha	`int isalpha(int c)`	Devuelve **true** si el carácter `c` es una letra.
isupper	`int isupper(int c)`	Devuelve **true** si el carácter `c` es una letra mayúscula.
islower	`int isulower(int c)`	Devuelve **true** si el carácter `c` es una letra minúscula.
isspace	`int isspace(int c)`	Devuelve **true** si el `c` es un espacio, tabulador o '\n'.
toupper	`int toupper(int c)`	Devuelve el carácter `c` en mayúsculas.
tolower	`int tolower(int c)`	Devuelve el carácter `c` en minúsculas.

- **Otras**: `stdlib.h`

Nombre	Sintaxis	Explicación
system	`int system(char *command)`	Ejecuta un comando del sistema operativo. Devuelve 0 si no hay error.
exit	`void exit(int status)`	Cierra todos los ficheros y termina la ejecución del programa, devolviendo `status`.

3. El preprocesador de C

- El **preprocesador:**
  - es parte del compilador, procesa los ficheros **.c** antes de compilar.
  - admite varios comandos, que empiezan por **#**.
- **Comando `#define`:**
  - Permite definir constantes, con o sin parámetros.
  - Apariciones de la constante son sustituidas textualmente por su valor
  - Estas constantes se llaman también **macros**.
  - **Macro sin parámetros**.
    - Normalmente se ponen en mayúsculas, para diferenciarlas de las variables.

```
#define NOMBRE   VALOR
```

```
#define PI 3.1415926
#define MAX_INT  ((unsigned)(~0)>>1)
#define MENSAJE "Mensaje predefinido"
```

  - **Macro con parámetros**.
    - Es parecido a una función, pero sigue siendo una constante.
    - Los parámetros no tienen tipo.
    - La expresión se sustituye textualmente en tiempo de compilación.

```
#define NOMBRE(P1,P2,...,Pk) EXPRESION
```

```
#define CUADRADO(N)  (N)*(N)
#define MAX(A,B)  ((A)>(B)?(A):(B))
...
int i, j, k;
k= CUADRADO(i+1);
j= MAX(E*PI, k);
...
```

    - Es recomendable usar paréntesis para evitar efectos indeseados. ¿Qué pasaría si pusiéramos: `#define CUADRADO(N) N*N` ?

- **Comando `#include`:**
  - Es como el *import* de java.
  - Incluye el contenido de un fichero en el punto donde aparece.
  - Normalmente será un fichero de cabecera de una librería (extensión **.h**), pero puede ser cualquier fichero.

`#include <NOMBRE_FICHERO>` → Para librerías de sistema

`#include "NOMBRE_FICHERO"` → Para librerías y ficheros propios (busca primero en directorio actual)

  - **`stdio.h`** → Librería estándar de entrada/salida
  - **`stdlib.h`** → Librería de funciones y tipos estándar
  - **`time.h`** → Funciones y tipos para manejo del tiempo
  - **`math.h`** → Funciones matemáticas
  - **`mem.h`** → Manejo de memoria
  - `...`

- **Comandos de compilación condicional:**
  - Permiten añadir o quitar trozos de código, en tiempo de compilación, según cierta condición.
  - La condición es del tipo "macro definido" o "macro no definido".

```
#define DEBUG
      ...
#ifdef DEBUG
      printf("Pasa por aquí.");
#endif
      ...

      ...
#ifndef OPTIMIZAR
      i= i + 1;
      j= j*(i+1);
      i= j;
#else
      i= j*= (++i + 1);
#endif
```

- **Indefinir una macro.** **`#undef`** `NOMBRE`

- **Ejemplo de uso.** Evitar que se incluya un fichero varias veces.

```
#ifndef _LIBRERIA_PILAS
#define _LIBRERIA_PILAS
#include "pilas.h"
.....
#endif
```

- **Ejemplo.** Probar compilando con
  - "`gcc fichero.c`"
  - "`gcc -DDEBUG fichero.c`" (significado de opción –D, usar `man`)

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

```
#define MAX_INT  ((unsigned)(~0)>>1)
#define MENSAJE "Mensaje predefinido"
#define MAX(A,B)  ((A)>(B)?(A):(B))

int main (void) {
  int i, j;
#ifdef DEBUG
  printf("Maximo entero: %d\n", MAX_INT);
  printf("%s\n", MENSAJE);
#endif
  printf("Introduce dos enteros:");
  scanf("%d %d", &i, &j);
  printf("Maximo: %d\n", MAX(i, j));
  return 0;
}
```

4. Programación modular

- La **programación modular** permite descomponer la complejidad de una aplicación en distintos trozos. Un **módulo** o **paquete** contiene un conjunto de funcionalidades relacionadas.

- **Ejemplo.** Aplicación de edición de textos. El código no va en un solo fichero, sino que se descompone en módulos:
  - **Tipos**. Incluye la implementación de los tipos de datos básicos necesarios en el programa (pilas, listas, diccionarios, etc.).
  - **Interface**. Funciones del interface de usuario (entrada y salida).
  - **Ficheros**. Funciones para el manejo de los documentos (lectura y escritura).
  - **Diccionario**. Código relacionado con el corrector ortográfico.
  - **Principal**. Contiene la estructura final de la aplicación, haciendo uso de todos los demás módulos.

- En C no existe, de forma explícita, el concepto de módulo o paquete (al contrario que en java, donde sí existe, *package*). Un programa C puede estar compuesto por varios **ficheros**. En uno de ellos tiene que estar definida la función `main`.

- **Programación modular en C.** Para cada módulo hay dos ficheros (en java 1):

  - **Fichero de cabecera (header)**. Extensión **.h**. Contiene:
    - la definición de los tipos de datos y
    - la declaración de las funciones (cabecera).

  - **Fichero de implementación.** Extensión **.c**. Contiene:
- la implementación de las funciones declaradas en el fichero de cabecera (y quizás de otras auxiliares de éstas).

- **Ejemplo.** Módulo Tipos, del editor de textos.

tipos.h

```
#ifndef _LIB_TIPOS
#define _LIB_TIPOS

#define MAX_CAPACIDAD 100

extern int estado_error;

typedef struct {
   int tope;
   int datos[MAX_CAPACIDAD];
} pila;

void push (pila *, int);
void pop (pila *);
int top (pila *);

...

#endif
```

?

tipos.c

```
#include "tipos.h"

int estado_error= 0;

void push (pila *p, int v)
{
  if (p->tope>=MAX_CAPACIDAD) {
    estado_error= 1;
    return;
  }
  p->datos[p->tope++]= v;
}

void pop (pila *p)
{
  p->tope--;
  estado_error|= p->tope<0;
}
...
```

principal.c

```
/* Módulo principal */

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

#include "tipos.h"
#include "interface.h"

int main (int na, char *va[])
{
  pila *p1;
  ...
}
```

- **Consideraciones de interés.**

1. Todas las macros y tipos de datos públicos (los que queremos que utilicen los usuarios) deben definirse en el fichero de cabecera del módulo.
2. Las funciones públicas también aparecerán en el fichero cabecera, pero sólo debe aparecer la declaración, es decir, la cabecera. La implementación de las funciones viene en el fichero de implementación.
3. En el fichero de implementación se hará normalmente un `#include` del fichero de cabecera del mismo módulo (para usar las macros y tipos definidos).
4. En el caso de las variables públicas, la variable debe estar definida (y, en su caso, inicializada) en el fichero de implementación. En el fichero de cabecera se pondrá: `extern tipo nombre;` (variable *externa*[1]).
5. Es aconsejable documentar bien ambos ficheros:
   - Documentación del fichero cabecera: de cara al usuario.
   - Documentación del fichero de implementación: cara a posibles modificaciones posteriores.
6. Si un módulo usa otro módulo, hará un `#include` del fichero cabecera correspondiente. Este `#include` puede estar en el fichero cabecera del primero.
7. Conviene usar `#ifndef ... #endif` en el fichero cabecera, para evitar que un fichero se incluya varias veces.

- **Compilación de la aplicación.** Supongamos el editor de textos. Al compilar se deben indicar todos los ficheros de implementación. No es necesario incluir los ficheros cabecera.

```
>> gcc tipos.c interface.c ficheros.c principal.c -o myword
```

[1] Una varible *externa* es una variable que está fuera de toda función, es decir, una variable global. Una función "ve" cualquier variable global **definida** en el mismo fichero y antes que ella. Para hacer visibles variables globales que no lo son, añadir **declaración** con palabra `extern`, en mismo fichero y antes.

5. El programa make

(breves apuntes disponibles en la página web, make.pdf)

- La generación de un programa implica **dos pasos**:
  - **Compilación:** Dado un fichero de **código fuente** (**.c**) generar un fichero de **código objeto** (**.o**).
  - **Linkado o enlace:** Dado uno o varios ficheros de **código objeto** (**.o**) producir un **fichero ejecutable**.

- Por defecto, **gcc** hace los dos pasos sin generar el fichero de código objeto.

- Es posible hacer que genere el fichero **.o**. **Opción -c**: compilar sólo, sin enlazar.

```
>> gcc -c tipos.c          → Genera tipos.o
```

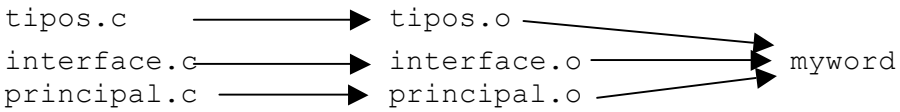
- Al compilar y enlazar un programa se pueden utilizar, y mezclar, ficheros fuente y objeto.

```
>> gcc tipos.o interface.c ficheros.o principal.c -o myword
```

- **Compilación en programas de gran tamaño:** muchos módulos, código fuente largo, mucho tiempo de compilación, modificaciones frecuentes, ...

- **Solución:** generar el fichero objeto de cada módulo por separado. Cuando se cambia un módulo, recompilar ese módulo y linkar el programa principal.

```
>> gcc -c tipos.c
>> gcc -c interface.c
>> gcc -c ficheros.c
>> gcc -c principal.c
>> gcc tipos.o interface.o ficheros.o principal.o -o myword
```



- **Problema:** si se modifica un fichero cabecera (.h) o fuente (.c) otros módulos se pueden ver afectados y hace falta recompilarlos. ¿Cuáles?

- **Programa make**.
  - Ayuda a automatizar el proceso de compilación de un programa con muchos módulos.
  - Busca y procesa un fichero llamado `makefile`, que indica las dependencias entre los módulos.
  - Estructura del fichero `makefile`:

```
OBJETIVO: FICHERO1 FICHERO2  ... FICHEROP
<Tabulador>COMANDO
```

- **Significado:**
  - El fichero `OBJETIVO` debe recompilarse cuando se modifiquen (o deban ser recompilados) `FICHERO1` o `FICHERO2`, ..., o `FICHEROP` (estos se denominan dependencias o prerrequisitos).
  - La forma de obtener este fichero `OBJETIVO` es ejecutando `COMANDO`.

```
myword: tipos.o interface.o ficheros.o principal.o myword.h
<Tabulador>gcc tipos.o interface.o ficheros.o principal.o -o myword

tipos.o: tipos.c tipos.h
<Tabulador>gcc -c tipos.c

interface.o: tipos.o interface.c interface.h
<Tabulador>gcc -c tipos.o interface.c

...
```

OJO: para generar los .o, incluir el .c y todos los .h que incluya el .c.

- Al ejecutar "`make`", éste busca el fichero `makefile`.
  - De todos los objetivos, intenta generar el primero que aparezca.
  - Si necesita otros objetivos, los procesa también, se sigue la cadena de dependencias hasta llegar a dependencias fichero.

☞ • **Ejemplo.** Editar los siguientes ficheros y compilar usando `make`.

arit.h

```
#ifndef _LIB_ARIT
#define _LIB_ARIT

int suma (int a, int b);
int mult (int a, int b);

#endif
```

arit.c

```
#include "arit.h"

int suma (int a, int b)
{
   return a+b;
}

int mult (int a, int b)
{
   return a*b;
}
```

calc.c

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

int main(void)
{
  int a, b;
  printf("Operando 1: ");
  scanf("%d",&a);
  printf("Operando 2: ");
  scanf("%d",&b);
  printf("%d+%d=%d\n",a,b,suma(a,b));
  printf("%d*%d=%d\n",a,b,mult(a,b));
  return 0;
}
```

makefile

```
#
# Programa de calculadora
#

calc: calc.o arit.o
<TABULADOR>gcc -o calc calc.o arit.o

calc.o: calc.c arit.h
<TABULADOR>gcc -c calc.c

arit.o: arit.c arit.h
<TABULADOR>gcc -c arit.c
```

Ejercicios

1. Usando el procedimiento del ejercicio 2 del seminario anterior, escribe un programa que multiplique dos matrices **A** y **B** leídas de ficheros, la primera de tamaño **n×m** y la segunda de tamaño **m×p**. Las matrices deben leerse cada una de un fichero distinto y el resultado debe estar en otro fichero. El formato del fichero es el siguiente: en la primera línea habrán 2 enteros, indicando el tamaño de la matriz en cada dimensión. A continuación vienen las filas de la matriz, cada una en una misma línea. Los valores son números reales. El resultado debe estar en otro fichero con el mismo formato.

   Por ejemplo, un fichero de definición de una matríz podría ser el siguiente:

```
3 4
1    0.1  0    1.2
0    1.2  3.4  0.3
2.8  2.4  5.3  5.2
```

   Los nombres de los ficheros serán los argumentos del programa o (si no existen) se solicitarán a través del teclado.

2. Dado el programa escrito para el ejercicio 3 de la 3º sesión del seminario (cálculo del producto de dos matrices) dividir el código en tres módulos: **Ficheros** (funciones de entrada salida), **Matematico** (tipo de datos matriz y producto de matrices) y **Principal** (programa principal). Crea un fichero `makefile` para generar automáticamente el programa ejecutable.

3. Escribe un programa que dado un nombre de fichero, que se pasará como parámetro en la línea de comandos, haga que la primera letra de cada palabra esté en mayúsculas. Se deberá poder compilar en "modo debug", mostrando en ese caso los principales pasos de ejecución. Crea un fichero `makefile` para generar automáticamente el programa ejecutable.