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 2

Contenidos:

1. Estructuras (registros) y uniones
2. Estructura de la memoria (línux)
3. Punteros
4. Arrays

Ejercicios

---

1. Estructuras (registros) y uniones

- Una estructura o registro en C es como una clase en java, pero:
  - sin métodos (sin funciones), sólo con campos de datos (*miembros*).
  - con todos esos campos públicos
  - usando la palabra clave *struct* en vez de *class*

- Ejemplo:

```
struct persona {
   unsigned long DNI;
   char nombre[100];
   int edad;
   enum sexo s;
};
struct persona pers1, pers2;
```

- En C no existen las clases; en C++ sí.

Importante diferencia en C respecto a java: aunque una estructura es un tipo de dato compuesto (no básico), funciona como un tipo elemental en cuanto a que es una ***referencia directa***, es decir:
- `pers1=pers2` copia los campos de pers2 sobre los de pers1 (no la referencia)
- `pers1== pers2` compara las dos estructuras miembro a miembro, en vez de comprobar si pers1 y pers2 son referencias a la misma estructura

- Al igual que en java, se accede a los miembros utilizando la **notación punto**:

```
variable.miembro
pers1.DNI = 27722;
printf("%s\n", pers1.nombre);
pers2.edad= pers1.edad + 1;
```

- **Inicialización de registros** (en la declaración). Indicar entre llaves el valor de cada miembro, en el mismo orden.

```
struct persona pers1= {77000000, "Juanito", 12, hombre};
```

- **Uniones.** Una **unión** es como un registro, pero donde todos los campos ocupan (comparten) la misma posición de memoria.
  - **Conclusión:** los miembros de la unión son *excluyentes*.
  - Su uso es mucho menos frecuente.
  - Utilidad: optimizar uso de memoria.
  - No existen en java.
  - Ojo: escritura/lectura deben ser coherentes, por el mismo campo.
  - Ejemplos:

```
union numero {
   int comoInt;
   float comoFloat;
   double comoDouble;
} n1;
n1. comoInt= 4;
printf("%g", n1. comoDouble);
```

```
union identificador {
   unsigned long DNI;
   long Npasaporte;
   char nombre[100];
};
union identificador id1, id2;
```

- **Definición de tipos.** C permite dar nombre a los nuevos tipos definidos (estructuras, registros, enumerados, etc.):

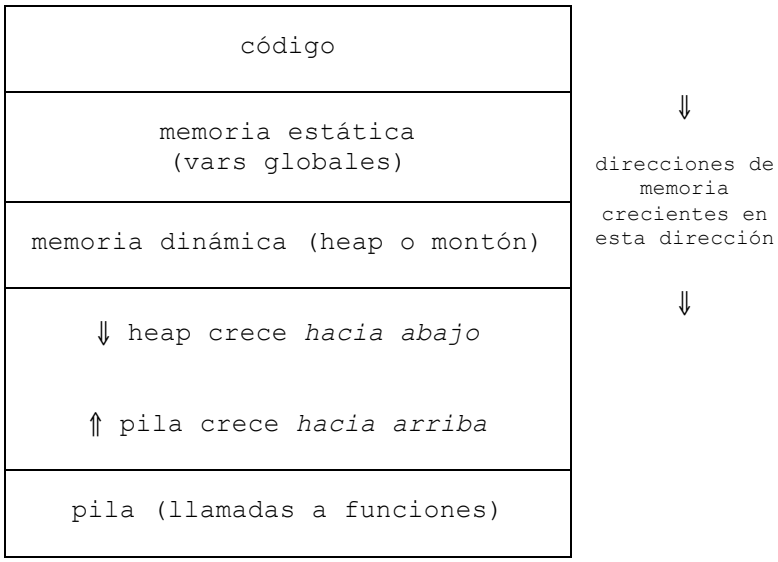
```
typedef expresión_tipo nombre_nuevo;
```

Ejemplos:

```
typedef unsigned char byte;
typedef byte * byte_pointer;
typedef struct persona tipo_persona;

byte b1= 1, b2[10];  /* Equivalente a: unsigned char b1, b2[10]; */
byte_pointer pb1= b2;  /* Equivalente a: unsigned char *pb1= b2; */
tipo_persona pers1= {200, "Pepito", 11, hombre}, *pp;
printf("Tamaño de persona: %d\n", sizeof(tipo_persona));
pp= &pers1;
pp->nombre[5]= 'a';
printf("Nombre: %s\n", pers1.nombre);
```

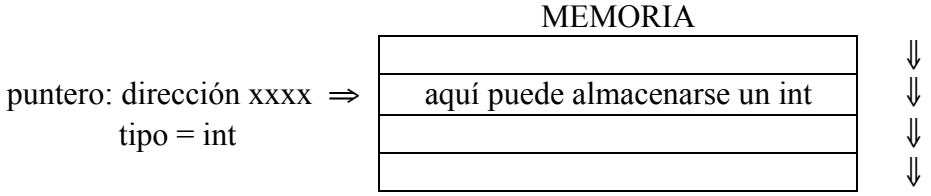
2. Estructura de la memoria (línux)



- La memoria ocupada por código y memoria estática está reservada para esos propósitos desde que el principio al fin de la ejecución del programa. No podemos liberar esa memoria, ni solicitar más memoria.
- La memoria dinámica es gestionada de forma explícita por el programador: podemos solicitar memoria y liberar memoria solicitada anteriormente. Según reservamos más memoria, la memoria dinámica crece hacia abajo. La memoria que solicitemos permanece reservada hasta que la liberemos, independientemente de desde qué punto del código la hemos reservado.
- La pila es gestionada automáticamente. Almacena lo necesario para implementar las llamadas a funciones, incluyendo paso de parámetros y variables locales de las funciones. Al acabar la función se libera la memoria. Por tanto es un error intentar acceder a un parámetro o variable local de una función cuando ya ha terminado la ejecución de ésta.

3. Punteros

- Un **puntero** de tipo T es una dirección de memoria donde "se puede almacenar" un valor tipo T. Es decir, un par (dirección de memoria, tipo "almacenable").
- Gráficamente:



- Un **puntero** es parecido a una *referencia indirecta* de java, es decir, al papel de una variable java cuyo tipo sea un tipo compuesto, por ejemplo, `String a`. Pero en C el tipo almacenable es cualquiera, tipos básicos y compuestos.
- **Declaración** de una variable de tipo puntero en C: mediante uso de *****

```
tipo * nombre;
```

```
int *p1, *p2;
float i, *p3, j;
unsigned *p4, k, l= 8;
```

- **Operadores** sobre punteros:
  - **Dirección: &**. Dada una variable de tipo T, devuelve un puntero a esa variable de tipo "puntero a T" (e.d., TIPO " `T *` ").

```
int a, *b;
b= &a; // b recibe la dirección de a
```

Equivalencia en java: rol equivalente al de una variable java cuyo tipo es compuesto (referencia indirecta). Para tipos simples no existe. Ejemplo:

```
String a, b;
a = new String("Hola");
b = a; // b referencia a, sólo hay un string
```

  - **Indirección: ***. Dado un puntero de tipo T, devuelve la variable de tipo T apuntada por ese puntero.

```
int a, *b= &a;
*b= 33;          // izquierda de asignación
a = *b + 2;      // derecha de asignación
```

Equivalencia java: no hay equivalencia directa del concepto, pero los métodos *equals* y *clone* de java tienen algo que ver con esta idea: se ocupan de comparar/asignar lo referenciado, en vez de la referencia. Ej.:

```
String a, b; boolean iguales;
a = new String("Hola");
b = a.clone(); // b recibe copia de a: 2 strings
iguales = a.equals(b); // compara contenido a/b
```

- **Puntero a tipo no definido (apunta a cualquier tipo): `void *`**

```
int i;
float f;
void * p2= &i;
p2= &f;
```

- **Puntero nulo**: **`NULL`** (definido en `stdio.h`) = `(void *) 0`
  - Se puede usar como valor de inicialización (**ojo**, en C no hay inicialización por defecto de punteros, a `NULL` ni a nada, al igual que los tipos numéricos no se inicializan a 0 ni a nada por defecto).
  - Usado en algunas funciones como valor de error.
  - Los punteros se pueden usar como booleanos. `NULL` es **false** y cualquier otra cosa es **true**.

REGLA DE ORO (vers. 2): siempre inicializar "todo" antes de usarlo, ¡ESPECIALMENTE LOS PUNTEROS!

- **Compatibilidad en la asignación** entre punteros:
  - Se pueden asignar punteros del mismo tipo.

```
int *p1, *p2;
int k;
p1= &k;
p2= p1;
```

  - Se pueden asignar punteros `(void *)` a cualquier otro, y viceversa.

```
int *p1;
void *p2;
p1= p2;
p2= p1;
```

  - Se pueden asignar punteros a tipos distintos con casting explícito.

```
int *p1;
float *p2;
p1= (int *) p2;
p2= (float *) p1;
```

  - Lo anterior posibilita la *mezcla* de tipos. Por ejemplo, si una variable contiene el número real 4, ¿qué obtendríamos si interpretáramos su representación como un entero?

```
float f= 4.0;
float *p1= &f;    /* p1 apunta a f */
int *p2;
p2= (int *) p1;   /* p2 apunta a f */
printf("%d", *p2); /* *p2 es f, pero "visto" como un entero */
```

  - ¿Cómo escribir un puntero con `printf`? Probar ejemplos.

- **Aritmética de punteros**: A un puntero se le puede sumar o restar un entero.
  - El valor de puntero avanza (o decrementa) según el tamaño del tipo referenciado. Por ejemplo: `p= p+1`
    - `char *p` → El puntero aumenta 1 byte
    - `int *p` → " " " 4 bytes (`sizeof(int)=4`)
    - `void *p` → " " " 1 byte

  - Ejemplo[1].

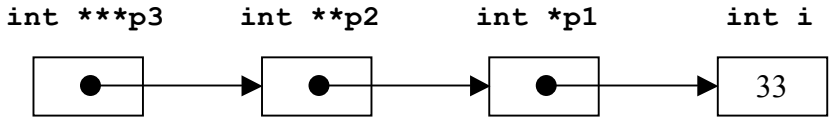
```
float f= 1, g= 2, h= 3, i= 4;   // variables globales
main () {
      float *p1= &f;
      float *p2;
      p1= p1 + 2;           /* Sumar a un puntero un entero*/
      printf("%g\n", *p1);
      p1--;                 /* Decrementar un puntero en uno */
      printf("%g\n", *p1);
      printf("%g\n", *(p1+2));
      p2= &i; p1= &f;
      printf("%d\n", p2-p1);  /* Restar dos punteros */
}
```

¡¡Cuidado con la aritmética de punteros!!
Los resultados de un error pueden ser catastróficos.

- **Otras posibilidades**. Comparar punteros (==), asignar a un tipo entero (casting implícito), usar como un booleano, etc.

- Punteros a punteros a punteros a ….

```
int i= 33;
int *p1= &i;
int **p2= &p1;
int ***p3= &p2;
...
```



- **Punteros a registros**. `(struct persona *)`. Como en java con clases, para acceder a los miembros se puede usar la **notación flecha**: `puntero->miembro`

```
struct persona *pt1;
pt1= &pers1;
pt1->edad = 9;     /* Equivalente a: (*p1).edad = 9; */
pt1->s = nsnc;
```

[1] NOTA: error en página 297 del texto guía (volumen I), porque se usan variables locales a `main`, almacenadas al revés (pila de llamadas a funciones).

4. Arrays

- Un **array** o tabla almacena un número fijo de datos en posiciones de memoria consecutivas.
- **OJO**: no son objetos, al contrario que en java. No tienen métodos. En particular, no hay forma de saber su tamaño, salvo "recordando" con cuál se creó.
- **Definición** de un array en C:
  `tipo nombre [tamaño];`

```
int a[10];
float i, b[20], c[10];
```

- **Ojo**: Sólo se indica el tamaño del array (en vez de rango): `int a[10]`. El primer elemento es siempre `a[0]`, el segundo `a[1]`, …, el último es `a[9]`.

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

ERROR COMÚN: acceder a la posición n de un array de tamaño n

- **OJO**: **no hay comprobación dinámica del rango de los arrays** al acceder a sus elementos; a través de acceso a array, podemos acceder a toda la memoria; si accedemos a posición fuera de la memoria del programa, *segmentation fault*; si no, el sistema no nos avisa, con lo que la detección de errores se complica.
- Los arrays así declarados ya tienen memoria asignada, no se hace `new` posterior.
- Un array **no** puede aparecer a la izquierda de una asignación. Ej.: `b=c`.
  Para hacer una copia de un array sobre otro, copiar elemento a elemento.
- Una comparación del tipo `b==c` no compara el contenido de los dos arrays, sino sus direcciones de memoria. Para comparar el contenido (al estilo de *equals*), hay que comparar "manualmente", elemento a elemento.
- **Inicialización** de los valores en la declaración:

```
int a[4]= {2, 4, 12, 3};
int b[]= {1, 2, 3, 4, 5, 6};      → Se puede omitir el tamaño (será 6)
int c[100]= {1, 2, 3, 4, 5};      → Sólo se inicializan los 5 primeros valores
```

- **Cadenas de caracteres**. En C no existe el tipo "cadena"n (sí en C++), se usan arrays de `char`, donde el número 0 (ó carácter `'\0'`) indica el fin de cadena.

```
char c1[20]= {'H', 'o', 'l', 'a', 0};
char c2[20]= "Hola cadena";
char c3[]= "Así es más fácil";

printf(c1);
printf(c2);
printf("\nLa cadena c1 vale: [%s] y la c3: [%s]\n", c1, c3);
c2[4]= '\n';
c2[5]= 0;        /* Equivalente a c2[5]= '\0'; */
printf(c2);
```

  - **OJO**: almacenar cadena de n caracteres requiere **array de tamaño n+1**, no n
  - **OJO**: cadenas = arrays; copia y comparación elemento a elemento, o usando funciones de librería `string.h` (la veremos): `strcpy`, `strcmp`

- **Arrays n-dimensionales.**

```
int matriz[10][4][20];
int m2[2][3]= {{1, 2, 3}, {2, 3, 1}};
float m3[][4]= {{0., 1., 2., 3.}, {1.1, 1.2, 1.3, 5.4}};
      // en array multidimensional podemos omitir primera dimensión
```

- **Arrays y punteros** (primos hermanos)**:**
  - Un array de tipo T es *equivalente* a un puntero a tipo T.

    ```
    int a[10], *p1;
    *a= 8;              → a equivale a &a[0]
    *(a+4) = 11;        → Acceso al 5º elemento de a, e.d. a[4]
    p1= a + 2;          → a+2 equivale a &a[2]
    ```
  - Se puede asignar un array a un puntero, pero no al revés.

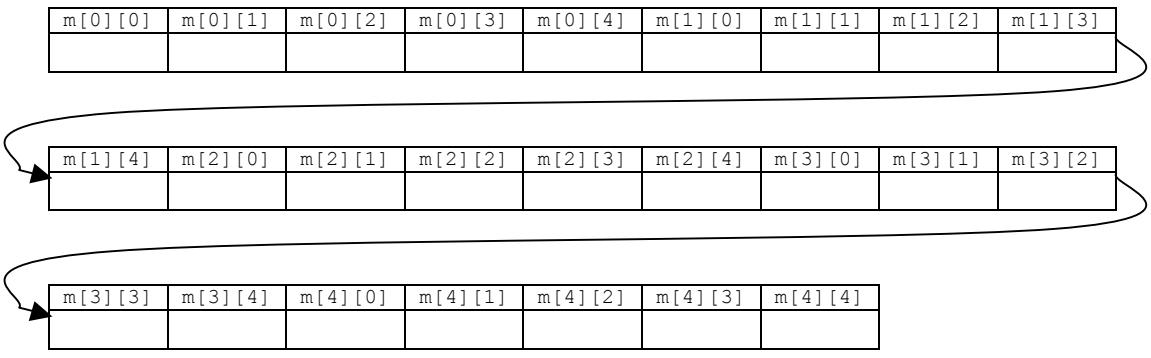
    ```
    p1= a;              → a= p1; daría un error de compilación
    ```
  - Con un puntero a tipo T se pueden usar los corchetes.

    ```
    p1[3]= 38;
    p1[0]= *(p1+1);
    ```

  - Una matriz (array bidimensional) de dimesiones **n**x**m** de tipo T es equivalente a un puntero de tipo T, con **n*m** elementos[2].

```
int i, j;
int mat[5][5];
int *p1= mat, *p2= mat[4];  // p2, dirección de quinta fila
for (i= 0; i<5*5; i++, p1++)
   *p1= 0;
p1= mat;
*p1= 1;
p1[8]= 2;
*p2= 3;
*(p2+2)= 4;
mat[1][1]= 5;
(++p1)[2]= 6;

for (i= 0; i<5; i++) {
   for (j= 0; j<5; j++)
      printf("%d  ", mat[i][j]);
   printf("\n");
}
```



OJO: un array es un puntero, pero su valor como puntero es inmutable, no se puede cambiar.

[2] NOTA: error en texto guía, páginas 300 y 301 (volumen I), comprobar diferencias.

- Arrays, punteros, memoria… ¡vaya mezcla! … ¿qué imprime el este código?:

```
int a=1,B[3]={2,3,4},c=5,d=6;   // variables globales
main () {
      int *pa= &a,*pB=B;
      printf("%d",*pa);
      printf("%d",pa[0]);
      printf("%d",pa[1]);
      printf("%d",pB[0]);
      printf("%d",pB[3]);
      printf("%d",pB[4]);
      printf("%d",*(pa--));
      *(pa--)=0;             // consecuencias !!??
}
```

Ejercicios

1. Encuentra los errores que hay en el siguiente trozo de código e indica la razón.

```
#include <stdio.h>

void media (int *a, int tam) /* 2 errores en esta función */
/* Calcula la media de un array de enteros de tamaño tam
   y devuelve el valor resultante */
{
   register float tot= 0;
   int i= 0;
   for (; i<=tam; i++)
      tot+= *a++;
   return tot/tam;
}

void minimo (int tam, a[]; float *res} /* 4 errores aquí */
/* Calcula el mínimo de un array de enteros de tamaño tam
   y almacena el resultado en res */
{
   res= a[tam];
   for (; tam;)
      *res= *res < a[--tam] ? *res : a[tam];
   return;
}

int main (void)  /* 3 errores */
{
  int eje[3]= {3, 5, 1, 9, 3, 6, 9};
  printf("Media: %g\n", media(eje, 7));
  printf("Mínimo: %g\n", minimo(eje, 7);
  return;
}
```

2. Escribe un procedimiento que reciba como entrada una matriz de float de tamaño **n** × **m**, donde **n** y **m** se pasan como parámetros. La función debe calcular el máximo y el mínimo de cada fila y columna. Esta función debe usar otras funciones más elementales, que calculen el máximo o el mínimo de una sola fila o columna. El resultado final deben ser dos arrays de registros con pares (max: double; min: double). Escribe un programa para probar el funcionamiento de la función.

3. ¿Por qué son válidas las primeras dos asignaciones siguientes pero no la tercera?

```
char cadena1[]= "Bien";
char *cadena2= "OK";

char cadena3[10];
cadena3= "Mal";
```