Algoritmos y Estructuras de Datos
Ingeniería en Informática, Curso 2º

SEMINARIO DE C
Sesión 2

Contenidos:

1. Punteros
2. Arrays
3. Estructuras (registros) y uniones
4. Funciones

Ejercicios

1. Punteros

- Un **puntero** es una dirección de memoria donde se puede almacenar un valor de cierto tipo.

- **Declaración** de un tipo puntero en C: **`*`**

```
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., `T *`).

```
int i=33, *p1;
p1= &i;
```

  - **Indirección: *****. Dado un puntero (o expresión puntero) de tipo T, devuelve el valor de tipo T apuntado por ese puntero (o expresión).

```
int i=33, *p1= &i;
*p1= 22;
```

- **Puntero a tipo no definido: `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).
  - Usado en algunas funciones como valor de error.
  - Los punteros se pueden usar como booleanos. `NULL` es **false** y cualquier otra cosa es **true**.

- **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.

```
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;
```

☞ o 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 */
```

☞ o ¿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

☞ o Ejemplo.

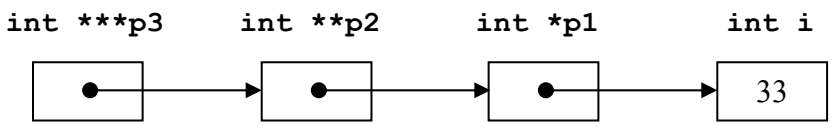
```
float f= 1, g= 2, h= 3, i= 4;

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**. 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;
...
```



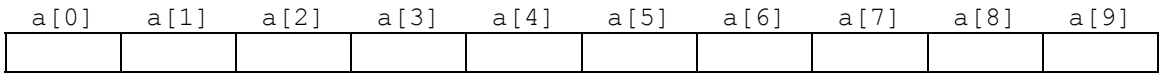
2. Arrays

- Un **array** o tabla almacena un número fijo de datos en posiciones de memoria consecutivas.

- **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. `int a[10]`. El primer elemento es siempre `a[0]`, el segundo `a[1]`, …, el último es `a[9]`.



- **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", 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);
```

- **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}};
```

- **Ojo**: no está definida la asignación entre arrays, ni la comparación. **Solución**: hacer un bucle que copie, o compare, los elementos uno a uno.

- **Arrays y punteros:**
  - 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.

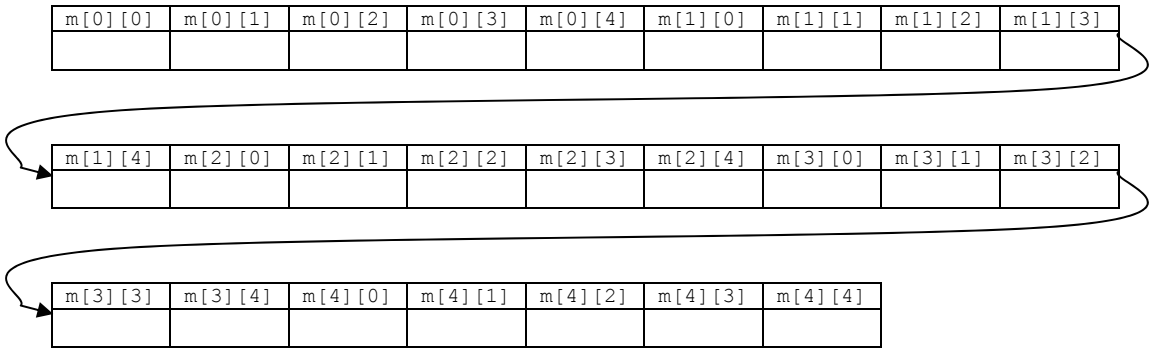
```
p1= a;            → a= p1; daría un error de compilación
```

  - Con un puntero a tipo T se pueden usar los corchetes.

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

- Una matriz (array bidimensional) de dimensiones **n**x**m** de tipo T es "equivalente" a un puntero de tipo T, con **n*m** elementos.

```
int i, j;
int mat[5][5];
int *p1= mat, *p2= mat[4];
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");
}
```



3. Estructuras (registros) y uniones

- Un **registro** (en C, **estructura**) agrupa un conjunto de campos de diversos tipos en un nuevo tipo compuesto.

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

```
struct fecha {
   int dia, mes, ano;
} fecha1, fechas[10];
```

- Los campos de la estructura se denominan también **miembros**.

- Se accede a los miembros utilizando la **notación punto**: `variable.miembro`

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

- **Punteros a registros**. `(struct persona *)`. El acceso a los miembros se puede hacer usando la notación flecha: `puntero->miembro`

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

- **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.

```
union numero {
   int asInt;
   float asFloat;
   double asDouble;
} n1;
n1.asInt= 4;
printf("%g", n1.asDouble);
```

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

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

- **Definición de tipos**: **`typedef`** `expresión_tipo nombre_nuevo;`

```
typedef unsigned char byte;
typedef byte * byte_pointer;
typedef struct persona tipo_persona;
typedef int[10][10] matriz;

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);
```

4. Funciones

- **Estructura de definición de funciones**:

```
tipo_devuelto nombre_función ( parámetros ) { cuerpo }
```

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

- Una función no se puede declarar dentro de otra (no se pueden anidar), aunque sí se pueden definir y anidar bloques: { … { … } … { { } … } …}

- **Valor devuelto.**
  - Sólo puede haber 1 tipo devuelto (como en Módula, Pascal o Java).
  - Si no devuelve nada se pone: **`void`**
  - Por defecto, si no se pone nada, se supone que devuelve un `int`.
  - Se puede devolver un `struct` o `union`, pero no un array. En su lugar, se puede devolver un puntero al array.
  - Acabar la ejecución del procedimiento: **`return;`**
  - Acabar y devolver un valor: **`return`** `expresion;`

- **Parámetros.**
  - Lista de pares: **`(`**`tipo1 nombre1, tipo2 nombre2, ...`**`)`**
  - Cada nombre debe llevar su tipo (aunque se repitan).
  - El paso es siempre **por valor**.
  - Simulación del paso **por referencia**: usar punteros.

```
void suma2 (int a, int *b)
{
   int r= a + *b;
   *b= r;
}
```

  - **Paso de arrays:** no se especifica el tamaño. Alternativa: usar punteros.

```
float media (int array[], int tamano) ...
```

- **Variables locales.**
  - Deben ir siempre al principio del cuerpo de la función, justo después de las llaves, `{`.
  - Se crean en la pila para cada llamada y se eliminan al acabar (`auto`).
  - **Variables locales `static`**: conservan sus valores entre distintas llamadas (equivalentes a variables globales). No usar junto con recursividad.
  - **Variables locales `register`:** *sugiere* al compilador que la variable sea almacenada en un registro de la CPU (a ser posible).

```
float media (int array[], int tamano)
{
   static float acum;
   register int i;
   for (i= 0, acum= 0; i<tamano; i++) /*Ver el uso de la coma*/
      acum+= array[i];
   return acum/tamano;
}
```

- **Declaración de una función.** En algunos casos puede ser necesario *declarar* la sintaxis de la función antes de *definirla* (p. ej. si hay doble recursividad).

```
tipo_devuelto nombre_función ( parámetros );
```

  - En la lista de parámetros se pueden poner sólo los tipos (omitir los nombres).

```
int suma (int, int);
float media (int [], int);
```

- Parámetros y valor devuelto por **la función `main`**.
  - **Valor devuelto.** Si existe, debe ser un `int`. Es el valor devuelto por el programa al sistema operativo, como código de error.
  - **Parámetros.** Indican los argumentos escritos por el usuario en la línea de comandos.
    - Primero: de tipo `int`. Número de argumentos introducidos.
    - Segundo: array de cadenas `(char *)`. Indica el contenido de esos argumentos.

```
main(int num_arg, char *str_arg[]){
   int i;
   printf("Hay %d argumentos.\n", num_arg);
   for(i=0; i<num_arg; i++)
      printf("Argumento %d: %s\n", i, str_arg[i]);
}
```

- **Paso de funciones como parámetros.**
  - Es posible definir **punteros a funciones**, pasarlas como parámetros de una función y aplicar los operadores de dirección e indirección (llamar a la función apuntada).
  - Variable o parámetro de tipo puntero a función:

```
valor_devuelto (*nombre_variable) (tipo1, tipo2, ...)
```

  - Ejemplo.

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

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

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

int opera (int a, int b, int (*op)(int,int))
{
   return(*op)(a, b);
}

main(int argc, char *argv[]) {
   int a, b;
   if (argc<3) return -1;  /* ERROR: faltan operandos */
   a= atoi(argv[1]);
   b= atoi(argv[2]);
   printf("Suma: %d\n",opera(a, b, &suma));
   printf("Prod: %d\n",opera(a, b, &mult));
}
```

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 */
{
   register float tot= 0;
   int i= 0;
   for (; i<=tam; i++)
      tot+= a++;
   return tot/tam;
}

void minimo (int tam, a[]; float *res} /* 3 errores aquí */
{
   res= 0;
   for (; tam;)
      *res= *res < a[--tam] ? *res : a[tam];
   return;
}

int main (void)  /* 4 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";
```