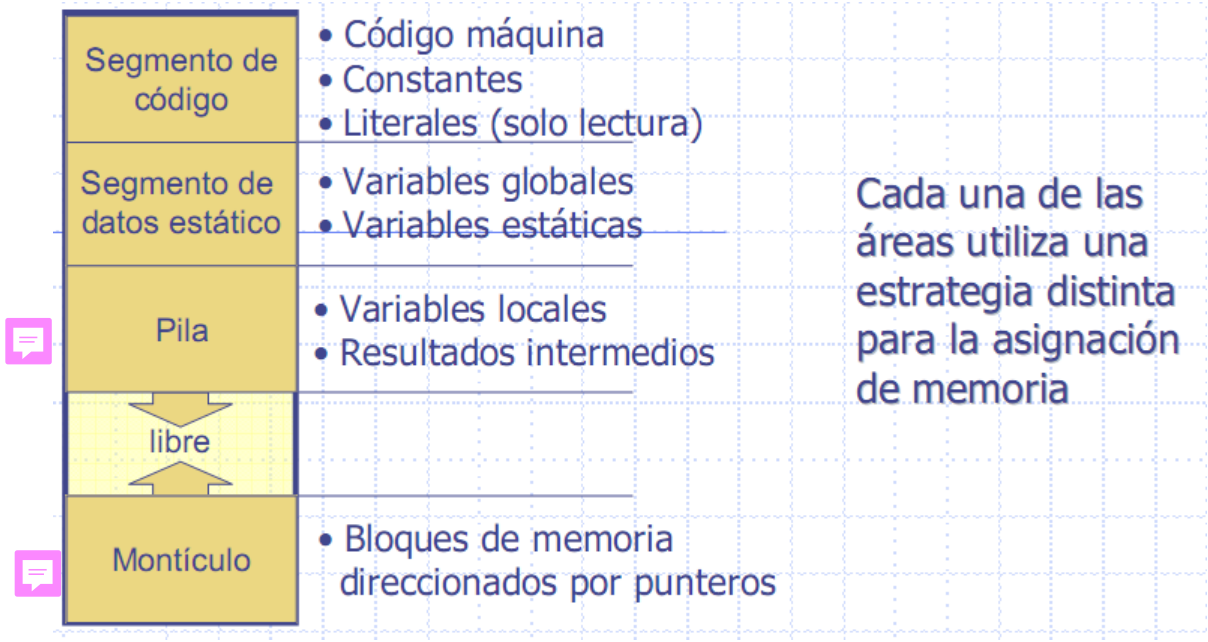
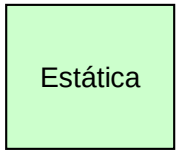
# Manejo Dinámico de la Memoria

# CATEGORÍAS DE DATOS



# CATEGORÍAS DE DATOS

## 1.Estáticos.




La extensión coincide con la ejecución de la totalidad del programa, por lo tanto se determinan en tiempo de compilación.

El ejemplo típico es un array.

El soporte de almacenamiento es un área fija de memoria. La contra es que puede

conllevar desperdicio o falta de memoria.

# CATEGORÍAS DE DATOS

## Los nombres tienen almacenamiento conocido en tiempo de compilación

- El compilador decide donde estará el registro de activación de un procedimiento y la cantidad de almacenamiento para cada variable a partir de su tipo

## Segmento de datos estático

- Estructuras de datos que no cambian su valor en toda la ejecución del programa: Variables globales, estáticas, …
- Acceso a través de direcciones absolutas de memoria
- Asignación de memoria gobernada mediante un puntero a la base del segmento, aumenta con el tamaño de cada estructura de datos

# CATEGORÍAS DE DATOS

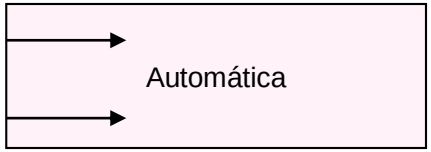
Example

```
int test( int n ) {
    int a, b;
    a = 2;
    b = 1;
    return a * n + b;
}
...
int r;
r = test(10);
```

- Declaraciones: tipo e identificador.
- Su validez depende su foco:
  - Global
  - Locales
  - `static`
- La reserva y liberación de memoria es automática.

# CATEGORÍAS DE DATOS

## 2. Automáticos.




La **extensión** está determinada por el **tiempo que toma la ejecución** de la totalidad de la unidad en la cual se encuentran definidos.

El soporte de almacenamiento es un **stack (pila)** de registros de activación.

# CATEGORÍAS DE DATOS

La memoria para variables locales en cada llamada a un procedimiento está contenida en el registro de activación de dicha llamada

- Las variables locales se enlazan a direcciones nuevas
- Los valores de las variables locales se pierden

No puede utilizarse si…

- Hay que retener los nombres locales cuando finaliza una activación (variables static en C)
- Una activación sobrevive al autor de la llamada

# CATEGORÍAS DE DATOS

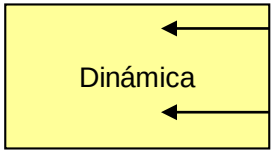

**Texto** Código del programa.

**Datos** Datos del programa.

**Pila** Datos *temporales*.
Estructuras de datos que se crean y se destruyen, por ejemplo las funciones.

# CATEGORÍAS DE DATOS

## 3. Dinámicos.




Su tamaño y forma es variable (o puede serlo) a lo largo de un programa. La **extensión** queda definida por el **programador**, quien los **crea** y **destruye** explícitamente. Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando.

# SOPORTE DE ALMACENAMIENTO

- El soporte de almacenamiento es el bloque de memoria denominado **heap**.
- La memoria se divide en partes contiguas (bloques), según necesidades de activación o datos.
- Las liberaciones de memoria se pueden realizar en cualquier orden
- Con el tiempo el heap tiene partes alternas libres y usadas

# SOPORTE DE ALMACENAMIENTO

Para trabajar con datos dinámicos necesitamos dos cosas:

1. Subprogramas predefinidos en el lenguaje que nos permitan gestionar la memoria de forma dinámica (asignación y liberación).

2. Algún tipo de dato con el que podamos acceder a esos datos dinámicos. Es decir, punteros.

# TIPO PUNTERO

Las variables de tipo puntero son las que nos permiten referenciar datos dinámicos.

Tenemos que diferenciar claramente entre:

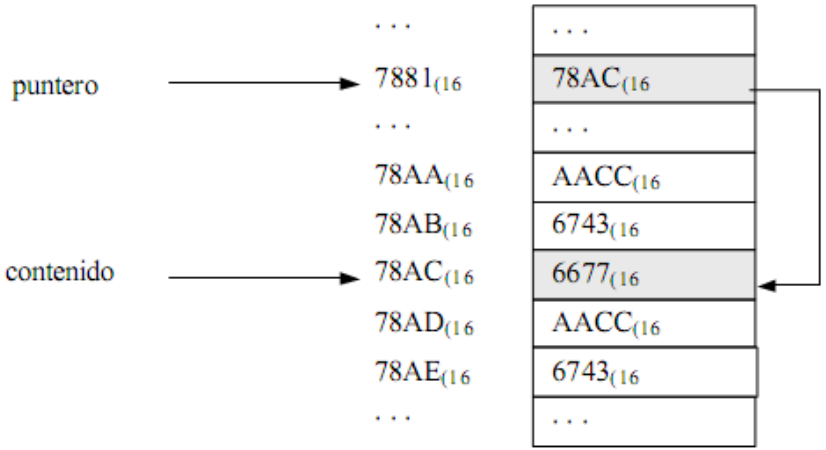
1. la variable referencia o apuntadora, de tipo puntero;

2. la variable anónima referenciada o apuntada, de cualquier tipo, tipo que estará asociado siempre al puntero.

Físicamente, un puntero no es más que una dirección de memoria.

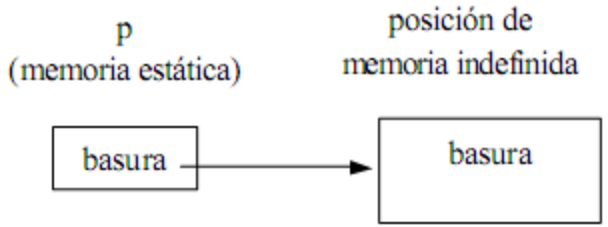
# SOPORTE DE ALMACENAMIENTO

En el siguiente ejemplo se muestra el contenido de la memoria con un puntero que apunta a la dirección 78AC, la cual contiene 6677



# GESTIÓN DE LA MEMORIA DINÁMICA

Cuando declaramos una variable de tipo puntero, por ejemplo int *p; estamos creando la variable p, y se le reservará memoria -estática- en tiempo de compilación; pero la variable referenciada o anónima no se crea. En este momento tenemos:



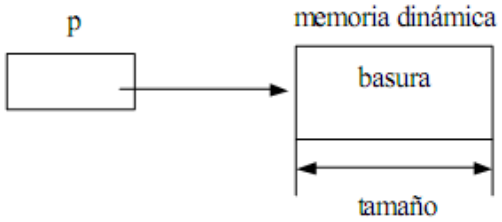
# GESTIÓN DE LA MEMORIA DINÁMICA

La variable anónima debemos crearla después mediante una llamada a un procedimiento de asignación de memoria -dinámica- predefinido. El operador malloc asigna un bloque de memoria que es el tamaño del tipo del dato apuntado por el puntero. El dato u objeto dato puede ser un int, un float, una estructura, un array o, en general, cualquier otro tipo de dato. El operador malloc devuelve un puntero, que es la dirección del bloque asignado de memoria.

puntero =(tipoPuntero) malloc (nombreTipo);

# GESTIÓN DE LA MEMORIA DINÁMICA

En tiempo de ejecución, después de la llamada a este operador, tendremos ya la memoria (dinámica) reservada pero sin inicializar:




Para saber el tamaño necesario en bytes que ocupa una variable de un determinado tipo, dispondremos también de una función predefinida: sizeof(Tipo) que nos devuelve el número de bytes que necesita una variable del tipo de datos Tipo.

# GESTIÓN DE LA MEMORIA DINÁMICA

## Definition

```
void *malloc(size_t size);
```

## Example

```
int_ptr = malloc(sizeof(int) * 10);
```

- Reservamos un `size` de bytes.
- Consejo: `sizeof(tipo)`.
- Nos devuelve un puntero a cualquier cosa,
- Si devuelve *NULL* malo.

# GESTIÓN DE LA MEMORIA DINÁMICA

## Punteros mal apuntados

1. Pedimos memoria.
2. Algo va mal:
   - Confundimos el tipo al pedir espacio.
   - Confundimos el cast.
   - malloc() falla pero no lo comprobamos.

### Example

```
int * p;
p = malloc( 8 * sizeof(char) );
```

3. Violación de segmento (SEGFAULT).

# GESTIÓN DE LA MEMORIA DINÁMICA

El **proceso de devolución** de elementos en desuso a la **lista de espacios disponible** es simple.

El proceso de **identificación de esos elementos** como tales es muy complejo.

# GESTIÓN DE LA MEMORIA DINÁMICA

## Dangling Reference

1. Pedimos memoria.
2. Perdemos memoria:
   - Nos olvidamos de liberar la memoria.
   - Perdemos la referencia a ese bloque de memoria.

Example

```
int * p;
p = (int *) malloc(5 * sizeof(int));
p = (int *) malloc(4 * sizeof(int));
```

3. Somos malos programadores ;-(

# GESTIÓN DE LA MEMORIA DINÁMICA

## Dangling Reference

- Liberar dos veces:

Example

```
p = q;
free(p);
free(q);
```

- Liberar un NULL:

Example

```
p = NULL;
free(p);
```

# GESTIÓN DE LA MEMORIA DINÁMICA

## Dangling Reference

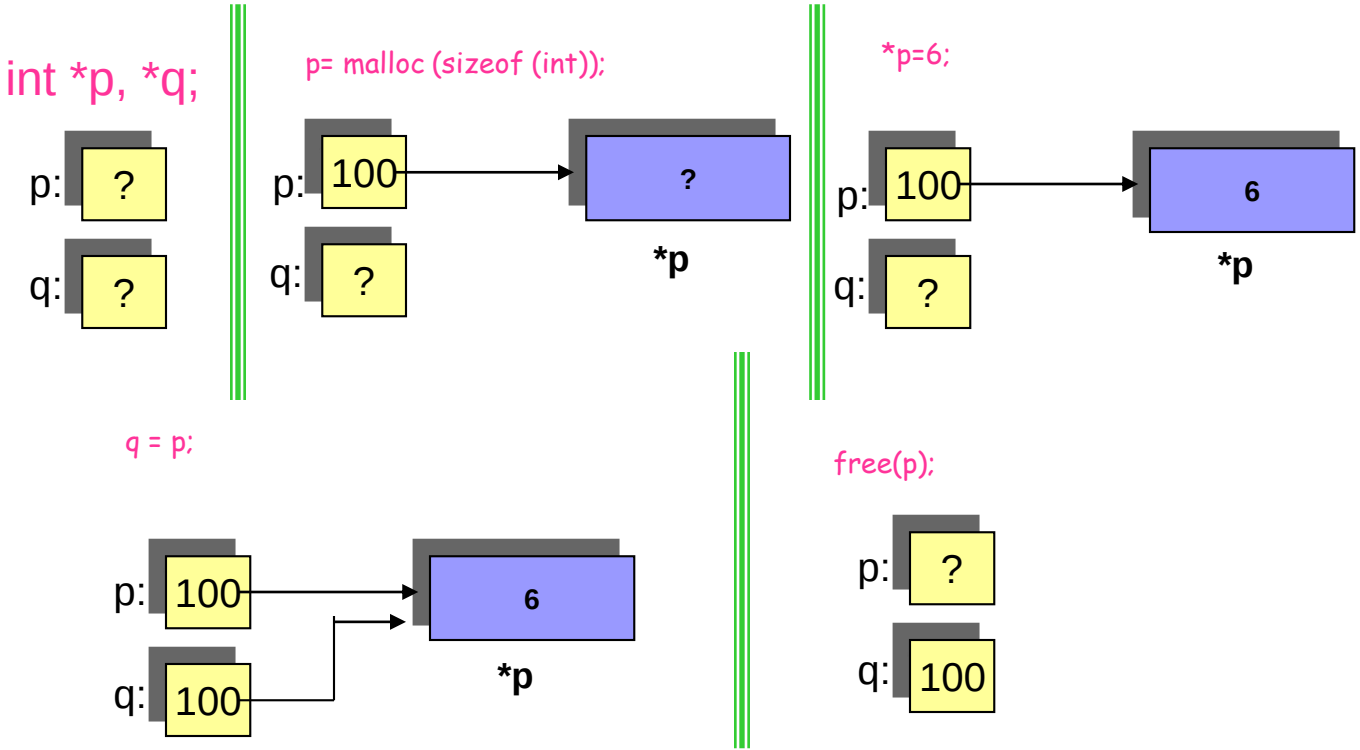
- Referenciar un espacio de memoria liberado:

Example

```
int *a, *b;
int c;
a = (int *) malloc(400);
b = a;
free(a);
c = b[1];
```

El principal problema es que suele funcionar.

# GESTIÓN DE LA MEMORIA DINÁMICA




**q** es una Dangling Refeence

# GESTIÓN DE LA MEMORIA DINÁMICA

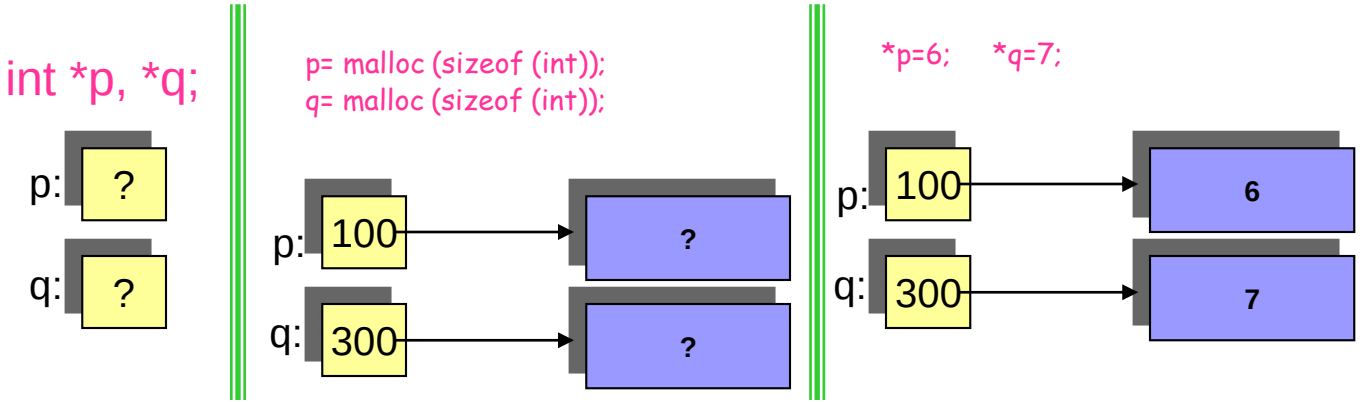
**b).Garbage.** Elemento en condición de ser reutilizado pero inaccesible debido a que **NO fue devuelto** explícitamente a la lista de espacio disponible.

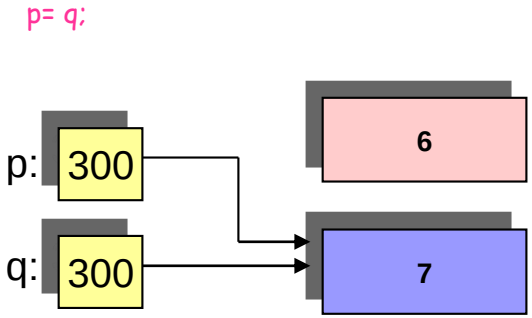
Conceptualmente, en C:

```
int *p, *q;
p= malloc(sizeof(int));
q= malloc(sizeof(int));
*p=6;
*q=7;
 p=q;
```

**Dangling reference** es, potencialmente, **más peligroso** que **garbage**.

# GESTIÓN DE LA MEMORIA DINÁMICA




Los bytes 100 al 104 constituyen garbage

# GESTIÓN DE LA MEMORIA DINÁMICA

- Un puntero puede almacenar una dirección de otro puntero.
- El valor final apuntado puede obtenerse en forma directa.
- “int n, *j, **p”; hacemos “n=4562”; “j=&n”; “p=&j”;
- Supongamos que:
- &n⇔ 0x3021
- &j ⇔ 0x4310 Entonces:
- &p⇔ 0x4F02

**p ⇔ 0x4310**
***p ⇔ j ⇔ 0x3021**
****p ⇔ *j ⇔ 4562**

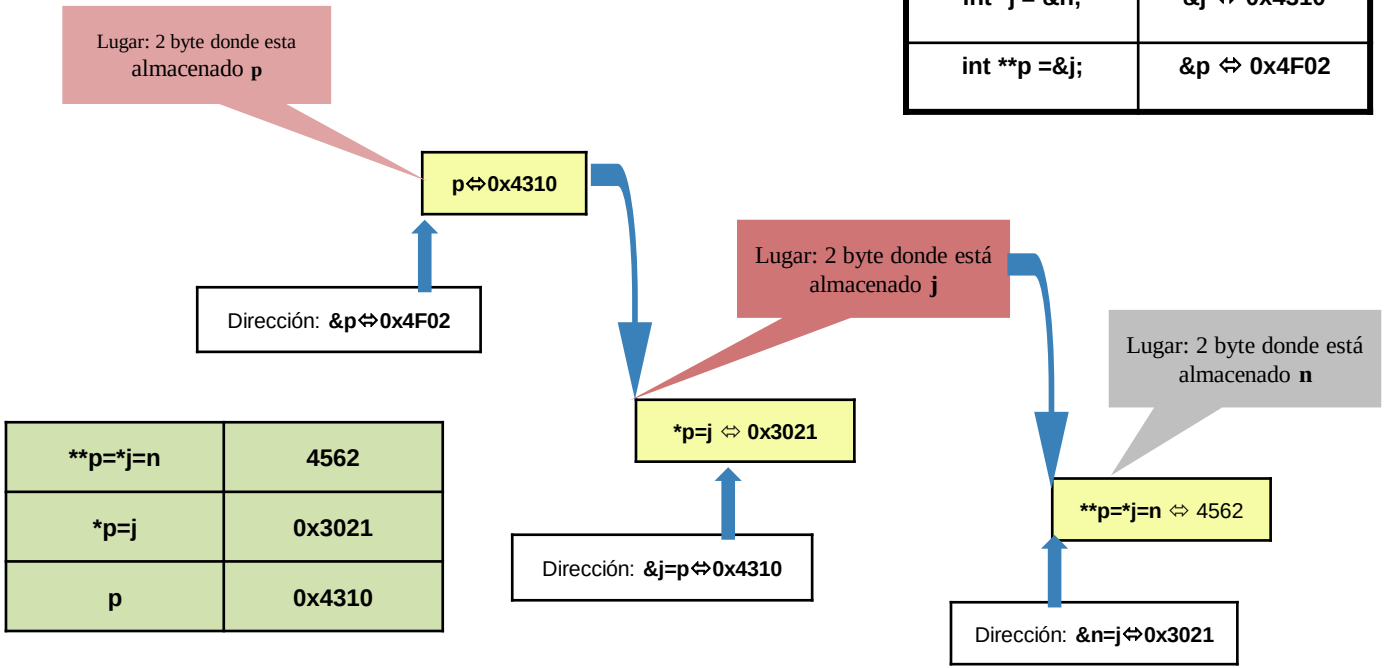
- Interpretación:
  - Con el valor de **p** obtiene la dirección de **j**: ***p ⇔j**
  - Con el valor de **j** obtiene la dirección de **n**: ***j ⇔n**
  - Con el **tipo** de **p** interpreta el valor de **n**.
  - Esta es la secuencia de ➔ ****p**

***(*(p)) ⇔ 4562**
***( j ) ⇔ 4562**
**n ⇔ 4562**

# PUNTEROS A PUNTEROS

**Representación gráfica de doble indirección**

| | |
|---|---|
| **int n= 4562;** | **&n ⇔ 0x3021** |
| **int *j = &n;** | **&j ⇔ 0x4310** |
| **int **p =&j;** | **&p ⇔ 0x4F02** |




| | |
|---|---|
| ****p=*j=n** | **4562** |
| ***p=j** | **0x3021** |
| **p** | **0x4310** |

# PUNTEROS A PUNTEROS

**Como se obtiene el valor de **p**

```
void main(void) {

  char **p, *j[10];
  char *a1 ="Azul";
  char *a2 ="Blanco";
  char *a3 ="Margenta";
  char *a4 ="Verde";
  char *a5 ="Rojo 1";
  char *a6 ="Rojo 2";

  j[0] = a1;
  j[1] = a2;
  j[2] = a3;
  j[3] = a4;
  j[4] = a5;

  p = j;
```

Toma la dirección de **p ⇔0x14E**

**Data:2**

Address: 0x14E Size: 2 | main

| | | |
|---|---|---|
| ⊟ p | 0x130 | * * unsigned char |
| ⊟ *p | "Azul" | * unsigned char |
| **p | 'A' 65 | unsigned char |
| ⊟ j | <20> | array[10] of * unsi |
| ⊟ [0] | "Azul" | * unsigned char |
| * | 'A' 65 | unsigned char |
| ⊟ [1] | "Blanco" | * unsigned char |
| * | 'B' 66 | unsigned char |
| ⊟ [2] | "Margenta" | * unsigned char |
| * | 'M' 77 | unsigned char |
| ⊟ [3] | "Verde" | * unsigned char |
| * | 'V' 86 | unsigned char |
| ⊞ [4] | "Rojo 1" | * unsigned char |
| ⊞ [5] | <2> | * unsigned char |
| ⊞ [6] | <2> | * unsigned char |
| ⊞ [7] | <2> | * unsigned char |
| ⊞ [8] | <2> | * unsigned char |
| ⊞ [9] | <2> | * unsigned char |
| ⊟ a1 | "Azul" | * unsigned char |
| *a1 | 'A' 65 | unsigned char |
| ⊞ a2 | "Blanco" | * unsigned char |
| ⊞ a3 | "Margenta" | * unsigned char |

**Memory**

```
0120 uu uu uu uu uu uu uu uu  uuuuuuuu
0128 19 19 19 19 19 19 19 02  ........
0130 19 02 19 07 19 0E 19 17  ........
0138 19 1D uu uu uu uu uu uu  ..uuuuuu
0140 uu uu uu uu 19 07 19 0E  uuuu....
0148 19 17 19 1D 19 24 01 30  .....$.0
```

**Memory**

```
1900 00 00 41 7A 75 6C 00 42  ..Azul.B
1908 6C 61 6E 63 6F 00 4D 61  lanco.Ma
1910 72 67 65 6E 74 61 00 56  rgenta.V
1918 65 72 64 65 00 52 6F 6A  erde.Roj
1920 6F 20 31 00 52 6F 6A 6F  o 1.Rojo
1928 20 32 00 A7 DE 45 19 02   2...E..
```

# PUNTEROS A PUNTEROS

Sumemos **“p += 2;”**. El valor de ***p** queda:

```
void main(void) {

  char **p, *j[10];
  char *a1 ="Azul";
  char *a2 ="Blanco";
  char *a3 ="Margenta";
  char *a4 ="Verde";
  char *a5 ="Rojo 1";
  char *a6 ="Rojo 2";

  j[0] = a1;
  j[1] = a2;
  j[2] = a3;
  j[3] = a4;
  j[4] = a5;

  p = j;
```

Toma la dirección de **p ⇔0x14E**

**Data:2**

Address: 0x14E Size: 2 | main

| | | |
|---|---|---|
| ⊟ p | 0x134 | * * unsigned char |
| ⊟ *p | "Azul" | * unsigned char |
| **p | 'A' 65 | unsigned char |
| ⊟ j | <20> | array[10] of * unsi |
| ⊟ [0] | "Azul" | * unsigned char |
| * | 'A' 65 | unsigned char |
| ⊟ [1] | "Blanco" | * unsigned char |
| * | 'B' 66 | unsigned char |
| ⊟ [2] | "Margenta" | * unsigned char |
| * | 'M' 77 | unsigned char |
| ⊟ [3] | "Verde" | * unsigned char |
| * | 'V' 86 | unsigned char |
| ⊞ [4] | "Rojo 1" | * unsigned char |
| ⊞ [5] | <2> | * unsigned char |
| ⊞ [6] | <2> | * unsigned char |
| ⊞ [7] | <2> | * unsigned char |
| ⊞ [8] | <2> | * unsigned char |
| ⊞ [9] | <2> | * unsigned char |
| ⊟ a1 | "Azul" | * unsigned char |
| *a1 | 'A' 65 | unsigned char |
| ⊞ a2 | "Blanco" | * unsigned char |
| ⊞ a3 | "Margenta" | * unsigned char |

**Memory**

```
0120 uu uu uu uu uu uu uu uu  uuuuuuuu
0128 19 19 19 19 19 19 19 02  ........
0130 19 02 19 07 19 0E 19 17  ........
0138 19 1D uu uu uu uu uu uu  ..uuuuuu
0140 uu uu uu uu 19 07 19 0E  uuuu....
0148 19 17 19 1D 19 24 01 34  .....$.0
```

**Memory**

```
1900 00 00 41 7A 75 6C 00 42  ..Azul.B
1908 6C 61 6E 63 6F 00 4D 61  lanco.Ma
1910 72 67 65 6E 74 61 00 56  rgenta.V
1918 65 72 64 65 00 52 6F 6A  erde.Roj
1920 6F 20 31 00 52 6F 6A 6F  o 1.Rojo
1928 20 32 00 A7 DE 45 19 02   2...E..
```

# PUNTEROS A FUNCIONES

Al igual que un array (donde la dirección de la primera posición de memoria del mismo, es la dirección de memoria del array) en las funciones, la dirección de la primera posición de memoria que ocupa la función, es la dirección de memoria de dicha función.

Y si las funciones ocupan un lugar en memoria, nada nos impide crear un puntero que apunte a ellas… Un puntero a función, es una posición de memoria, que puede contener la dirección de memoria de una función.

# PUNTEROS A FUNCIONES

Un **puntero a función es una variable** que guarda la dirección de comienzo de la función

- Puede considerarse como una especie de “alias” de la función que hace que pueda pasarse como parámetro a otras funciones
- Las reglas del paso de parámetros se aplican también para el paso de funciones como parámetro

```
X (*fptr) (A);
```

donde fptr es un puntero a función que recibe A como argumento y devuelve X

# PUNTEROS A FUNCIONES

```
void (*fptr)();
```
fptr es un puntero a una función, sin parámetros, que devuelve void.

```
void (*fptr)(int);
```
fptr es un puntero a función que recibe un int como parámetro y devuelve void.

```
int (*fptr)(int, char);
```
fptr es puntero a función, que acepta un int y un char como argumentos y devuelve un int.

# PUNTEROS A FUNCIONES

```
int* (*fptr)(int*, char*);
```
fptr es puntero a función, que acepta sendos punteros a int y char como argumentos, y devuelve un puntero a int.

```
int const * (*fptr)();
```
fptres un puntero a función que no recibe argumentos y devuelve un puntero a un int constante

```
float (*(*fptr)(char))(int);
```
fptr es un puntero a función que recibe un char como argumento y devuelve un puntero a función que recibe un int como argumento y devuelve un float.

# PUNTEROS A FUNCIONES

```
void * (*(*fptr)(int))[5];
```
fptr es un puntero a función que recibe un int como argumento y devuelve un puntero a un array de 5 punteros-a-void (genéricos).

```
char (*(*fptr)(int, float))();
```
fptr es un puntero a función que recibe dos argumentos (int y float), devolviendo un puntero a función que no recibe argumentos y devuelve un char.

```
long (*(*(*fptr)())[5])();
```
fptr es un puntero a función que no recibe argumentos y devuelve un puntero a un array de 5 punteros a función que no reciben ningún parámetro y devuelven long.

# PUNTEROS A FUNCIONES

Ejemplo1

```
#include <stdio.h>

int* fun0(int i) {                  // L.4
  printf("%d", i);
  return &i;
}
int fun1(int i) {     // L.8
  printf("%d", i);
  return 10*i;
}

void fun2(int *pf(int)) { pf(3); } //L.13
void fun3(int (*pf)(int)) { pf(20); } //L.14
```

# PUNTEROS A FUNCIONES

L.4: definición de una sencilla función que recibe un **int** y devuelve un puntero-a-**int**. La cual, si bien parece correcta está encubriendo un error grave ya que la variable, al ser pasada por valor, es copiada en un lugar de memoria en el stack el cuál, al finalizar la función es liberado.

L.8: función que recibe un **int** y devuelve un **int**.

L.13: Función problemática. Supuestamente función que no devuelve nada y recibe una función que recibe un **int** y devuelve un puntero-a-**int** (ver L.4).

L.14: Función que no devuelve nada y recibe un puntero-a-función que recibe un **int** y devuelve un **int**. (ver L.1 de la sinopsis ). Observe que tanto esta como la anterior, ejecutan la función señalada por su argumento.

# PUNTEROS A FUNCIONES

```
int main(void) {                        // =========
  int x = 10;
  int y = fun1(x);                      // M.3
  fun1(fun1(y));                        // M.4
  int (*pf1)(int) = &fun1;              // M.5
  pf1(x);                               // M.6
  fun3(pf1);                            // M.7
  int* (*pf2)(int) = &fun0;             // M.8
  pf2(y);                               // M.9
  fun2(pf2);                            // M.10
  fun2(fun0);                           // M.11
  return 0;
}
```

# PUNTEROS A FUNCIONES

M.3: Definimos un entero y, igualándolo al valor devuelto por la `fun1` definida en L.8; esta sentencia produce la primera salida del programa.

M.4: En esta sentencia, responsable de la segunda y tercera salidas, se muestra el resultado de una invocación recursiva a la `fun1`. La primera invocación (la interior), utiliza el valor y como argumento (un **int**). Observe que la segunda invocación (la exterior) **no** utiliza una función como argumento, en realidad utiliza un **int**, (el valor 1000 devuelto por la primera invocación).

M.5: Definición de `pf1`, un puntero-a-función que recibe un **int** y devuelve un **int**. Lo igualamos a la dirección de la función `fun1` (definida en L.8), que cumple las condiciones exigidas en la declaración.

M.6: Invocamos la función `fun1` utilizando su puntero. Es la responsable de la salida 4.

# PUNTEROS A FUNCIONES

M.7: Ejecutamos la función `func3` definida en L.14, utilizando el argumento adecuado (`pf1` cumple las condiciones exigidas). A su vez ejecuta la función `fun1` señalada por el puntero. Es la responsable de la salida 5.

M.8: Definimos `pf2` como puntero-a-función que recibe un **int** y devuelve un puntero-a-**int**. Lo iniciamos con la dirección de la función `fun0` (definida en L.4) que cumple con los requisitos exigidos.

M.9: Invocamos `fun0` utilizando su puntero y el argumento adecuado. Es la salida 6.

M.10: Aquí está la comprobación del misterio: invocamos la función problemática (fun2), definida en L.13, utilizando el puntero `pf2` como argumento. Es responsable de la salida 7, ya que ejecuta la función `fun0` señalada por su argumento.

# PUNTEROS A FUNCIONES

M.11: Esta sentencia, responsable de la última salida, parece contradecir nuestra hipótesis, ya que aparentemente `fun2` acepta aquí una función como argumento y proporciona una salida coherente. La razón es que en este caso, el compilador construye un objeto temporal de tipo adecuado: puntero-a-función que recibe un **int** y devuelve un puntero-a-**int**, lo iguala a la dirección de `fun0` y lo utiliza como argumento pasado a la función.

# PUNTEROS A FUNCIONES

Con lo que la salida queda finalmente:

```
Salida:
10
100
1000
10
20
100
3
3
```

# PUNTEROS A FUNCIONES

Los punteros a funciones, pueden permitirnos, entre otras cosas **Callbacks.**

Una función **Callback**, es una función que no es llamada explícitamente por el programador. Sino que es llamada por otra función, que a su vez recibe la función **Callback** (la función a llamar implícitamente) como parámetro.
Veamos un ejemplo del uso de los **Callback .**

# PUNTEROS A FUNCIONES

```
typedef struct
{
    int edad;
    int sexo;
    int numero;
} Esclavo;

int main() {
    Esclavo esclavos[20];
    int i = 0;
    for (i = 0; i < 20; i++) {
        srand(time(NULL) + i);
        esclavos[i].edad = rand() % 100 + 1;
        esclavos[i].sexo = rand() % 2 + 1;
        esclavos[i].numero = i;
    }
}
```

# PUNTEROS A FUNCIONES

Simplemente creamos una estructura Esclavo y un array de datos Esclavo, y lo inicializamos con valores aleatorios. Ahora imaginemos que queremos ordenar dicho array de tipos Esclavo, por edad. Entonces hacemos una función ordenar() que implemente un método de ordenamiento ordenando a los esclavos por edad:

# PUNTEROS A FUNCIONES

```
void ordenar(Esclavo esclavos[], int n) {
            int i = 0;
            int j = 0;
            Esclavo buffer;
            for (i = 0; i < n; i++) {
                for (j = 0; j < n; j++) {
                    if (esclavos[j].edad > esclavos[i].edad) {
                        buffer = esclavos[j];
                        esclavos[j] = esclavos[i];
                        esclavos[i] = buffer;
                    }
                }
            }
        }
```

# PUNTEROS A FUNCIONES

Pero… ¿qué pasaría si ahora además de querer ordenarlo por edad, queremos poder ordenarlo por número o por sexo?

Bueno, podríamos hacer 3 funciones distintas: ordenar_por_edad(), ordenar_por_id() y ordenar_por_sexo()…

Aunque eso no estaría nada bien, ¿qué pasaría si quisiéramos después ordenarlo por otro parámetro (como estatura por ejemplo)? tendríamos que crear otra función, repitiendo código… por lo cual tendremos que buscar otra forma de hacerlo.

# PUNTEROS A FUNCIONES

Otra opción menos peor sería que a una función ordenar() se le indique mediante un parámetro si queremos ordenar los elementos por edad, número o sexo, y con una sentencia switch/if se le indique un comportamiento diferente… Obviamente teniendo en cuenta de que si después queremos ordenar el array siguiendo otro parámetro (como por ejemplo ordenar los esclavos por estatura) entonces tendríamos que modificar la función…

Por suerte, hay una mejor manera de hacer esto: usar un puntero a función

# Punteros a Funciones

Podríamos crear una función comparar_por_edad(), otra función comparar_por_id() y otra comparar_por_sexo(), que acepten como parámetro dos datos Esclavo, y devuelvan cual es mayor o menor mediante un int.

Entonces, a la función ordenar() se le podría pasar como parámetro un puntero a función (la función que se va a utilizar para comprar) y dentro del comportamiento de ordenar() utilizar dicha función. De esta manera estaríamos alterando el comportamiento de ordenar() pasándole como parámetro la función que queremos que use para comparar los elementos. Ordenándolos de esta manera como más queramos. De esta manera, tenemos que ordenar a los esclavos por estatura, no vamos a tener que modificar la función ordenar(), sino que simplemente vamos a crear una función comparar_por_estatura(), y se la vamos a pasar como parámetro a ordenar().

# PUNTEROS A FUNCIONES

Las funciones para comparar serían así:

```
int comparar_por_edad(Esclavo a, Esclavo b) {
    return a.edad < b.edad;
}

int comparar_por_sexo(Esclavo a, Esclavo b) {
    return a.sexo < b.sexo;
}

int comparar_por_numero(Esclavo a, Esclavo b) {
    return a.numero < b.numero;
}
```

# PUNTEROS A FUNCIONES

Y la función ordenar() toma como parámetro una función para comparar, y la utiliza para ordenar los elementos del array:

```
void ordenar(Esclavo esclavos[ ], int n, int(*comparar)(Esclavo, Esclavo)) {
    int i = 0;
    int j = 0;
    Esclavo buffer;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (comparar(esclavos[i], esclavos[j])) {
                buffer = esclavos[j];
                esclavos[j] = esclavos[i];
                esclavos[i] = buffer;
            }
        }
    }
}
```

# PUNTEROS A FUNCIONES

Cada vez que queramos ordenar el array, simplemente llamamos a la función ordenar() y le pasamos como parámetro un puntero a una función de comparación (por ejemplo, comparar_por_edad()):

```
1	ordenar(esclavos, 20, comparar_por_edad);
```