LENGUAJE C: PUNTEROS

- Las variables de tipo puntero (**pointer**) contienen direcciones de memoria donde se almacenan los valores. Existe una dirección especial que se representa por medio de la constante **NULL** (definida en <**stdlib.h**>) que se utiliza cuando se quiere indicar que un puntero no apunta a ninguna dirección (útil al comparar asignación de memoria válida).
- En la **declaración** se indica el tipo de dato del objeto referenciado por el puntero y el nombre (identificador) de la variable: <tipo> *<identificador> (int *ptri, char *ptrc).
- Cuando se declara un puntero se reserva memoria para almacenar una dirección de memoria, pero **no para almacenar el dato al que apunta el puntero**. Esta reserva es independiente del tipo de dato ocupado (float, char, int, …).
- Operador **&** (dirección): devuelve la dirección de memoria donde comienza la vble.
- Operador * (indirección): devuelve el contenido del objeto referenciado. Así, *ptr es el contenido de la dirección de memoria almacenada en la posición de memoria ptr.

PUNTEROS – INICIALIZACION

- Operador = (asignación): a un puntero se le puede asignar:

1. Una dirección de memoria concreta:

```
int *ptri;
ptri = 0x0022FFAD;        // como se determina una dirección correcta ?
ptri = NULL;
```

2. La dirección de una variable del tipo al que apunta el puntero:

```
char c;
char *ptrc;
ptrc = &c;                // ptrc apunta a la posición de memoria ocupada por c
```

3. Otro puntero del mismo tipo:

```
char c;
char *ptrc1;
char *ptrc2;
ptrc1 = &c;               // ptrc1 apunta a la posición de memoria ocupada por c
ptrc2 = ptrc1;            // ptrc2 apunta a la misma posición de memoria que ptrc1
```

PUNTEROS – Ejemplo

```
int main()
{
    float fvalor = 2.3;          // 0x40133333
    short hvalor = 255;          // 0xFF
    char ccar = 'A';             // 65 = 0x41

    float *ptrf;          // apunta a cualquier posición
    short *ptrh;          //  no se inicializa (basura)
    char *ptrc = NULL;    // valor conocido

    ptrf = &fvalor;       // referencia a fvalor
    ptrh = &hvalor;       // referencia a hvalor
    ptrc = &ccar;         // referencia a ccar

    hvalor = 5;           // modifico el valor de hvalor
    *ptrh = 579;          // cambio el contenido del
    *ptrc = 'C';          // valor almacenado en la
    *ptrf = 2.7E-3        // memoria apuntada por ptr
    hvalor = (*ptrh) + 5;
}
```

fvalor:	22FF10	33	33	13	40
hvalor:	22FF0E	FF	00		
ccar:	22FF0D	41			

ptrf:	22FF1C	C8	22	2B	00
ptrh:	22FF18	5D	00	00	00
ptrc:	22FF14	00	00	00	00

ptrf:	22FF1C	10	FF	22	00
ptrh:	22FF18	0E	FF	22	00
ptrc:	22FF14	0D	FF	22	00

hvalor:	22FF0E	05	00		
hvalor:	22FF0E	43	02		
ccar:	22FF0D	43			
fvalor:	22FF10	7C	F2	30	3B
hvalor:	22FF0E	48	02		

PUNTEROS – ERRORES COMUNES (1/2)

- Asignar punteros de distinto tipo:

```
int  a = 10;          int *ptri;          ptri = &a;
double x = 5.0;       double *ptrf;       ptrf = &x;
ptrf = ptri;          // ERROR (compilador avisa con warning)
```

- Utilizar punteros no inicializados:

```
char  *ptrc;
*ptrc = 'a';          // ERROR, compilador no avisa pero cuelga el programa
```

- Asignar valores a un puntero y no a la variable a la que apunta.

```
int  n;
int  *ptrn = &n;      // OJO!  se inicializa el puntero y no su contenido!!
ptrn = 9;             // ERROR, se toma un int como dirección de memoria
```

- Intentar asignarle un valor al dato apuntado por un puntero NULL.

```
int  *ptrn = NULL;
*ptrn = 9;            // ERROR,  compilador no avisa pero cuelga el programa
```

PUNTEROS – ERRORES COMUNES (2/2)

- Las variables de tipo puntero deben apuntar al tipo de dato correcto

```
int main()
{
    unsigned char a = 255;                  // 255 = 0xFF
    unsigned int b = 269, c;                // 269 = 0x10D, mas de 8 bits!!

    char *p1;                               // p1 es un puntero a signed char
    unsigned char *p2;                      // unsigned char son 8 bits!!

    p1 = (char *)&a;                        // cast para que no lance un warning
    printf("%d, '%c'", *p1, *p1);           // -1, ' ' (–1 no es un caracter)

    p2 = (unsigned char *)&b;               // p2 sólo toma 8 de los 9 bit
    c = *p2;                        // c almacena los 8 bits menos significativos de b
    printf("\n\nValor = %u\n", c);          // muestra el valor 13 en vez de 169

    return 0;
}
```

- Un puntero con un valor erróneo es **muy peligroso** (resultados inesperados) y es un **error muy difícil de depurar**, la compilación se completa pero la ejecución es impredecible. → **TENER EN CUENTA LOS WARNINGS** y corregirlos.

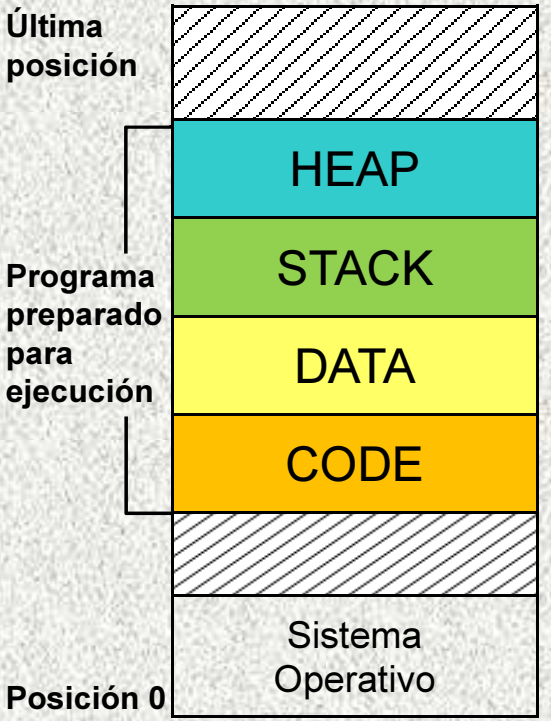
Ej. 2: errores

ORGANIZACIÓN DE LA MEMORIA

- Para poder realizar la ejecución de un programa, el sistema operativo y la CPU adoptan un esquema de partición de memoria con funcionalidades distintas: **code**, **stack**, **data** y **heap**. Se utilizan durante la ejecución a demanda del usuario.
- Cuando se inicia la ejecución del programa, el sistema operativo carga el **código ejecutable** en una zona de memoria que esté libre (**code**), y además, reserva al menos dos espacios más de memoria para que el programa pueda ejecutarse, y almacene allí los datos que necesite: son el **stack** (**pila**) y el **heap** (**montículo**).
- En el **stack** se almacenan los **parámetros de las funciones** (de entrada y retorno) y las **variables locales** utilizadas por cada una.
- El **heap** es utilizado por las funciones que realizan **reserva dinámica** de memoria.
- En aquellos lenguajes que permiten la creación de **variables globales** o **constantes**, (C, Pascal), se reserva además una tercera zona de longitud predeterminada llamada **data** (**zona de datos estáticos**).

ORGANIZACION DE LA MEMORIA

- Cuando se le pide al SO que ejecute un programa, éste carga el código ejecutable en **code**, y además, reserva zonas para **data** (variables globales y constantes), para el **stack** (llamado a funciones y variables locales), y para el **heap** (gestión dinámica de memoria).
- Cuando el programa está listo, el SO carga el **IP** (instruction pointer) de la CPU con el valor de la primera instrucción del código, y el **SP** (stack pointer) que debe apuntar a la primera dirección de la pila.
- La gestión del heap y del área de datos suele quedar a cargo del código del programa, y no de la CPU.
- En algunos casos, el SO no prepara un área de heap para cada aplicación, sino que gestiona un área global para almacenar las demandas de todos los programas.



ASIGNACION DINAMICA DE MEMORIA

- Corresponde a la asignación de zonas de memoria en tiempo de ejecución a demanda del usuario. Los arreglos utilizan una asignación estática (stack); luego de la compilación el tamaño de los objetos no puede cambiarse. El tamaño de la pila es generalmente fijo y limitado, por lo que no se recomienda para el almacenamiento de grandes cantidades de datos.
- El manejo dinámico de memoria requiere de funciones que efectúen la reserva de memoria (solicitud al sistema operativo de un espacio libre) y su posterior liberación cuando la misma no se continuará utilizando. C no realiza **garbage collector**, (recolector de basura) que libera memoria automáticamente cuando se detecta que ya no se utiliza por lo que el correcto uso de la capacidad de memoria es responsabilidad del programador.
- Las funciones dedicadas al manejo de memoria se encuentran agrupadas en la librería estándar **malloc.h** (**malloc()** y **free()** también se declaran en **stdlib.h**).

ASIGNACION DINAMICA DE MEMORIA

- **void *malloc(*size*)**: busca un bloque libre en la memoria para poder almacenar *size* bytes en forma consecutiva y retorna un puntero a la primera posición de ese bloque. Por defecto el tipo de puntero retornado es genérico (**void**). Si se necesita trabajar con otro tipo de datos se debe utilizar el **cast** adecuado. Si no puede hacer la reserva el valor retornado es **NULL** (0x0). No se inicializa el bloque asignado.

```
double *ptd;      ptd = (double *)malloc(30*sizeof(double));
```

  Es asonsejable la utilización de *sizeof()* porque el tamaño de cada tipo de dato depende del compilador, por lo que el bloque de memoria puede ser insuficiente.

- **void free(*ptr*)**: libera la zona de memoria apuntada por *ptr.* Cuando no se utiliza más la zona de memoria debe declararse como disponible para ser reutilizada.

- **void *calloc(*n_elem*, *size_elem*)**: busca un bloque libre para almacenar un bloque de (*n_elem*size_elem*) bytes. A diferencia de *malloc* el bloque apuntado se inicializa con 0. Si no puede reservar la memoria retorna **NULL**.

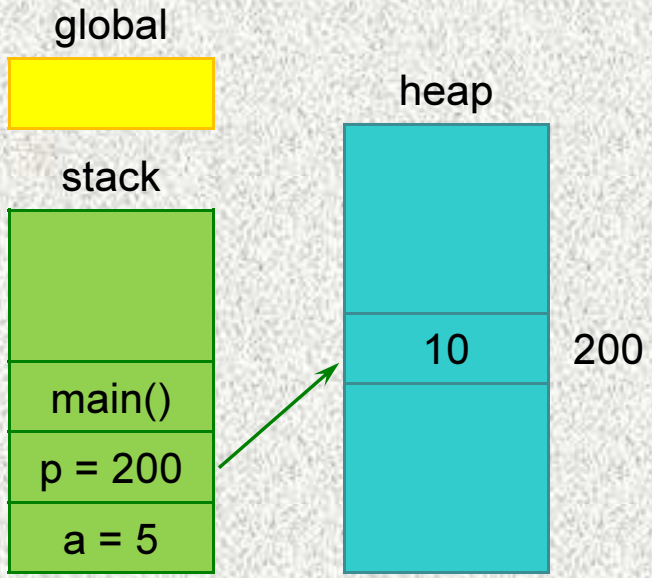
ASIGNACION DINAMICA DE MEMORIA

- **void *realloc(void *_ptr_, _size_)**: cambia el tamaño del bloque de memoria apuntado por *ptr* para que se adapte a *size* bytes. Si el nuevo tamaño es mayor que el anterior y la diferencia (en bytes) no está disponible, entonces se pide una nueva reserva con el tamaño mayor y se copian los datos a la nueva zona, retornando el nuevo valor de la posición inicial del bloque. La zona anterior se marca como libre.

- Todos los valores de *size* utilizados en las funciones son del tipo ***size_t***, que corresponde a un **unsigned int** de acuerdo al estándar C99 (1999 ISO C).

- Ejemplo de asignación:

```
int main()
{
    int a = 5;          // a se almacena en el stack
    int *p;             // p se almacena en el stack
    p = (int *)malloc(sizeof(int));     // p = 200
    *p = 10;            // se almacena en el heap
}
```



EJEMPLO UTILIZACIÓN DE MEMORIA

```
int vec[10];              // GLOBALES
int vec2[5];
float f = 3.14;
char ch = 'A';
int vec3[2] = {20, 21};

void main()
{
   int v2[100];           // LOCALES
   int *ptri;
   void *ptr;
   float *ptrf;
   char ch2[10];
   float f2 = 25.58;
   char ch3 = 'B';

   ptr=malloc(10);        // DINAMICA
   ptri=(int *)malloc(25*sizeof(int));
   ptrf=(float *)malloc(12*sizeof(float));
}
```

DATA

	Posición	Contenido	Δ
vec2	0x40AA28	0000…	
vec	0x40AA00	0000…	**40**
vec3	0x407008	0x14 … 00	248
ch	0x407004	0x41000000	**4**
f	0x407000	0xC3F54840	**4**

STACK

	Posición	Contenido	Δ
ch3	0x22FF07	0x42	
ptr	0x22FF00	0x98186A00	**4**
ptri	0x22FEFC	0xC0186A00	**4**
ptrf	0x22FEF8	0x40196A00	**4**
f2	0x22FEF4	0xD7A3CC41	**4**
v2	0x22FD64	xxxxx	**400**
ch2	0x22FD5A	xxxxx	**10**

HEAP

	Posición	Contenido	Δ
***ptrf**	0x006A1940	xxxxx	
***ptri**	0x006A18C0	xxxxx	128
***ptr**	0x006A1898	xxxxx	40

Ej. 3: memoria

IMPORTANCIA de free()

- La cantidad de memoria estática y global es fija en tiempo de compilación y no cambia en tiempo de ejecución. El stack (variables locales) crece y decrece a medida que el programa está corriendo. Pero la cantidad de memoria reservada dinámicamente puede crecer indefinidamente si se omite el uso de free():

```
int main()
{
    double valor;
    ……for(int cnt=0; cnt<1000; cnt++)
                leer_archivo(nombre, &valor);
……}

void leer_archivo(char nombre[], double *valor)
{
      double *temp = (double *)malloc(2000*sizeof(double));
      ….. /* free(temp);   // olvido usar free()  */
}
```

- Cada vez que se llama a leer_archivo se reserva un bloque de memoria de 16.000 bytes que no se liberan. Se utilizan 16.000.000 bytes!!! = ***memory leak***

ARITMETICA DE PUNTEROS

- Las variables de tipo puntero soportan algunas operaciones aritméticas:
  - Al puntero se le pueden sumar ó restar valores enteros (**+**, **-**, **++**, **--**).
  - Se pueden comparar ó relacionar dos punteros (**<**, **>**, **<=**, **>=**, **==**, **!=**).
  - Es válida también la comparación con **NULL** .

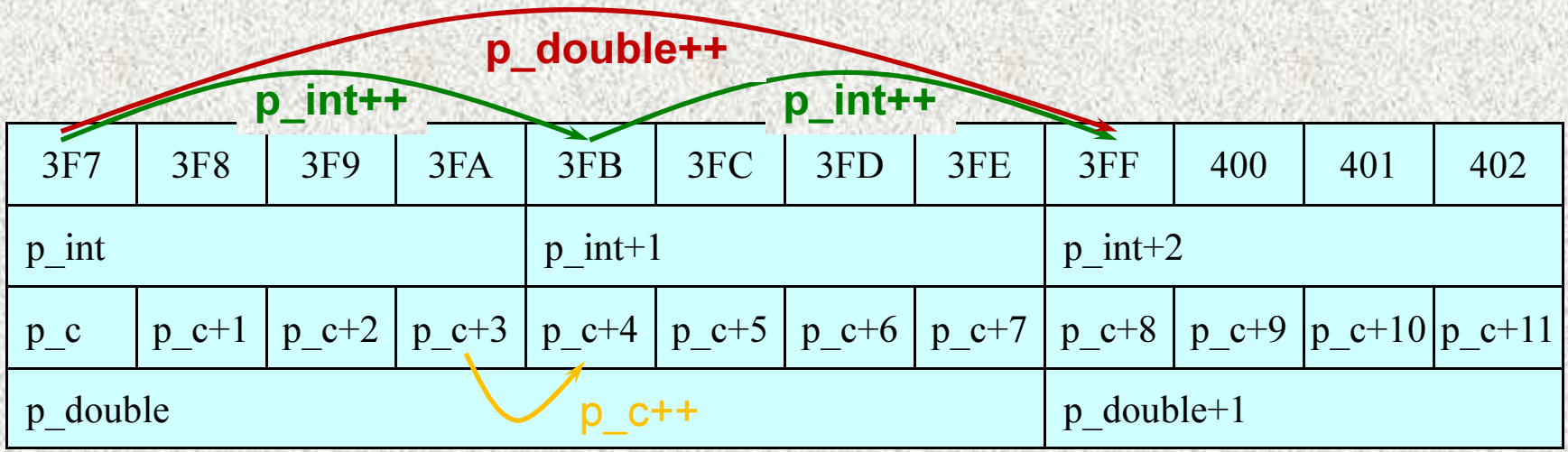
- Ejemplo: la siguiente función copia una cadena de caracteres (referenciada por el puntero *orig*) en otra (referenciada por el puntero *dest*); sin alterar la original.

```
void copiar(char *dest, const char *orig)          // const previene la modificación
{
    if(orig && dest)                  // equivalente a (orig != NULL) && (dest != NULL)
    {
        while(*orig)                  // mientras *orig != 0x0 (fin de cadena)
        {
            *dest++ = *orig++;        // el contenido de origen se copia en destino
        }                             // puntero++ avanza a la próxima posición
    }
}
```

ARITMETICA DE PUNTEROS

- Las operaciones de suma o resta sobre punteros modifican el valor del mismo dependiendo del tipo del dato apuntado:

```
int *p_int;
char *p_c;
double *p_double;
p_int++;                    // p_int = p_int + sizeof(int)
p_c++;                      // p_c = p_c + sizeof(char)
p_double++;                 // p_double = p_double + sizeof(double)
```



PUNTEROS Y ARRAYS

- Cuando se declara un vector <*tipo*> <*identificador*> [<*dim*>] se reseva memoria para almacenar <*dim*> elementos de tipo <*tipo*> y se hace que <*identificador*> contenga el valor de la primera posición del bloque reservado. **Por lo tanto el nombre de un vector es un puntero**.

- Se puede acceder a los elementos del array de dos formas:
  - mediante su índice
  - mediante su dirección de memoria.

Ejemplo:

```
char v[]="Colección";
char *p=v;
```

C	o	l	e	c	c	i	ó	n	\0
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]
*p	*(p+1)	*(p+2)	*(p+3)	*(p+4)	*(p+5)	*(p+6)	*(p+7)	*(p+8)	*(p+9)

PUNTEROS Y ARRAYS – Ejemplo

- Se declara un vector y un puntero al cuál se le asigna la misma dirección:

```
int lista[6] = { 10, 7, 4, -2, 30, 6};
int *ptr = lista;        // es equivalente a  ptr = &lista[0];
```

memoria	1500	1504	1508	150C	1510	1514
contenido	10	7	4	-2	30	6
índice	lista[0]	lista[1]	lista[2]	lista[3]	lista[4]	lista[5]
dirección1	*lista	*(lista+1)	*(lista+2)	*(lista+3)	*(lista+4)	*(lista+5)
dirección2	*ptr	*(ptr+1)	*(ptr+2)	*(ptr+3)	*(ptr+4)	*(ptr+5)

```
int  x = lista[1];      // asigna x = 7
x  = *(lista+4);        // otra forma de usar el vector,    x   =
x = ptr[3];             // otra forma de usar el puntero,   x   =
ptr = &lista[3];        // modifica el valor del puntero,   ptr =
x = *(ptr+2);           // nuevo desplazamiento,            x   =
```

ARITMETICA DE PUNTEROS – Quiz

- Se tiene las siguientes secuencias de sentencias:

```
int arr[10];                    // vector de 10 enteros
int *p_int ;                    // puntero a entero, sizeof(int) = 4
char *pc;                       // puntero a carácter, sizeof(char) = 1;
p_int = arr;                    // p_int apunta a la misma dirección del vector
pc = (char *)p_int;             // pc apunta a la misma dirección del vector
int x=0;
do
{
    printf("\nValor de x = %d",x++);
} while ( (int *)(pc + x) != (p_int + 3) );
printf("\n\nValor de x = %d", x);          // Valor de x =
```

MANEJO DE STRINGS

- Una cadena de caracteres (**string**) se almacena en un array unidimensional tipo char con el carácter '\0' (agregado automáticamente por C) marcando el final.

```
char cadena1[] = "Hola";                    // almacena 5 bytes
char cadena2[] = { 'h', 'o', 'l', 'a', '\0'};   // almacena 5 bytes
char vector[] = { 'H', 'o', 'l', 'a'};          // vector de 4 elementos (ojo! falta '\0')
char cadena3[20] = "Una cadena en C";  // almacena 20 bytes
char cadena4[] = "";                        // almacena 1 byte = 0x0
```

- Se puede calcular la longitud de una cadena:

```
int largo=0;
while (cadena[largo] != '\0')
    largo++;
```

- **No se puede hacer** cadena1 = "Hola"; (salvo en la declaración), **ni se puede copiar una cadena a otra** (cadena2 = cadena1). Usar **string.h**.

FUNCIONES EN string.h (1/2)

- Copiar una cadena de caracteres (*fuente*) en otra (*destino*), incluyendo '\n':
  - char ***strcpy**(**destino*, const char **fuente*); // fuente→destino. Retorna *destino

    No realiza comprobación de tamaños, puede sobreescribir información.
  - char ***strncpy**(**destino*, const char **fuente*, *n*); // copia sólo *n* caracteres (+ 0x0s)
- Obtener la longitud de un string:
  - int **strlen**(const char **str*); // retorna la longitud de *str* (no considera el \0 final)
- Comparar strings:
  - int **strcmp**(const **str1*, const **str2*); // =0→iguales, >0→*str1*>*st2*, <0→*str1*<*str2*

    La comparación se realiza como caracteres (no longitud) → 'A' < 'B'; 'a' > 'A'
  - int **strncmp**(const **str1*, const **str2*, *n*); // compara sólo sobre *n* caracteres
- Buscar la aparición de una cadena (*str2*) dentro de otra (*str1*):
  - char ***strstr**(const char **str1*, const char **str2*); // no modifica *str1* ni *str2*

    Retorna un puntero con la primer ocurrencia o NULL en caso de no producirse.

FUNCIONES EN string.h (2/2)

- Concatenar (unir) dos cadenas de caracteres *str1* y *str2*:
  - char ***strcat**(*str1*, const char **str2*); // agrega *str2* al final de *str1* y retorna *str1*

    No realiza comprobación de tamaños, puede sobrescribir información.
  - char ***strncat**(*str1*, const char **str2*, *n*); // agrega sólo *n* caracteres de *str2*

    En ambos casos *str1* debe tener suficiente espacio para los nuevos caracteres.
- Buscar la aparición de un caracter *c* (o varios) en un *str*:
  - char ***strchr**(const char **str*, *c*); // retorna puntero a la primer ocurrencia o NULL
  - char ***strrchr**(const char **str*, *c*); // retorna puntero a la última ocurrencia de *c*
  - int **strcspn**(const char **str1*, const char **str2*); // retorna un índice

    Busca la primera aparición de alguno de los caracteres de *str2* en *str1*.
- En la librería también se declaran funciones que trabajan directamente sobre bloques de memoria (**buffers**), que en algunos casos son más eficientes que su equivalente vectorial. Siempre se debe conocer el tamaño (no reconocen el \0).

MANEJO DE BUFFERS

- void ***memcpy**(void **dest*, void **src*, *count*); // copia *count* bytes entre bloques

  Se copia byte por byte sin tener en cuenta el tipo del dato a diferencia de strcpy que trabaja sólo con tipo char. Si los bloques se solapan algunos de los datos pueden ser sobrescritos.

- void ***memmove**(void **dest*, void **src*, *count*); // mueve *count* bytes entre bloques

  Puede manejar un solapamiento de memoria, asegurándose que los datos se copien antes de ser sobrescritos.

- void ***memset**(void **dest*, int *ch*, *count*); // inicializa un bloque de memoria con *ch*

  Establece *count* posiciones de memoria a partir de *dest*. Rápido para inicializar. Se puede utilizar también con 0 (solamente) para datos de otro tipo distinto de char.

- void ***memchr**(const void **buffer,* int *ch*, *count*); // busca la aparición de *ch*

  Busca en un bloque de tamaño *count* posiciones de memoria a partir de *buffer*.

Ej. 5 y 6: velocidad

CONVERSIÓN DE TIPOS – stdlib.h

- int **atoi**(const char *_str_); // convierte _str_ a entero, retorna 0 si no puede convertir

  str debe comenzar con números o espacios en blanco.

- long **atol**(const char *_str_); // convierte _str_ a long (mientras sean dígitos o ' ')
- double **atof**(const char *_str_); // convierte str a un double (se permite 'e' o 'E')
- char ***itoa**(int _val_, char *_str_, int _b_); // convierte _val_ (en base _b_) al string _str_
- char ***ltoa**(long _val_, char *_str_, int _b_); // convierte _val_ (en base _b_) al string _str_
- char ***gcvt**(double _val_, int _prec_, char *_str_); (**ecvt**/**fcvt**) // convierte un _val_ real a _str_
- int **sprintf**(char *_buffer_, const char *_format_, ...); similar a printf() pero la salida se envía a buffer en vez de la pantalla. Retorna el número de caracteres escritos.
- double **strtod**(const char *_str ,_ char **_end_); // _end_ contiene el final de _str_
- long **strtol**(const char *_str_, char **_end_, int _base_); // _str_ a long en _base_

Ej. 7: conversiones

CONVERSIONES – ctype.h

- int **tolower**(int *ch*); // retorna la versión minúscula de *ch*
- int **toupper**(int *ch*); // retorna la versión mayúscula de *ch*
- int **isupper**(int *ch*); // retorna !0 si *ch* es mayúscula, 0 en otro caso
- int **isdigit**(int *ch*); // retorna !0 si *ch* es 0-9, 0 en otro caso
- int **isxdigit**(int *ch*); // retorna !0 si *ch* es 0-9 ó a-f ó A-F
- int **isspace**(int *ch*); // retorna !0 si *ch* es ' ', '\t', VT, FF, '\r', '\n'
- int **isalpha**(int *ch*); // retorna !0 si *ch* es una letra del alfabeto

PUNTERO A ESTRUCTURA

- Existen por lo menos tres razones para utilizar punteros a estructuras:
  - Facilita el paso por referencia de una estructura como parámetro. En algunos casos es más simple manejar el puntero que la estructura.
  - Existen datos complejos que utilizan estructuras cuyos miembros contienen punteros a otras estructuras (listas enlazadas).
  - En algunos compiladores no se puede pasar una estructura como parámetro, pero sí un puntero a la misma (en TC genera un warning al pasar por valor).
- La **declaración** se realiza del mismo modo que con los otros tipos de dato:
  - Se debe contar con el prototipo de la estructura (header):

    a) **struct** *nombre_str* {*campos*}; b) **typedef struct** {campos} *new_type*.
  - Se declaran los punteros necesarios:

    a) **struct** *nombre_str* *ptrStruct; b) *new_type* *ptrNewStr;

PUNTERO A ESTRUCTURA

- La **inicialización** del puntero se puede realizar de dos maneras:
  - Asignándole la dirección de una estructura ya declarada, teniendo en cuenta que a diferencia de los arreglos, el nombre de la estructura no es su dirección.

    a) **struct** *nombre_str* datos; ptrStruct = &datos;

    b) *new_type* datos; ptrNewStr = &datos;
  - Reservando memoria en forma dinámica para su almacenamiento.

    a) ptrStruct = (**struct** *nombre_str* *)malloc(sizeof(**struct** *nombre_str*));

    b) ptrNewStr = (*new_type* *)malloc(sizeof(*new_type*));
- El acceso a los datos se realiza mediante el operador flecha: **->** (guión + >), utilizándose de manera equivalente al operador punto '**.**'. **Modos equivalentes**:
  - datos.campo = valor;
  - ptrStruct->campo = valor;
  - (*ptrStruct).campo = valor;

ACCESO A LOS DATOS

```
struct  structura                          typedef   struct
{   int   iNumero;     float  fNumero;     {   int   iNumero;     float  fNumero;
    char  cCar[20];    };                      char  cCar[20];    } myStruc;

int main()
{
    struct estructura st;                                  // variable de struct estructura
    myStruc *pst;                                          // puntero al tipo de dato mySt

    pst = (myStruc *)&st;          // inicializo puntero con la dirección de la estructura
    st.iNumero = 17;       /*op. punto */    pst->iNumero = 33;          /* op. flecha */
    st.fNumero = 23.45;                       pst->fNumero = 19.59;
    strcpy(st.cCar,"Prueba de variable");     pst->cCar[4] = 0x0;        // mensaje corto
    printf("int = %d, float = %7.3f, str = %s\n\n", st.iNumero, pst->fNumero, st.cCar);

    pst = (myStruc *)malloc(sizeof(myStruc));     // apunto a otro bloque de memoria
    pst->iNumero = 0xFF;                          // operador flecha usado con puntero
    (*pst).fNumero = 2.4e5;                       // operador punto usado con puntero
    strcpy( (*pst).cCar, "Prueba de puntero");
    printf("int=%d, float=%7.3f, str=%s\n\n",pst->iNumero,pst->fNumero,pst->cCar);

    free(pst);
    return 0;
}
```

ESTRUCTURAS COMO PARÁMETROS

- Paso por **valor** , para modificar los valores se debe retornar toda la estructura:

```
myStruct presenta_datos(myStruct valores)                    // definición de la función
 {
    printf("El campo char contiene = %s", valores.cCar);        // acceso a los datos
    (&valores)->iNumero = 45;                                  // para modificar el valor
    return valores;                                            // original se retorna la estructura

int main()
{
    myStruct datos;
    datos = presenta_datos(datos);     // la función retorna la estructura modificada
```

- Paso por **referencia**, se utilizan punteros (recordar el operador &):

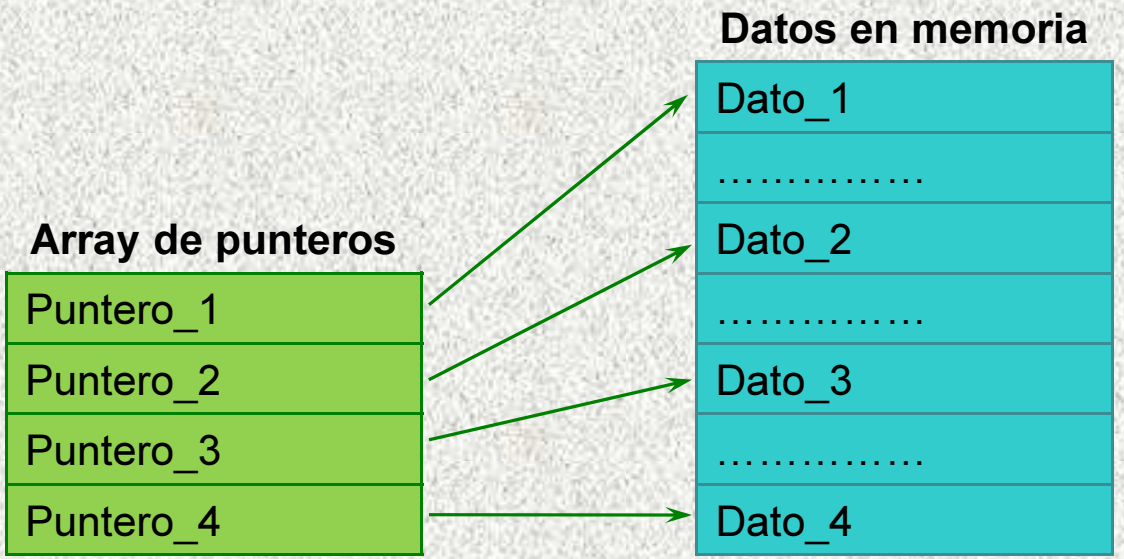
```
void presenta_datos(myStruct *valores)                       // definición de la función
 {
    valores->iNumero = 37;                                   // modifica los datos originales

int main()
{                                          //    forma alternativa:
    myStruct  *datos;                      //    myStruct datos;
    presenta_datos(datos);                 //    presenta_datos(&datos)
```

Ej. 8: ptr2est

ARRAY DE PUNTEROS

- Un arreglo multidimensional puede ser pensado como un vector de elementos que a su vez son vectores (o matrices). Considerando además la equivalencia entre vector y puntero, el arreglo puede ser expresado como un vector de punteros.
- **Declaración**: *tipo_dato* **nombre*[*tamaño*];
- El vector contiene *tamaño* elementos que son direcciones. Cada dirección apunta a datos de *tipo_dato*. **Es preciso inicializar todos los punteros**.
- Un array de punteros a char es similar a un arreglo de strings.



ARRAY DE PUNTEROS – Ejemplo

- Arreglo bidimensional de carateres:

```
/**** array de 3 cadenas de 80 caracteres cada una ****/
char mens[3][80] = { “Inicial”,                    // 8 caracteres
                     “Central”,                    //  8 caracteres
                     “Ultimo” };                   //  7 caracteres
/**** se reserva mucha más memoria de la necesaria ****/
```

- Array de punteros a caracteres:

```
/**** se puede fijar longitud en tiempo de ejecución ****/
char *mens[3];

/**** se podría reservar distinta cantidad de memoria para cada puntero ****/
mens[0] = (char *)malloc(longitud1 * sizeof(char) );
mens[1] = (char *)malloc(longitud2 * sizeof(char) );
mens[2] = (char *)malloc(longitud3 * sizeof(char) );

strcpy(mens[0], “Inicial”);
strcpy(mens[1], “Central”);
strcpy(mens[2], “Ultimo”);
```

Ej. 9: ptr2estArray

PUNTERO A PUNTERO

- Un puntero a puntero representa una indirección múltiple: el primer puntero contiene la dirección del segundo puntero, el cual apunta al dato.

- **Declaración**: *tipo_dato **nombre_puntero*;

```
void main()
 {
    int valor = 5;
    int *pSimple;        // puntero a int
    int **pDoble;        // puntero a puntero
```

pDoble: 22FF1C	02	00	00	00
pSimple: 22FF18	E0	3A	4B	00
valor: 22FF14	05	00	00	00

```
    pSimple = &valor;
```

pDoble: 22FF1C	02	00	00	00
pSimple: 22FF18	14	FF	22	00
valor: 22FF14	05	00	00	00

```
    pDoble = &pSimple;
    printf("valor=%d, *pSimple=%d,
     **pDoble=%d\n", valor, *pSimple, **pDoble);
 }
```

pDoble: 22FF1C	18	FF	22	00
pSimple: 22FF18	14	FF	22	00
valor: 22FF14	05	00	00	00

PUNTERO A PUNTERO – Utilización

- Útil para almacenar cadenas de caracteres de longitudes distintas y variables:

```
int main()
{
    char **cadenas;                          // puntero a cadenas de 5, 7, 12 y 20 caracteres
    cadenas = (char **)malloc(4*sizeof(char *));          // vector de 4 punteros
    *cadenas = (char *)malloc(5*sizeof(char));            // string de 5 elementos
    *(cadenas+1) = (char *)malloc(7*sizeof(char));        // string de 7 elementos
    *(cadenas+2) = (char *)malloc(12*sizeof(char));       // string de 12 elementos
    *(cadenas+3) = (char *)malloc(20*sizeof(char));       // string de 20 elementos

    strcpy(*(cadenas+0),"Sr.");                                  // recordar el 0x0
    strcpy(*(cadenas+1),"Pedro");
    strcpy(*(cadenas+2),"Picapiedra");
    strcpy(*(cadenas+3),"Piedra Dura 2345");

     return 0;
}
```

PUNTEROS Y MATRICES

- Si *datos*[M][N] es un array bidimensional, el elemento *datos*[*i*][*j*] puede accederse también mediante la aritmética de punteros: *(*(*datos*+*i*) + *j*):

```
int main()
{
    int valor = 5;
    int **pDoble; // Puntero a puntero
    int mat[3][4] = { {0,  1,  2,  3}, {4,  5,  6,  7}, {8,  9, 10, 11} };

    pDoble = (int **)malloc(3*sizeof(int *));
    *pDoble = &mat[0];
    *(pDoble + 1) = &mat[1];
    *(pDoble + 2) = &mat[2];

    valor = *(*(pDoble + 1) + 2);          // valor  =
    return 0;
}
```

PUNTEROS Y MATRICES

- Si *datos*[M][N] es un array bidimensional, el elemento *datos*[*i*][*j*] puede accederse también mediante la aritmética de punteros: *(*(*datos*+*i*) + *j*):

```
int main()
{
    int valor = 5;
    int **pDoble; // Puntero a puntero
    int mat[3][4] = { {0,  1,  2,  3}, {4,  5,  6,  7}, {8,  9, 10, 11} };

    pDoble = (int **)malloc(3*sizeof(int *));
    *pDoble = &mat[0];
    *(pDoble + 1) = &mat[1];
    *(pDoble + 2) = &mat[2];

    valor = *(*(pDoble + 1) + 2);              // valor   =   6

    return 0;
}
```

Ej. 10: ptr2ptr

ARGUMENTOS DE main()

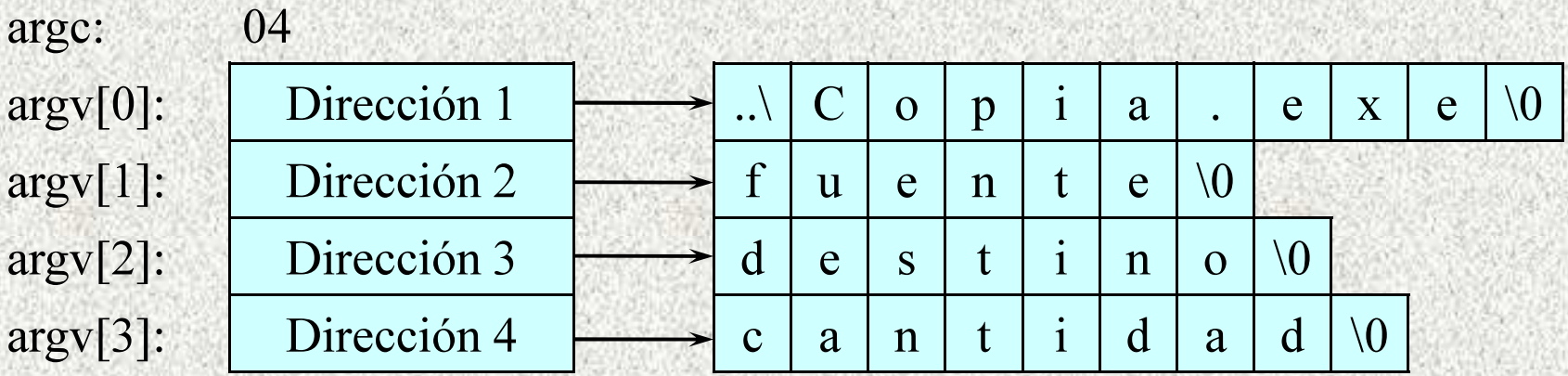
- La función *main*() puede recibir argumentos que permiten acceder a los parámetros con los que es llamado el ejecutable.

- **Declaración**: int **main**(int ***argc***, char ***argv[ ]**)

  int ***argc***: cantidad de parámetros.

  char ***argv[ ]**: strings con cada parámetro.

- Ejemplo: *Copia[.exe] fuente destino cantidad* // línea de comandos




```
printf("Nombre del programa: %s\n", argv[0]);        // incluye el path completo
for (int cnt = 1; cnt < argc; cnt++)
    printf("Parametro_%d: %s\n", cnt, argv[cnt]);    // son todos strings
```

Ej. 11: main_par

PUNTERO A FUNCION

- El nombre de una función también puede ser un puntero. Es útil para pasar como argumento a una función el nombre de otra función (funciones **callback**).
- **Declaración**: *tipo_dato* (**pfunc)(par1, par2, …)*; corresponde a una función llamada *pfunc* que recibe la lista de parámetros *par1*, *par2*, … y retorna un *tipo_dato*.
- Ejemplos:

```
int (*pfuncion1)();                 // pfuncion1 no acepta parámetros y retorna un entero
void (*pfuncion2)(int);             // pfuncion2 no retorna valor y recibe un dato de tipo int
float * (*pfuncion3)(char *, int);  /* pfuncion3 retorna un puntero a float y admite
                                       dos parámetros: un puntero a char y un int */
void (*pfuncion4)(void (*)(int));   /* pfuncion4 retorna void y admite como parámetro
                                       un puntero a otra función que no devuelve valor y recibe un entero */
int (*pfuncion5[10])(int);          /* se declara un arreglo de punteros a función, cada
                                       una admite como parámetro y retorna un int.*/
```

PUNTERO A FUNCION – USO

- Suponiendo la definición de una función de la forma:

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

Se puede definir un puntero a la función de la forma:

```
int (*suma_ptr)(int , int) = suma;                    // no se utilizan los () de suma
```

Nota: aunque algunos compiladores lo admiten, no es recomendable aplicar el operador de dirección (&) al nombre de la función: suma_ptr = &suma;

- Para invocar a la función usando el puntero, sólo hay que usar el identificador (su nombre) como si se tratase de una función.

```
int result = suma(10, 20);        // invocación de suma en forma directa, result = 30
int result = suma_ptr(10, 20);  // invocación en forma indirecta, result = 30
```

- Permite la ejecución de distintas funciones a partir de una sóla. Es de utilidad cuando se personalizan ciertas funciones de biblioteca como **qsort**.

PUNTERO A FUNCION – qsort

- Esta función es propia de C y se encuentra declarada en stdlib.h:

  void **qsort**(void **base*, size_t *numero*, size_t *tamaño*,

  int (**comparar*)(const void **ptr1*, const void **ptr2*));

  Permite ordenar el arreglo *base* (de tipo void) que tiene *numero* elementos de *tamaño* bytes cada uno (size_t es un alias de unsigned int). Dado que *base* puede ser de cualquier tipo, hace falta definir una función de comparación para indicarle a qsort qué elemento es menor, mayor o igual. El cuarto parámetro es un puntero a la función *comparar* que devuelve un int y admite dos parámetros de tipo (void *).

- De esta forma la biblioteca stdlib puede definir una función para ordenar arrays independientemente de su tipo, ya que es el usuario (programador) el que debe realizar la función de comparación. Se debe notar que los punteros *ptr1* y *ptr2* son parámetros de *comparar* y no de qsort. Además se ven afectados por el modificador const por lo que la función callback no los puede modificar.

PUNTERO A FUNCION – qsort (ejemplo)

```
#include<stdlib.h>
int int_cmp(const void *ptr1, const void *ptr2);                    // prototipo de la función
int main()
{
    int (*cmp)(const void *, const void *) = int_cmp;      // puntero a la función int_cmp
    int i, numero = 7, arr[ ] = {5, 4, 3, 2, 10, 1, 7};                  // OJO estandar RPP

    for(i=0; i<numero; i++)    printf("%d ", arr[i]);                    // OJO estandar RPP

    //qsort(arr, numero, sizeof(int), int_cmp);              // usa el nombre de la función
    //qsort(arr, numero, sizeof(int), &int_cmp);                  // forma no recomendable
    qsort(arr, numero, sizeof(int), cmp);              // usa puntero a la función

    for(i=0; i<numero; i++)  printf("%d ", arr[i]);                   // OJO estandar RPP
    return 0;
}
int int_cmp(const void *ptr1, const void *ptr2)
{
    int a = *(int *)ptr1,  b = *(int *)ptr2;                          // OJO estandar RPP
    return (a > b) ? 1 : ( (a == b) ?  0 :  -1);
}
```

Ej. 12: ptr2fun