

METODOLOGÍA DE PROGRAMACIÓN

Alejandro Gómez Amaro

Profesor: Eva Lucrecia Gibaja Galindo

70% Exámen

50% Práctica (Mínimo un 4)

20% Teórico (Mínimo un 4)

20% Resolución

10% Cuestionarios

Punteros

Cuando se declara una variable, se asocian 3 atributos

- Nombre
- Tipo
- Dirección

Ej: int numero=6;

1011
1007
1003
1002
1001

6

numero

Un puntero es un tipo de dato que contiene la dirección de memoria de un dato.

- Tipo de variable* Nombre de la variable

o cualquier tipo predefinido/creado

- Al declararlo se reserva memoria para la dirección de un dato, no el dato en si

Ej: char c='a';

1011
1007
1003
1002
1001

?
?
a

ptri
ptrc
c

Operadores de dirección: & <variable>

- Devuelve la dirección donde empieza <variable>

Ej: int i, *ptri; ptri=&i

asignar valores a punteros

- ptri apunta / referencia a i
- Se puede aplicar incluso sobre punteros

Operadores de contenido: $*$ < puntero >

- Devuelve el contenido referenciado por <puntero>

Ej: char c, *ptrc;

$$ptrc = \&c;$$

$$*ptrc = 'A'; \rightarrow c = 'A'$$

- $*ptrc$ es el objeto apuntado por el puntero, c
- Solo se aplica a datos tipo puntero

Ej: 1. int x = 2;

1011	1	
1007	?	
1003	y	
1002	x	
1001	2	
	...	

1011	2	
1007	1002	y
1003	1003	
1002	2	x
1001	...	

1011	3	
1007	1002	y
1003	1003	
1002	9	x
1001	...	

x = $'y' + 7 = x + 7 = 2 + 7 = 9$	4	
1011	4	
1007	?	mptr
1003	1002	mptr
1002	'3'	z
1001	'5'	y

Ej: 1. void main () {

1007	1	
1003	?	mptr
1002	?	nptr
1001	'5'	y

1007	1	
1003	?	mptr
1002	1002	nptr
1001	'3'	z

2. nptr = &y;

1007	2	
1003	1002	mptr
1002	'5'	nptr
1001	'5'	y

1007	2	
1003	1002	mptr
1002	'5'	nptr
1001	'7'	y

3. z = *nptr; z=y

1007	3	
1003	1002	mptr
1002	'5'	nptr
1001	'7'	y

1007	3	
1003	1002	mptr
1002	'5'	nptr
1001	'7'	y

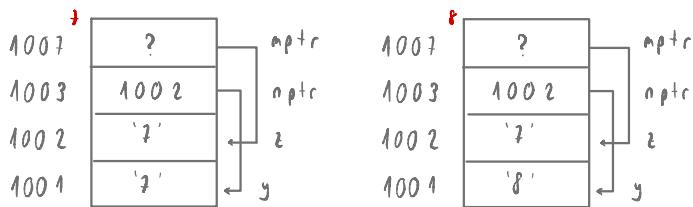
4. *nptr = '7'; y='7'

5. mptr = nptr;

6. mptr = &z;

7. *mptr = *nptr; z=y

8. y = (*mptr + 1); y=z+1



- El tamaño reservado a los punteros son 32 bits normalmente

Para escribir el valor de un puntero utilizamos `printf` y uno de los siguientes códigos :

- `%lu` : visto como decimal 14745564
- `%X` : visto como hexadecimal mayúsculas E0FFDC
- `%x` : visto como hexadecimal minúsculas e0ffdc
- `%p` : visto como hexadecimal minúsculas e0ffdc

Punteros especiales :

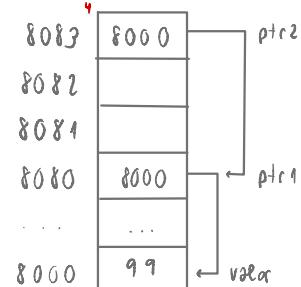
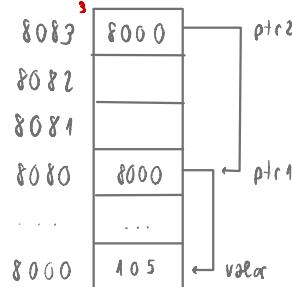
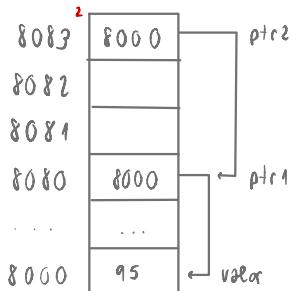
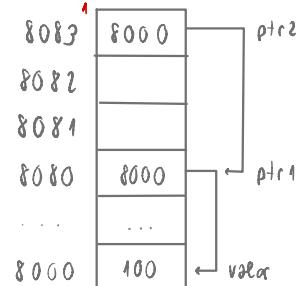
- `NULL`
 - Puntero nulo. No apunta a nada
 - Posibilidad de saber cuando un puntero no direcciona de forma válida
 - Librerías stddef.h, stdio.h, stdlib.h, string.h o #define NULL 0
- `void`
 - Void apunta a cualquier dato
 - `void * pptr;`

Punteros a punteros:

- **Tipo de variable ** Nombre de variable**

Ej:

1. int valor = 100;
2. int* ptr1 = &valor;
3. int** ptr2 = &ptr1;
4. valor = 95;
5. *ptr1 = 105;
6. **ptr2 = 99;



Operadores de estructuras

- Se utilizan punteros a estructuras

Ej:

```
struct punto {  
    int x;  
    int y;  
};
```

```
struct punto a;  
struct punto* p;  
p = &a;           paréntesis necesario, .>*  
printf ("%d, %d", (*p).x, (*p).y);
```

- "p → miembro de estructura"

Ej:

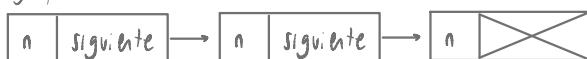
```
struct punto a;
a.x = 2; a.y = 4;
struct punto* p;
p = &a;
printf ("%d, %d", p->x, p->y);
```

Estructuras autoredefinidas

- Estructuras que se definen de forma recursiva

Ej:

```
struct nodoLista {
    int n;
    struct nodoLista* siguiente;
};
```



Temas de listas,
pilas y colas.

Paro de parámetros

- Los módulos se comunican a través de los parámetros
 - Pase de parámetros por valor o por copia
 - La función recibe una copia de los valores parámetro.
 - Si el parámetro se modifica en el módulo, esto no se vera fuera.
 - Pase de parámetros por referencia o por variable
 - La función recibe la dirección de memoria del valor del parámetro.
 - Si el parámetro se modifica en el módulo, esto se vera fuera.

Paro por valor

Ej: int cuadrado (int x) {
 x = x*x;
 return x; }

void main () {
 int a, b = 4;
 a = cuadrado (b); } a=16, b=4

- Tipos de parámetros

- Formales: Identificadores definidos en la declaración de un módulo (cabecera)
- Reales o actuales: Expresiones pasadas como argumentos a una función o en un módulo

Ej: float media 3 (float x1, float x2, float x3)
 { return (x1+x2+x3 / 3.0; } formales (prototipo)
int main () {
 float med, x;
 ...
 med = media (1, 2, x);
 actuales (llamada)

Paso de parámetros con referencia

- En C# todas las parámetros pasan por valor
- El paso por referencia se simula mediante punteros:

1. Cabecera: Delante del identificador del parámetro un *

void modulo (int* parámetroPorReferencia);

- Indica que el parámetro puede cambiar el valor en función de las operaciones dentro del módulo

2. En el módulo: el parámetro es un dato más, si es necesario se usa *

3. En la llamada: el parámetro consiste en la dirección de la variable, obtenida mediante el operador de dirección & (si la variable no es de tipo puntero)

```
int variable, int* ptr = &variable;  
modulo (&variable);  
modulo (ptr);
```

es un puntero

Intercambiar valores de dos variables

Ej: #include <stdio.h>

```
void cambiaBueno (char *a, char *b) { } 1  
    char aux;  
    aux = *a; *a = *b; *b = aux; } 2
```

```
int main () {
```

```
    char x=0, y=1;
```

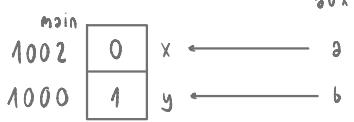
```
    printf ("No cambia Bueno : x=%d y=%d => ", x, y);
```

```
    cambiaBueno (&x, &y); } 3
```

```
    printf ("x=%d y=%d ", x, y);
```

```
    return 0; }
```

main		x
1002	0	
1000	1	y



Cambia Bien	?
1508	1002
1504	1000
1500	

Llamada a función

main		x
1002	0	x ← a
1000	1	y ← b

Cambia Bien	0
1508	1002
1504	1000
1500	

20x = * a

main		x
1002	1	x ← a
1000	1	y ← b

Cambia Bien	0
1508	1002
1504	1000
1500	

* a = * b

main		x
1002	2	x ← a
1000	0	y ← b

Cambia Bien	0
1508	1002
1504	1000
1500	

* b = 20x

main		x
1002	1	
1000	0	y

Pasos de parámetros por referencia

Pasos por referencias entre módulos

Ej: #include <stdio.h>

```
void B(int *vb){  
    *vb = *vb + 5; }
```

```
void A(int *va){  
    *va = *va + 1;  
    B(va); } ← como va ya es un puntero no necesita &
```

```
int main(){  
    int h;  
    h=5;  
    A(&h);  
    printf("%i",h);  
    return 0; }
```

Calcula el cociente y el resto de una división entera

Ej: #include <stdio.h>

```
void CocienteEntero (int divdo, int divisor, int *c) {  
    *c = divdo / divisor; }  
int RestoEntero (int divdo, int divisor) {  
    return (divdo % divisor); }  
void division (int divdo, int divisor, int *c, int *r) {  
    CocienteEntero (divdo, divisor, c); ← no hay que poner &  
    *r = RestoEntero (divdo, divisor); }
```

```
int main () {  
    int a, b, z, w;  
    a = 6;  
    b = 3;  
    Division (a, b, &z, &w); ← primero parámetros de valor  
    printf ("%i entre %i = %i\n", a, b, z);  
    printf ("y el resto es %i", w);  
    return 0; }
```

main	w
1006	?
1004	?
1002	3
1000	6

main	w ←	r ←	División	
1006	?		1006	2008
1004	?	c	1004	2004
1002	3	divisor	3	2002
1000	6	divdo	6	2000

main		División		
1006	?	w ← r	1006	2008
1004	?	z ← c	1004	2004 ← c
1002	3	b ← divisor	3	1004
1000	6	a ← dividendo	6	2304
				2300

main		División		
1006	?	w ← r	1006	2008
1004	2	z ← c	1004	2004 ← c
1002	3	b ← divisor	3	1004
1000	6	a ← dividendo	6	2304
				2300

main		División		
1006	?	w ← r	1006	2008
1004	2	z ← c	1004	2004
1002	3	b ← divisor	3	2002 ← divisor
1000	6	a ← dividendo	6	2000 ← dividendo
				3
				6
				2304
				2300

main		División		Resto Entero
1006	0	w ← r	1006	2008 ← 0
1004	2	z ← c	1004	2004
1002	3	b ← divisor	3	2002
1000	6	a ← dividendo	6	2000

Paso por estructuras o funciones

- Se pueden pasar estructuras o funciones, por el valor o por referencia utilizando & en los llamados.
- Paso por referencia cuando
 - La estructura es grande, el tiempo necesario para copiar struct a los pines puede ser prohibitivo
 - Que se nos cambie el contenido de algún campo de la estructura dentro de la función

Paso por valor

Ej: void funcion1 (struct punto p) {

p.x = -3;
p.y = -3; }

Paso por referencia

void funcion2 (struct punto* p) {
 $p \rightarrow x = -4$; $\rightarrow (*p).x = -4$
 $p \rightarrow y = -4$;

void main () {
 struct punto p;
 p.x = 2;
 p.y = 2;
 funcion1 (p);
 funcion2 (&p);

Punteras y arrays unidimensionales

- El identificador de un vector estático es un puntero constante
 - o es dirección de memoria del primer elemento

Ej: int v[3];
int* ptr;
...

$ptr = \&v[0]$; $\rightarrow ptr = v$



- v es igual a $\&v[0]$

- $*v$ equivale a $v[0]$

Ej: $v[0] = 6 \equiv *v = 6 \equiv *(\&v[0]) = 6$

Aritmética de punteros

- En general

- Asignación ($=$). Deben de ser del mismo tipo o utilizar casting

- $p = q$

- Relacionales

- $p == q$

- $p != q$

- $p == \text{NULL}$

- $p != \text{NULL}$

- Solo con vectores

- Suma / resta con literales (enteras)

- $p = p + n$. Si p es un puntero a un tipo, $p + n$ devuelve un puntero a la posición de memoria $\text{sizeof(tipo)} * n$ bytes

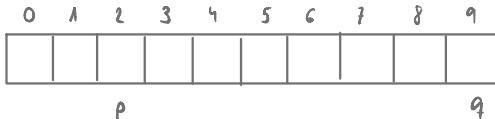
- $p = p + 10$

- $p + 1$

- $p - 1$

- $p + +$

- $p --$



- Relacionales

- $p < q$

- $p > q$

- $q - p$. Número de objetos que separan p y q

- Operaciones no válidas

- Sumar, multiplicar o dividir punteros

Punteras y arrays unidimensionales

- Como puntero, un vector v obedece las leyes de

aritmética de punteros

- v apunta a $v[0]$
- $(v+i)$ apunta a $v[i]$
- $*(v+i)$ es equivalente a $v[i]$



- Recíprocamente, a los punteros se les puede poner corchetes
 - $*(ptr+i)$ es equivalente a $ptr[i]$

- Un vector estático es un puntero constante

Ej: `int v[2] = {0, 1};`

`v++;` → error de compilación

`p = v;`

`p++;` → si es válido

Ej: `int vector[5] = {10, 20, 30, 40, 50};`

vector

`int*`



vector[0]	<code>&vector[0]</code>
vector[1]	<code>&vector[1]</code>
vector[2]	<code>&vector[2]</code>
vector[3]	<code>&vector[3]</code>
vector[4]	<code>&vector[4]</code>

`vector + 0` ≈ `&vector[0]`

`vector + 1` ≈ `&vector[1]`

`vector + 2` ≈ `&vector[2]`

`vector + 3` ≈ `&vector[3]`

`vector + (nElements - 1)` ≈ `&vector[nElements - 1]`

Formas de sumar elementos de un vector

Ej: void main () {
 int i, SUMA, nEle = 5;
 int* ptr, *ptrfin;
 int v[5] = {1, 2, 3, 4, 5};

①

```
SUMA = 0;  
for (i=0; i < nEle; i++) {  
    SUMA = SUMA + V[i]; }  
printf ("Resultado para suma = SUMA + V[i] : %d\n", SUMA);
```

②

```
SUMA = 0;  
ptr = V;  
for (i=0; i < nEle; i++) {  
    SUMA = SUMA + ptr[i]; }  
printf ("Resultado para suma = SUMA + ptr[i] : %d\n", SUMA);
```

③

```
SUMA = 0;  
ptr = V;  
for (i=0; i < nEle; i++) {  
    SUMA = SUMA + *(ptr+i); }  
printf ("Resultado para suma = SUMA + *(ptr+i) : %d\n", SUMA);
```

④

```
suma = 0;  
ptr = V;  
for (i=0; i < nEle; i++) {  
    suma = suma + (*(&V[i])); }  
printf ("Resultado para suma = suma + (*(&V[i])) : %d\n", suma);
```

⑤

```
suma = 0;  
ptrfin = V+nEle;  
for (ptr=V; ptr < ptrfin; ptr++) {  
    suma = suma + *ptr; }  
printf ("Resultado para suma = suma + *ptr : %d\n", suma);
```

⑥

```
suma = 0;  
ptr = V;  
for (i=0; i < nEle; i++) {  
    suma = suma + *ptr++; } → primero * despues ++  
printf ("Resultado para suma = suma + *ptr : %d\n", suma);
```

Punteros y arrays bidimensionales

- Dado un array bidimensional (matriz estática)

Ej: int h[DIMF][DIMC];

$$\begin{pmatrix} 5 & 1 & 9 \\ 7 & 8 & 6 \end{pmatrix} \rightarrow 5 \ 1 \ 9 \ 7 \ 8 \ 6$$

- $h[i]$ es un puntero que apunta al principio de la fila i de la matriz

Ej: $h[0]$ apunta a $h[0][0]$

$h[1]$ apunta a $h[1][0]$

- Para resolver la dirección de $h[i][j]$ el compilador calcula:

- dirección $(i, j) = \text{dirección}(0,0) + i * \text{DIMC} + j$

- $i * \text{DIMC}$ nos permite acceder a las filas

- j indica el desplazamiento en las filas

- No es necesario indicar el número máximo de filas al pasar del array bidimensional a una función

Ej: void imprimirMatriz (int matriz[][][DIMC], int nFil, int nCol)

- Para acceder a $h[i][j]$ utilizamos *:

- $h[i][j] = * \text{dirección}(i, j) = * (\text{dirección}(0,0) + (i * \text{DIMC} + j))$

Elimina los valores negativos de una matriz bidimensional y sustituirlos por ceros

Ej:

```
#define DIMF 3
#define DIMC 2
int main () {
    int i, j, nElem, *ptr;
    int matriz [DIMF][DIMC] = {{1,1}, {-1,-1}, {1,1}};
    nElem = DIMF * DIMC;
    ptr = matriz [0]; → ptr = matriz [0][0]
    for (i=0; i<DIMF; i++) {
        for (j=0; j<DIMC; j++) {
            printf ("matriz [%d][%d]: %d ptr(%d): %d\n", i, j, ...
                    ... matriz [i][j], i * DIMC + j, * (ptr + i * DIMC + j));
    }
    return 0;
}
```

Arrays unidimensionales y funciones

- Un módulo puede recibir de entrada un array unidimensional.
- Como un vector es un puntero, las siguientes cabeceras serían válidas
 - void imprimirElementos (int vector [], int tope);
 - void imprimirElementos (int *vector, int tope);
- Al hacer paso de un vector a una función decimos que hace "paso por referencia"
 - El vector (puntero que pasa por el primer elemento), se pasa por valor, no puede cambiar la dirección de comienzo del vector.
 - Los elementos apuntados por el vector si que pueden modificarse y se verán fuera de la función.

Arrays bidimensionales y funciones

- Un array bidimensional es un puntero al primer elemento de un array de punteros (`int* ptr[]`), no un puntero a puntero (`int** ptr[]`)
- Con memoria estática
 - Notación de corchetes
 - `void imprimirMatriz (int mat[][DIMC], int nfil, int ncol);`
- Con memoria dinámica
 - Notación de punteros
 - `void imprimirMatriz (int** mat, int nfil, int ncol);`

Cadenas de caracteres

Longitud de una cadena

Ej : #include <stdio.h>
#define TOPE 30
void main () {
 char cadena [TOPE] = "Hola";
 int longitud;

char* ptr;

①

for (longitud = 0; cadena [longitud] != '\0'; longitud++)
printf ("\nLongitud : %d", longitud); }

②

for (ptr = cadena; *ptr != '\0'; ptr++)
printf ("\nLongitud : %d", longitud); }

Manejo de cadenas de caracteres

- Las funciones para caracteres están en **string.h**
- **int strlen (char *)**. Longitud de una cadena de caracteres (sin caracteres nulos)

Ej:

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

```
void main () {
    char cad [15] = "Hola";
    printf ("%d", strlen (cad)) } → Imprime 4
```

- **char *strcpy (char *dest, char *orig)**. Copia orig en dest, se suele ignorar el valor devuelto y se usa como void

Ej:

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

```
void main () {
    char cad [15] = "Hola", cad2 [15];
    strcpy (cad2, cad);
```

- **int strcmp (char *cad1, char *cad2)**. Comparar cadenas, devuelve:
 - 0 si iguales
 - <0 si las primeras cadenas preceden alfabéticamente a las segundas
 - >0 si las segundas cadenas preceden alfabéticamente a las primeras

Ej:

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

```
void main () {
    char cad [15] = "Hola", cad2 [15] = "holo";
    if (strcmp (cad, cad2) == 0) {
        printf ("Las cadenas son iguales"); }}
```

- `char *strcat (char *cad1, char *cad2)`
 - Concatena cad2 a cad1. Devuelve cad1, se usa como void.
 - Asegurar que cad1 tiene espacio para concatenar
- `char *strstr (char *cad1, char *cad2)`
 - Busqueda de subcadenas. Devuelve un puntero a la primera aparición de cad2 en cad1 o NULL si no se encuentra
- `char *strchr (char *cad, int c)`
 - Busqueda de un carácter en una cadena. Devuelve un puntero a la primera aparición de c en cad o NULL si no se encuentra
- `char *strrchr (char *cad, int c)`
 - Busqueda de un carácter en una cadena. Devuelve un puntero a la última aparición de c en cad o NULL si no se encuentra

```

#include <stdio.h>
#include <string.h>
void main () {
    char cadena [100] = "Hola como estas";
    char* ptr;
    ptr = strstr(cadena, "como"); ← Busca subcadenas
    printf ("\n<%s> contiene a <como> en la dirección...
... <%p> posición %d", cadena, ptr, ptr-cadena);

    ptr = strchr(cadena, 'o'); ← Busca primera aparición carácter
    printf ("\nPrimera aparición de <o> en <%s>: dirección...
... <%p> posición %d", cadena, ptr, ptr-cadena);

    ptr = strrchr(cadena, 'o'); ← Busca última aparición carácter
    printf ("\nÚltima aparición de <o> en <%s>: dirección...
... <%p> posición %d", cadena, ptr, ptr-cadena);

```

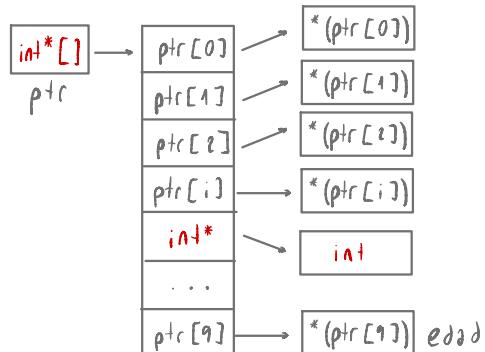
Punteros literales de cadenas

- Conjunto de caracteres encerrados entre comillas
- Ej: `char *ptr = "Hasta luego";`
 - Crea dos entidades
 - El puntero en la zona de datos locales
 - El literal en zona de datos
- Los literales son tratadas como constantes y almacenadas en segmentos de datos
- Un error común es tratar de cambiar el contenido del objeto apuntado por `ptr` siendo constante.

Arrays de punteros

- Cuando se necesiten reservar muchas punteros a muchos valores distintos
- Cada elemento $\text{ptr}[i]$ almacenará una dirección sin necesidad de ser consecutiva al resto

Ej: $\text{int}^* \text{ptr}[10]; \leftarrow 10 \text{ punteros}$
 $\text{ptr}[9] = \& \text{edad};$



```
#include <stdio.h>
void main(){
    int *ptr[10];
    int vector[10];
    int i;
    for (i=0; i<10; i++) {
        vector[i] = i;
        ptr[i] = &vector[i];
        *(ptr[i]) = vector[i];
        printf ("\n*(ptr[%d]): %d", i, *(ptr[i]));
    }
}
```

Punteros constantes y a constantes

	Declaración	Ejemplo	p	*p
Punteros constantes	<tipo> *const <nombrePuntero> = <direcciónDeVariable>	int x; int* const p = &x;	Es constante. No puede cambiar su valor	Es variable. Puede cambiar su valor
Punteros a constantes	const <tipo> * <nombrePuntero> = <direcciónDeConstante>	const int x=25; const int* p=&x;	Es variable, puede cambiar su valor (apuntar a otra constante)	Es constante. No puede cambiar su valor
Punteros constantes a constantes	const <tipo> * const <nombrePuntero> = <dirección de constante>	const int x=25; const int* const p=&x;	Es constante	Es constante

Ej: int y=0;
int x=3;
int* const p =&x; \leftarrow puntero constante

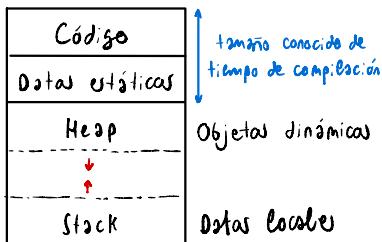
const int x1=4;
const int* p1=&x1; \leftarrow puntero a constante

Const int x2=5;
Const int* const p2=&x2; \leftarrow puntero constante a constante

*p=30; \leftarrow p es constante, *p no } p2 y *p2 son constantes
p1=&y; \leftarrow *p1 es constante, p1 no }

Gestión dinámica de memoria

Organización de la memoria en tiempo de ejecución →



Objetas dinámicos

- Se crean y destruyen a voluntad, según necesidad y tiempo de ejecución
- El número de objetos dinámicos es desconocido, puede variar durante la ejecución
- Se aloja en heap

Identificación

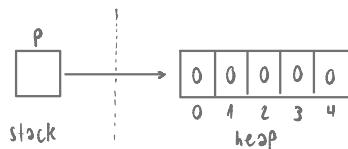
- Cuando declaramos un vector estático debemos conocer su tamaño al compilar
 - No siempre sabemos el tamaño
 - Al poner tamaño "máximo" se desperdicia memoria.
- Para ello C permite gestión dinámica de memoria (solicita memoria para alojarse el contenido de estructuras de datos de tamaño no fijo hasta iniciar el programa)
- Formas de superar las limitaciones de tamaño C
 - Aproximaciones basadas en punteros
 - Vectores con tamaño basado en tiempo de ejecución (consecutivos)
 - Registros enlazados (listas) (no consecutivos)

Gestión dinámica de memoria en C

- Reserva de memoria:
 - `calloc`
 - `malloc`
 - `realloc`
- Liberación de memoria
 - `free`
- `stdlib.h`
 - `nº elementos`
↑
- `void *calloc (size_t nelem, size_t size)`
 - Devuelve un puntero a un vector `nelem` de tamaño `size` bytes cada uno
 - Los parámetros indican `nº` de bytes reservados (`nelem * size`)
 - La reserva de memoria se hace en `heap`
 - Inicializa los huecos con 0
 - Esta reserva no se puede usar hasta liberarla.
 - Se puede hacer casting del valor devuelto a un puntero de tamaño menor o igual a `size`
 - Si no puede satisfacer la petición devuelve `NULL`

Ej:

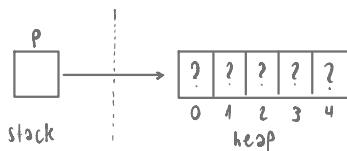
```
int* p, n=5; casting
if ((p = (int*) calloc (n, sizeof (int))) == NULL) {
    printf ("Error: no pudo asignarse memoria");
    exit (-1);
}
```



- **void *malloc (size_t size)**
 - Devuelve un puntero a un vector nulo de tamaño size bytes cada uno
 - Los parámetros indican nº de bytes reservados
 - La reserva de memoria se hace en heap
 - No inicializa ningún valor en las zonas de memoria reservadas
 - Esta reserva no se puede usar hasta liberarla.
 - Se puede hacer casting del valor devuelto a un puntero de tamaño menor o igual a size
 - Si no puede satisfacer la petición devuelve NULL

Ej:

```
int* p, n=5;
if ((p = int*) malloc (n*sizeof(int))) == NULL {
    printf("Error: No pudo asignarse memoria");
    exit(-1);
}
```



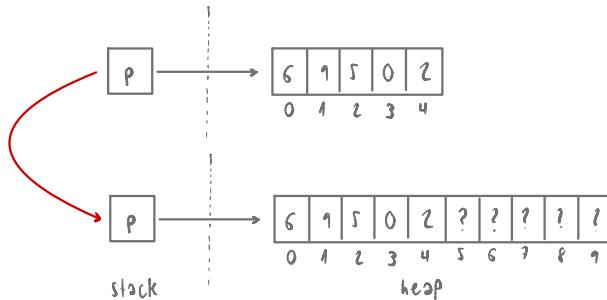
- **void *realloc (void *ptr, size_t size)**
 - Reserva memoria para un objeto tamaño size bytes
 - Devuelve la dirección del nuevo objeto o NULL si hay un error
 - Si ptr es NULL, no almacenar valores en el nuevo objeto → malloc
 - ptr debe ser la dirección de un objeto reservado previamente (c/m/realloc)
 - Si el objeto nuevo < antiguo, copia el objeto previo al inicio del nuevo objeto reservado
 - Si no, copia la parte inicial del objeto previo que cabe en el objeto reservado
 - Si no tiene éxito devuelve NULL y ptr no se pierde
 - Si size es 0 y ptr no nulo el objeto se libera
 - Si realloc reserva memoria, libera la memoria del objeto previo, si no, no cambia

Ej:

```

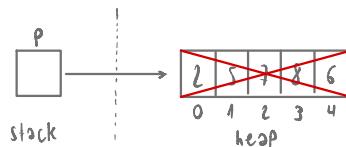
int* p, n=5,
if ((p = int*) realloc (2*n*sizeof(int))) == NULL) {
    printf("Error: No pudo asignarse memoria");
    exit(-1);
}

```



• Void free (void *ptr)

- Libera la zona **heap** referenciada con `ptr`
 - `ptr` mantiene su valor (apuntando a la misma dirección)
- `ptr` debe ser un puntero devuelto de `alloc`, `malloc` o `realloc`
- Si `ptr` es `NULL` no tiene efecto



Ej:

```

int* p;
p = (int*) malloc (5, sizeof(int));
free(p);

```

- Reserva ($n * \text{sizeof}(\text{int})$) bytes
- Marca como reservado la zona en el **heap**
- Hace conversión del tipo devuelto (`void*`) a `int*`
- Puntero `p` contiene la dirección inicial de la memoria o `NULL` si hay algún error

El destino de un puntero

- Un puntero puede tener dos posibles caminos
 - Puede apuntar a un espacio de memoria de otra variable

Ej.: int *a, b = 10;

a = &b;

- Puede apuntar a un espacio de memoria propia

int *a;

a = (int*) malloc (3*sizeof (int));

Consideraciones

- Sobre declaración de un puntero:
 - Valor inicial basura
 - No implica reserva heap
 - Es una variable más (almacena una dirección de memoria)
 - Se aloja en zona de datos globales o en la pila
 - Puede referenciar datos
 - Globales
 - Local (stack)
 - Dinámicos (heap). Se le debe asignar un resultado m/c/realloc.
- Sobre la petición de memoria (malloc, calloc, realloc)
 - Comprobar si devuelve NULL
 - Debe tener void* por lo que requiere casting al tipo de puntero
 - sizeof permite calcular el tamaño del tipo de objeto
 - Se puede reservar memoria para vectores, matrices y estructuras

- Sobre las zonas de memoria asignada
 - Queda reservada y no se asigna a otra petición
 - Puede accederse a ella
 - No se garantiza que el programa no pueda acceder fuera de los límites. Será un error en tiempo de ejecución
- Sobre la liberación con free
 - Esta zona puede reutilizarse
 - La zona liberada queda marcada como libre. No se borra
 - Despues de liberarlas puede accederse a ellas (free no pone a ptr NULL) aunque es desaconsejable
 - Debe haber un free por calloc o malloc

Ej:

```
int* reservaVector (int nElementos) {
    int* p; ← puntero a enteros
    if ((p = (int*) malloc(nElementos, sizeof(int))) == NULL) {
        printf("Error en reserva de memoria\n");
        exit(-1);
    }
    return (p);
}
```

```
void main() {
    int* p;
    int nElementos = 5;
    p = reservaVector (nElementos) ← reserva de memoria
    free (p); } ← libera memoria
```

Ej: void reservaVectorReferencia (int** Vector, int nElementos) {
 if ((*Vector = (int*)calloc(nElementos, sizeof(int))) == NULL) {
 printf("\nError en reserva de memoria\n");
 exit(-1); } }

```
void main () {  
    int* p; ← puntero a enteros  
    nElementos = 5;  
    reservaVectorReferencia (&p, nElementos);  
    free (p); }
```

Reservar vectores de otros tipos

- vector int

Ej: int* p;
p = (int*) malloc (5, sizeof(int));

- vector float

Ej: float* p;
p = (float*) malloc (NULL, 5 * sizeof(float));

- vector struct

Ej: struct punto {
 int x;
 int y; };
struct punto* p;
p = (struct punto*) malloc (5 * sizeof(struct punto));

Reserva de cadenas de caracteres

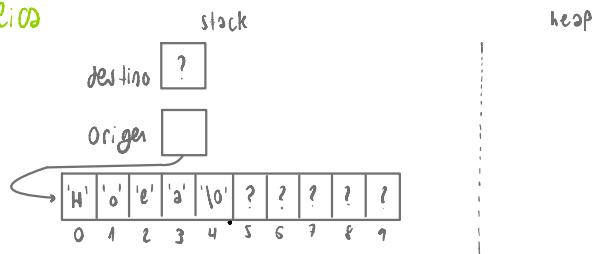
- `duplicad.c`: copia cadenas estáticas a dinámicas
- `realloc.c`: lee una cadena carácter a carácter usando realloc

Ej: `char* duplica (char* origen) {`

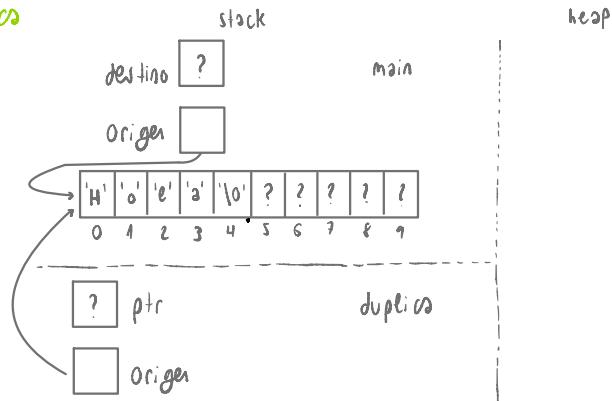
① `char* ptr;`
② `if ((ptr = (char*) malloc ((strlen (origen) + 1) * sizeof (char))) == NULL) {`
 `printf ("Error: no pudo asignarse memoria \n");`
 `ptr = NULL ; }`
③ `else strcpy (ptr, origen);`
④ `return (ptr)`

```
int main () {
    char origen [10];
    char* destino;
    printf ("Introducir cadena origen:\n");
    ① gets (origen);
    destino = duplica (origen);
    printf ("Origen: <%s> Longitud: %d\n", origen, strlen (origen));
    ⑤ printf ("Destino: <%s> Longitud: %d\n", destino, strlen (destino));
    free (destino);
    return (0); }
```

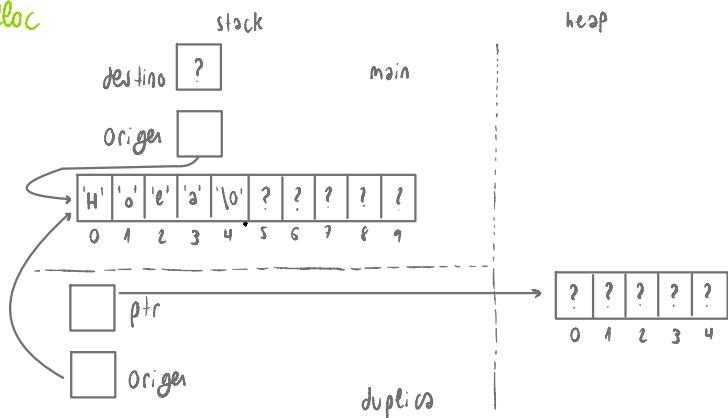
1. Antes de llamar `duplica`



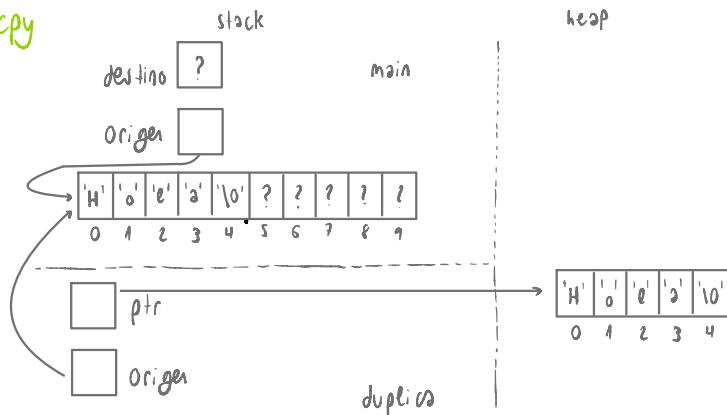
2. Los mads a duplicos



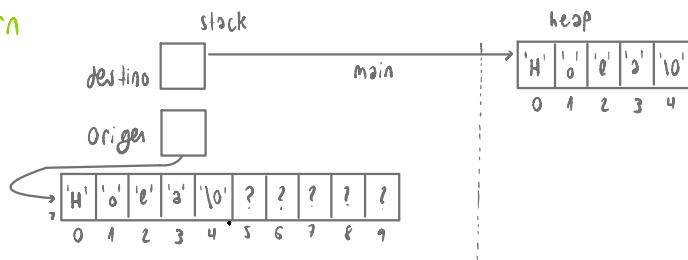
3. Despues de malloc



4. Despues de strcpy



5. Despues de return



Ej:

```
#include <stdio.h>
#include <stdlib.h>
int main () {
    char c, *cad = NULL;
    int nElee = 0;
    while ((c = getchar ()) != '\n') { → Lee las cadenas con realloc
        cad = (char *) realloc (cad, (nElee + 1) * sizeof (char));
        cad [nElee] = c;
        nElee++;
    }
    cad = (char *) realloc (cad, (nElee + 1) * sizeof (char)); → Añade \0
    cad [nElee] = '\0';
    printf ("\nLa cadena leida es <%s>", cad);
    return 0;
}
```

Vectores de estructuras

Ej:

```
struct dato {
    int n;
};

struct dato* reservaVectorStr (int nElee) {
    struct dato* ptr;
    if ((ptr = (struct dato*) malloc (nElee * sizeof (struct dato))) == NULL) {
        printf ("\nError en la reserva de memoria");
        exit (-1);
    }
    return (ptr);
}
```

Ej:

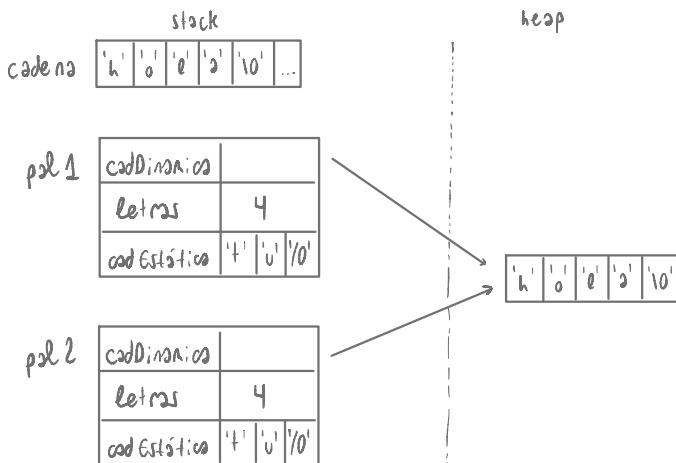
```

void reservaVectorStrReferencia (struct dato** ptr, int nEle) {
    if ((*ptr = (struct dato*) malloc (nEle * sizeof (struct dato))) == NULL) {
        printf ("\n Error en la reserva de memoria");
        exit (-1);
    }
}

void liberaVectorStr (struct dato* ptr) {
    free (ptr);
}

```

Estructuras y cadenas



Ej:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct palabra {
    char* cadDinamica;
    int letras;
    char cadEstatica[3];
};

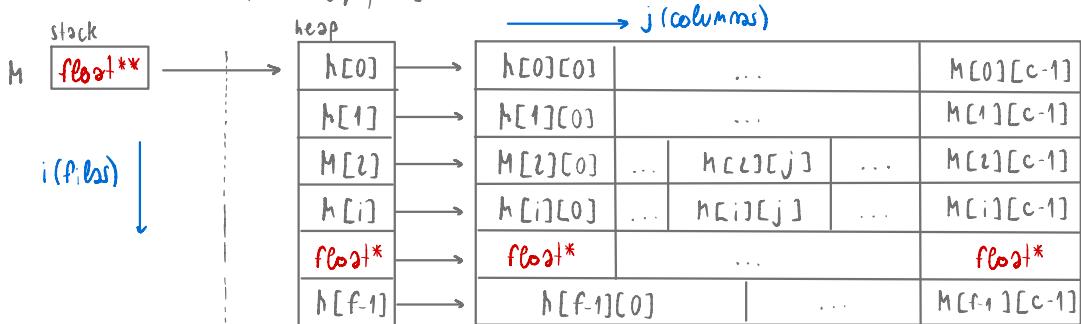
int main() {
    struct palabra pal1, pal2;
    char cadena[256];
    printf("Introduce una cadena: ");
    scanf("%s", cadena);
    pal1.cadDinamica = (char*) malloc(sizeof(char)*(strlen(cadena)+1));
    strcpy(pal1.cadDinamica, cadena);
    pal1.letras = strlen(pal1.cadDinamica);
    strcpy(pal1.cadEstatica, "tu");
    pal2 = pal1;
    printf("Direccion memoria pal1.cadDinamica: %p\n" &(pal1.cadDinamico));
    printf("Direccion memoria pal2.cadDinamica: %p\n" &(pal2.cadDinamico));
    printf("Direccion de inicio de cadena <%s> almacenado en pal1.cadDinamica: %p\n" pal1.cadDinamica, pal1.cadDinamico);
    printf("Direccion de inicio de cadena <%s> almacenado en pal2.cadDinamico: %p\n" pal2.cadDinamica, pal2.cadDinamico);
    printf("Direccion de inicio de cadena <%s> almacenado en pal1.cadEstatica: %p\n" pal1.cadEstatica, pal1.cadEstatico);
    printf("Direccion de inicio de cadena <%s> almacenado en pal2.cadEstatica: %p\n" pal2.cadEstatica, pal2.cadEstatico);
```

Matrices dinámicas

- Teneremos **dos opciones**

- Reservar y liberar la matriz por filas.

Ej: float** reservarMatrizDinamicaPorFilas (int nFil, int nCol) {
 float** matriz;
 int i;
 matriz = (float**) malloc (nFil * sizeof (float**));
 for (i=0; i<nFil; i++) {
 matriz [i] = (float*) malloc (nCol * sizeof (float)); }
 return (matriz); }



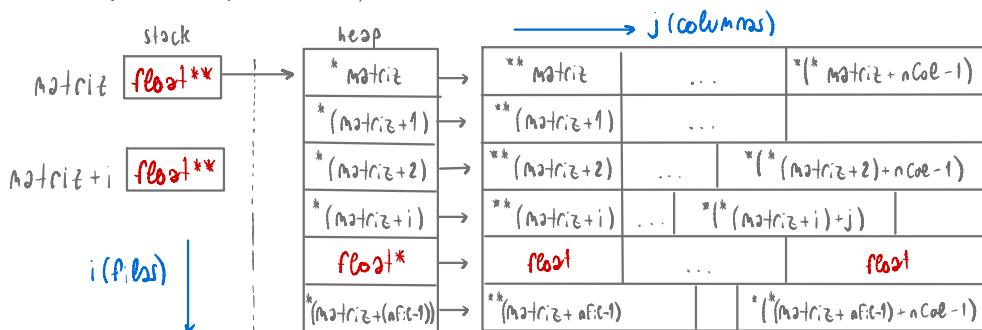
Ej: void liberarMatrizDinamicaPorFilas (float** matriz, int nFil) {
 int i;
 for (i=0; i<nFil; i++) {
 free (matriz [i]); }
 free (matriz); }

Ej: void liberarMatrizDimensionesPorFilas (float*** matriz, int nFil) {

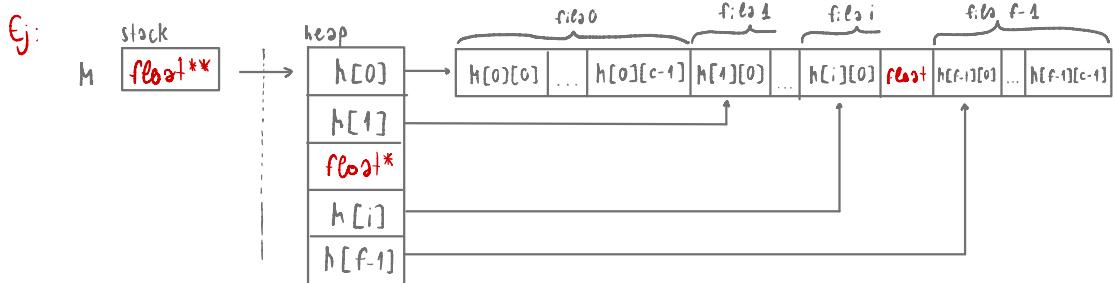
```

int i;
for (i=0; i<nFil; i++) {
    free ((*matriz)[i]);
}
free (*matriz);
(*matriz) = NULL;
}

```



- Reservar y liberar la matriz en un solo bloque

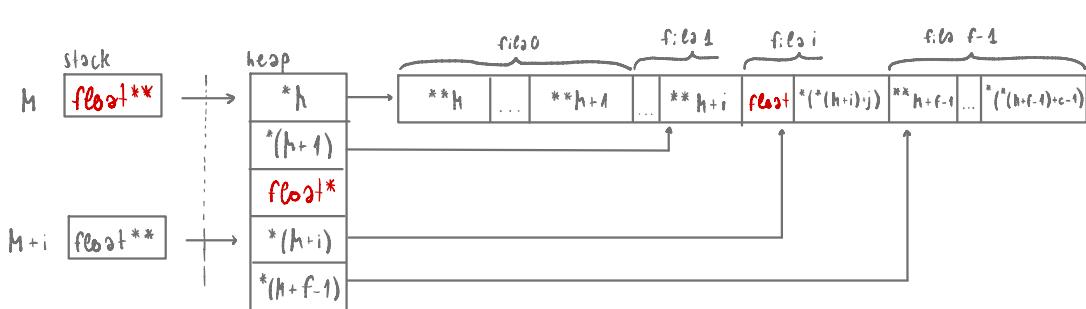


```

Ej: float** reservaMatrizDinamicoUnSoloBloque (int nFil, int nCol) {
    float** matriz;
    int i;
    matriz = (float**) malloc (nFil * sizeof (float)*));
    Matriz [0] = (float*) malloc (nFil * nCol * sizeof (float));
    for (i=1; i < nFil; i++)
        matriz[i] = matriz[i-1] + nCol;
    return (matriz);
}

void liberarMatrizDinamicoUnSoloBloque (float** matriz) {
    free (matriz[0]);
    free (matriz);
}

```



Punteros y matrices bidimensionales

- Podemos utilizar notación de punteros o de corchetes

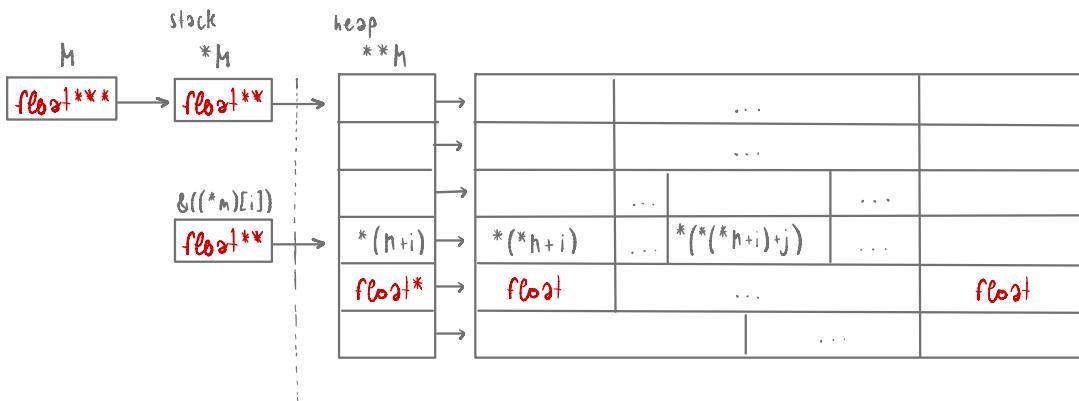
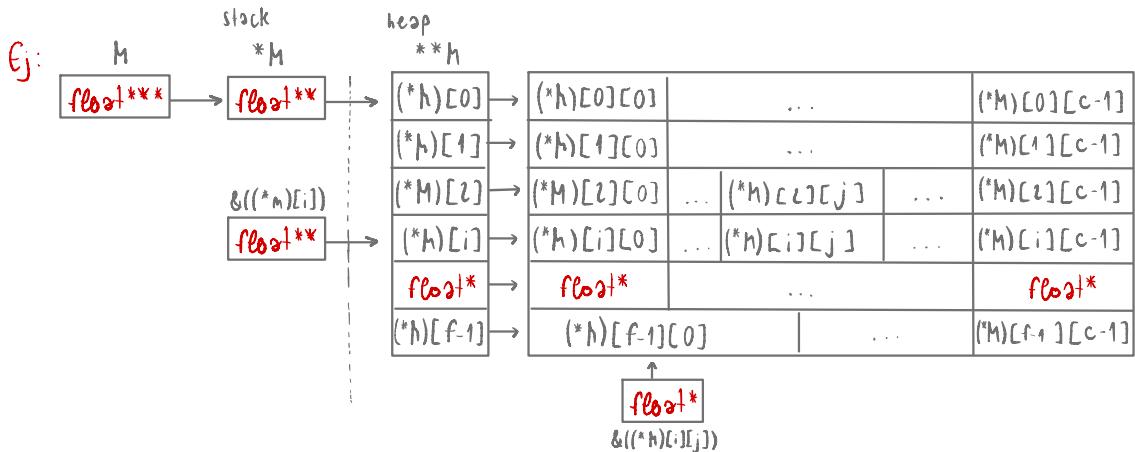
Matriz bidimensional reservada dinámicamente

Ej:

<code>*h</code> : puntero a las primera filas <code>*h+1</code> : puntero a las segundas filas <code>**h</code> : valor <code>h[0][0]</code> <code>**h+1</code> : valor <code>h[1][0]</code> <code>*(*(h+1)+2)</code> : valor <code>h[1][2]</code>	$\left. \begin{array}{l} M[i][j] \approx *(*(h+i)+j) \\ M[i][j][k] \approx *(*(*(h+i)+j)+k) \end{array} \right\}$
--	---

Paso de matrices por referencia

- A veces, es necesario pasar el puntero a los primeros filas de la matriz por referencia
 - When queremos cambiar la dirección de comienzo de la matriz
 - Al crearla (si los pasamos como parámetro)
 - Para que apunte a NULL



Ej: void reservaMatrizDinamicaPorFilasRef (float*** matriz, int nfil, int ncol){

```
int i;
*matriz = (float**) malloc(nfil * sizeof(float *));
for (i=0; i<nfil; i++) {
    (*matriz)[i] = (float*) malloc (ncol * sizeof (float)); }
```

Matrices de estructuras

Ej: struct dato** reservaMatrizStr (int nfil, int ncol) {

```
struct dato** ptr;
int i;
if ((*ptr = (struct dato**) malloc (nfil * sizeof (struct dato*))) == NULL){
    printf ("\nError en la reserva de memoria (1)");
    exit (-1); }
for (i=0; i<nfil; i++){
    if ((*ptr[i] = (struct dato*) malloc (ncol * sizeof (struct dato))) == NULL){
        printf ("\nError en la reserva de memoria (2)");
        exit (-1); } }
return (ptr); }
```

Ej: void reservaMatrizStrReferencia (struct dato*** ptr, int nfil, int ncol) {

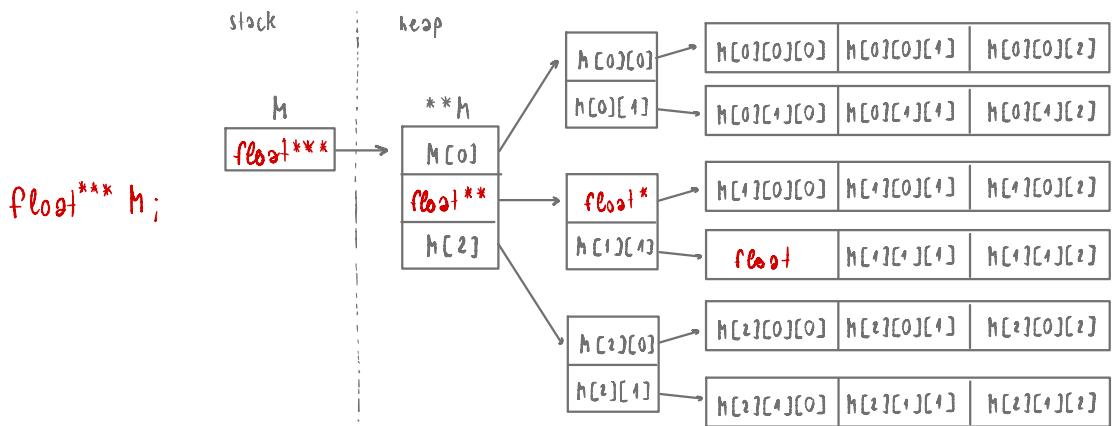
```
int i;
if ((*ptr = (struct dato**) malloc (nfil * sizeof (struct dato*))) == NULL){
    printf ("\nError en la reserva de memoria (1)");
    exit (-1); }
for (i=0; i<nfil; i++){
    if ((*ptr[i] = (struct dato*) malloc (ncol * sizeof (struct dato))) == NULL){
        printf ("\nError en la reserva de memoria (2)");
        exit (-1); } } }
```

Ej: void LiberaMatrizStr (struct dato** ptr, int nFil) {
 int i;
 for (i=0; i<nFil; i++) {
 free (ptr[i]); }
 free (ptr); }

Matrices de n-dimensiones

Ej: float*** reservaMatrizTridimensional (int nFil, int nCol, int nAlt) {
 float*** matriz;
 int i, j;

 matriz = (float***) malloc (nFil * sizeof (float**));
 for (i=0; i<nFil; i++) {
 matriz[i] = (float**) malloc (nCol * sizeof (float*));
 for (j=0; j<nCol; j++) {
 matriz[i][j] = (float*) malloc (nAlt * sizeof (float)); } }
 return [matriz] }

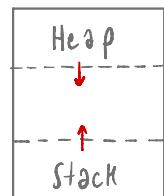


Ej: void liberarMatrizTridimensional (float*** matriz, int nFil, int nCol) {
int i, j;
for (i=0; i<nFil; i++) {
 for (j=0; j<nCol; j++) {
 free (matriz [i][j]); }
 free (matriz [i]); }
free [matriz] }

Organización de la memoria en tiempo de ejecución

Administración de la memoria en tiempo de ejecución

- Al ejecutar el programa la memoria se divide (de forma lógica) en:
 - Código ejecutable
 - Código máquina, constantes y literales (lectura)
 - Memoria estática (permanente durante la ejecución)
 - Datos
 - Variábles globales y estáticas (lectura/escritura)
 - Pila (stack)
 - Cuando se produce una llamada se interrumpe la ejecución y se guarda en pila
 - Variábles locales y resultados intermedios
 - Información sobre el estado de la máquina
 - Montículo (heap)
 - Guarda el resto de información generada durante la ejecución
 - Bloques de memoria direccional (por punteros)



Estrategias de asignación de memoria

- Asignación estática: Dispone la memoria para todos los datos durante la compilación
- Asignación por medio de una pila: Trata la memoria de ejecución como una pila
- Asignación por medio de un montículo: Desasigna la memoria conforme se necesita (en ejecución)

Asignación estática de memoria

- Tiempo de compilación: tamaño de memoria que va a ser reservada
- Los objetos están vigentes desde la ejecución hasta que termina el programa
- La memoria se reserva de forma consecutiva
- La dirección de memoria de los objetos se puede conocer en tiempo de compilación
 - Si la primera variable está en dirección x y ocupa n_1 , la segunda estará en $x+n_1$, y la tercera en $x+n_1+n_2\dots$
- Se usan asignaciones estáticas para datos globales
- Ventajas
 - La implementación de la estrategia es simple
 - Conocer la dirección de las variables al generar el código objeto mejora el tiempo de ejecución
- Inconvenientes
 - El tamaño del objeto debe conocerse al compilar
 - Al no asignar memoria en el tiempo de ejecución
 - No permite procedimientos recursivos y que utilicen las mismas posiciones de memoria
 - Las estructuras de datos no se pueden crear dinámicamente (listas, árboles...)

Asignación mediante pila (stack)

- Registro de activación (stack frame)
 - Información que se transmite a una función al llamarla
 - Su magnitud se puede determinar por tiempo de compilación
- Función recursiva
 - El número de llamadas recursivas (\rightarrow nº de registros de activación) no se conoce en tiempo de compilación
 - Para implementar la recursividad se usa una estructura de pila (stack)
 - Al llamar se coloca un nuevo registro de activación en la pila
 - Al finalizar la función se extrae
 - En la ejecución pueden invocarse otros bloques

Asignación mediante montículo

- La estrategia anterior no es suficiente para manejar el tráfico de datos (su magnitud puede cambiar durante ejecución y no es conocida en tiempo de compilación (sino de ejecución))
- Para gestionarlos se reserva un gran bloque contiguo (montículo, montón o heap)
- Cuando se realiza la petición adicional de memoria un controlador localiza el espacio heap en tiempo de ejecución
- Cuando un espacio ha dejado de utilizarse, ha de ser liberada. Hay tres técnicas:
 - No liberar: cuando se alquila la memoria se para el programa
 - Liberación explícita: se deja al programador la responsabilidad de liberar memoria
 - Liberación implícita: recuperación automática del espacio heap (recolección de basura)
- Estrategias de asignación implícita
 - Desocupación sobre marcha (free-as-you-go)
 - Desbloquear los bloques del montón cuando no estén referenciados por punteros
 - Se necesita que el sistema lleve internamente un contador del número de punteros que se dirigen a los bloques del montón
 - Método de Marcar y Barrer (mark and sweep)
 - Activar un procedimiento especial al detectar ciertas condiciones en el uso de memoria

Documentación

Introducción

- Un programa es escrito por un programador y será consultado por muchas otras personas. Puede tener errores que parecen de percibidos y posteriormente se corrijan por otro programador (mantenimiento)

Documentación

- Con el tiempo se olvidan y se pierden muchos detalles relacionados con un programa, la documentación consiste en describir lo que hace y como
 - Interna: aportada en el propio código
 - Externa: fuera del código

Documentación interna. Comentarios

- Se pueden añadir comentarios en cualquier lenguaje (/*/, //!, <!---->)
- Los comentarios suelen ir en un color distinto
- Si un algún algoritmo en pseudo-código incorpora comentarios, deben de trasladarse al código del programa
- No es conveniente que los comentarios sean excesivos
- Cabecera de funciones o procedimientos
 - Nombre, propósito y autor
 - Parámetros de entrada y salida
 - Procedimientos y funciones a los que llama o por los que es llamado
 - Códigos de error
 - Fecha de creación y modificaciones

Ej: /*****

Nombre: BuscarPorNombre.

Tipo: int.

Objetivo: Visualiza todos los registros que tengan un nombre dado por el usuario.

Parámetros de entrada:

- char *fichero: Nombre del fichero.

- char *auxNombre: Nombre de los registros a visualizar.

Precondiciones: El fichero ha de existir.

Valor devuelto: 0 si no se encuentra ningún registro y 1 si se encuentra alguno.

Funciones a las que llama:

- escribir Datos Personales.

Fecha de creación: 7-01-03.

Autor:

/*****

int buscarPorNombre (char *fichero, char *auxNombre) {

} //Cuerpo de la función

- Declaraciones de variables o tipos compuestos

- Asignar un nombre adecuado a la función (autodocumentación). Si no es posible añadir comentarios

- Esquemas condicionales e iterativos

- Clasificar su propósito y condiciones.

- Llamadas a subprogramas o función.

- Indicar la función y efecto sobre las variables usadas

Documentación interna. Presentación

- El sangrado o identificación de párrafos
 - Utilizar distintas niveles de sangrado (esquemas jerárquicos)
 - Facilita la comprensión de código
 - Si existen muchos niveles de anidamiento se reducen los sangrados
- Hacer corresponder las líneas de código con una línea física
- Espacios en blanco
- Líneas en blanco para separar bloques
- Agrupamiento de sentencias de E/S
- Correspondencia entre orden de ubicación de los bloques y orden de ejecución

Ej: Nombre de las variables

```
get a b c  
if a < 24 and b < 60 and c < 60  
return true  
else  
return false
```

Ej: get horas minutos segundos

```
if horas < 24 and minutos < 60 and segundos < 60  
return true  
else  
return false
```

Ej: Indentación

```
if(horas < 24 && minutos < 60 && segundos < 60){  
    return true ; }  
else { return false ; }
```

Ej: if(horas < 24 && minutos < 60 && segundos < 60){
 return true ; }
else {
 return false ; }

Ej: Espaciado

```
int cuenta;  
for (cuenta=0; cuenta < 10; cuenta++) { printf ("%d",  
    cuenta*cuenta + cuenta); }
```

Ej: int cuenta;

```
for (cuenta=0; cuenta < 10; cuenta++) {  
    printf ("%d", cuenta*cuenta + cuenta); }
```

Documentación externa

- Manual de usuario
 - Documento destinado a facilitar el uso del programa a los usuarios
 - Debe reflejar una explicación detallada
 - Debe contener un ejemplo práctico de uso del programa
- Manual del operador
 - Destinado al operador informático (Contiene las especificaciones que han de cumplir)
- Manual de mantenimiento
 - Destinado al programador que se dediquen al mantenimiento
- Especificaciones del programa
 - Elaborado por el programador, refleja las especificaciones del cliente
- Lista de datos de prueba y resultados
 - Refleja pruebas realizadas al programa
- Historia de desarrollo del programa y modificaciones posteriores
 - Realizado por el que implementa la aplicación y los encargados de mantenimiento
- Diseño descendente con detalle de módulos y submódulos
 - Siguiendo metodología específica
- Versiones del programa
 - Reflejando diferencias, ventajas e inconvenientes entre las posibles versiones

Principios del programador

- Principio del Carácter Personal
 - Escribe el código en forma que refleje lo mejor de tu carácter personal
- Principio de la Estética
 - Esfuerzate por conseguir la belleza y elegancia en tu trabajo
- Principio de la Claridad
 - Dale el mismo valor a la claridad y a la corrección.
- Principio de la Distribución
 - Usa una distribución visual de tu código para comunicar al lector humano
- Principio de lo Explícito
 - Fomentar lo explícito y lo implícito
- Principio de Código Auto-Documentado
 - La documentación más fiable para el software es el propio código
- Principio de los Comentarios
 - Comenta mediante frases completas resumir y comunicar intención
- Principio de las Suposiciones
 - Da los pasos necesarios para comprobar, documentar y prestar atención a las suposiciones hechas.
- Principio de las Interfaz con el Usuario
 - Nunca hagas que el usuario se sienta estúpido
- Principio de Volver Atrás
 - El momento de escribir un buen código es el preciso momento en el que lo estás escribiendo
- Principio de el Tiempo y el Dinero de Otras
 - No gastar el tiempo ni el dinero de otras personas

Herramientas para documentar código

- Java (Javadoc)
- C (Doxygen)
- .NET (Ndoc)
- A partir de un diseño (Rational Rose)
- Casi cualquier lenguaje (Doc-O-Main)

¿Qué es Doxygen?

- Doxygen es un programa generador de documentación de códigos fuente
- Soporta C, C++, Java, Python, IDL, Fortran, VHDL, PHP...
- Bajo licencia GNU
- Formatos de salida
 - HTML
 - LaTeX
 - PDF
 - Página de manual
- Instalación
 - Bajar archivos de www.doxygen.org
 - Linux, Windows...
 - Configurar en PATH para encontrar ejecutables
 - Operación mediante consola gráfica

Utilizar Doxygen

1. Generar fichero de configuración doxygen -g ficheroConfiguracion
2. Editar fichero con las preferencias
3. Generar documentación a través del fichero de configuración
doxygen ficheroConfiguracion

Fichero de configuración

- PROJECT_NAME = Nombre del proyecto
- OUTPUT_LANGUAGE = Spanish (o English)
- OPTIMIZE_OUTPUT_FOR_C = YES
- SOURCE_BROWSER = YES
- GENERATE_HTML = YES

¿Qué se comenta en doxygen?

- Información sobre el fichero - programa
 - Funciones
 - Estructuras definidas
 - Defines
 - Variables globales
- ¿Qué información incluir en la documentación?
 - Descripción breve y detallada
 - Autor, fecha
 - Información específica sobre lo documentado
 - Parámetro y valor devuelto
 - Campos para las estructuras

Comentarios en doxygen

- Comentario en C
 - Una línea // Texto
 - Varias líneas /* líneas 1...
...línea 2 */
- Comentarios en Doxygen
 - Se incluye el código en bloque de comentarios
 - Doxygen permite dos tipos de comentarios (C o Qt)

Comentarios Doxygen	Estilo C	Estilo OT
Varias líneas (d. detallada)	/** Texto */	/*! Texto */
Una línea (d. breve)	/// Texto	/// Texto

Comentarios en el código

- Pueden ir antes o después del bloque de código
- Después del bloque de código debe de llevar ↗

Comentarios Doxygen	Antes	Después
Varias líneas (d. detallada)	/** Suma enteras */ int suma (int a, int b);	/**< Suma enteras */ int suma (int a, int b);
Una línea (d. breve)	/// Suma enteras int suma (int a, int b);	///< Suma enteras int suma (int a, int b);

Descripción breve y detallada

- Sin dejar una linea en blanco

```
///< Descripción breve
/**< Descripción detallada */
```
- Utilizando el comando brief

```
/** @brief Descripcion breve
La descripción breve llega hasta las líneas en blanco
La descripción detallada llega hasta las líneas en blanco
*/
```

Comandos. Modificar aspecto

Comando	Función
@n	Salto de línea
@b	Pone en negrita la siguiente palabra
@em	Pone en cursiva la siguiente palabra
@p	Pone en estilo <i>curier</i> la siguiente palabra
\@, \\, \\$, \&, \#, \%, <, >	Escribe los símbolos @, \, \$, &, #, %, <, >
@li <descripción de un ítem>	Genera una lista de argumentos. Cada elemento de la lista empieza con @li
@f\$	Incluir una fórmula de texto (formato LaTex)
@f[Comienzo de una fórmula larga (formato LaTex)
@f]	Fin de una fórmula larga (formato LaTex)

Comandos. Estructurales

- Las comandos pueden estar precedidas de \ o @
 - En prácticas las ejemplos están montados con \
 - En clase las ejemplos están montados con @

Comandos estructurales

- @fn <función>
- @file <fichero>
- @struct <estructura>
- @var <variable>
- @def <define>
- @typedef <typedef>

Ej: `/**
 * @struct punto
 * @brief Definicion breve de estructura tipo punto`

Definicion detallada de la estructura
*/

```
struct punto {  
    float x; /**< Coordenada x del punto (D. larga) */  
    float y; /**< Coordenada y del punto (D. breve)  
};
```

Ej: `/**
 * @typedef t-punto
 * @brief Definicion breve del typedef
 * Definicion detallada del typedef
 */`

```
typedef struct punto t-punto;
```

Ej: `/**
 * @def PI
 * @brief Definicion breve del numero PI
 * Definicion detallada del numero PI
 */`

```
#define PI 3.14
```

Comentarios ficheros (para micros)

- Dentro de los comentarios de doxygen se incluyen comentarios
- Cabecera del fichero
 - @file <comentario sobre el fichero>
 - @author <comentario sobre el autor>
 - @date <fecha>
 - @version <version del archivo>

Ej: /**

```
@file myfile.h
@brief descripción breve sobre fichero
@author Eva Gibaja
@date 13-04-2012
@version 1.0
```

Este fichero contiene las funciones necesarias para trabajar con imágenes:

```
@li Cargar imagen
@li Grabar imagen
@li Espejo horizontal
*/
```

Comentarios funciones (para micras)

• Comandos para documentar funciones

- @fn <Nombre o prototipo>
- @param <Comentario de un parámetro>
- @return <Comentario sobre valor devuelto>
- @author <Comentario sobre el author>
- @date <fecha>
- @post <Postcondicion>
- @pre <Precondicion>

Ej:

```
/**  
 * @fn Nombre o prototipo  
 * @brief Descripcion breve de la funcion
```

Descripcion detallada de la funcion

```
@param a Descripcion del parametro a  
@param b Descripcion del parametro b  
@return Informacion sobre el valor devuelto  
*/
```

Ej: /* Nombre : restaCuadrados

Tipo: entero (int)

Objetivo: calcula la diferencia de los cuadrados de dos números ($n1^2 - n2^2$)

Parámetros entrada:

int n1: primer numero

int n2: segundo numero

Precondiciones: Ninguna

Valor devuelto: $n1^2 - n2^2$

Utiliza: cuadrado

Autor: María Luque

Fecha: 20-01-2008

*/

int restaCuadrados (int n1, int n2);

Ej: /**

@fn restaCuadrados (int n1, int n2)

@brief Calcula la diferencia de los cuadrados

@param n1 Primer numero

@param n2 Segundo numero

@pre Ninguna

@return @f\$ $n_1^2 - n_2^2$ \$

@author María Luque

@date 20-01-2008

Esta función calcula la diferencia entre los cuadrados de los referenciados

*/

int restaCuadrados (int n1, int n2);

Ej:

```
/**  
 * @file 3.1.c  
 * @brief Este archivo es un ejemplo  
 * @author Eva Gibaja  
 * @date 26/08/2011  
 #include "3.2.h"  
/**  
 * @fn void main()  
 * @brief funcion main()  
 * @return nada  
 */  
void main(){  
    int i;  
    boolean c;  
    c=(char)i;  
    suma((int)c,5);  
    return;  
}  
/**  
 * @mainpage Introduccion
```

En esta pagina podemos incluir una descripción general
*/

Recursividad

Problemas recursivos

- Un problema T es recursivo cuando T define términos de versiones más pequeñas de sí mismo
- Para que una definición recursiva este identificada hace falta un caso base que no se calcule con casos anteriores

Ej: $n! = n * (n-1)!$
 $0! = 1$ (caso base)

Ej: $x^n = x * x^{n-1}$
 $x^0 = 1$ (caso base)

Ej: Calcula $n!$ con $n=3$

$$3! = 3 \cdot 2!$$

$$2! = 2 \cdot 1!$$

$$1! = 1 \cdot 0! \rightarrow 1! = 1 \cdot 1 = 1$$

$$2! = 2 \cdot 1 = 2$$

$$3! = 3 \cdot 2 = 6$$

- Existen dos procesos fundamentales:
 - Llegada al caso base
 - Vuelta hasta el caso original

Algoritmos recursivos

- Un algoritmo es recursivo si realiza al menos una llamada a si mismo
- Un módulo recursivo tiene dos partes:
 - Condición de parada (caso base): Establece el momento de terminación del procedimiento recursivo
 - Cuerpo del módulo: Colabora a realizar para resolver problemas (incluir otras llamadas recursivas)
- Tipos de recursividad:
 - Directa: contiene una llamada a si mismo
 - Simple o lineal: El nombre aparece una sola vez en el cuerpo del algoritmo
 - Doble, múltiple o no lineal: El nombre aparece más de una vez en el cuerpo del algoritmo
 - Indirecta: El algoritmo P contiene una llamada al algoritmo Q y este una llamada directa o indirecta al P

Ej: int factorial (int n){
 int resultado;
 if (n == 0) {
 resultado = 1; *Caso base*
 }
 else {
 resultado = n * factorial (n-1); *Cuerpo*
 }
 return resultado
}

Ejecución de un módulo recursivo

- La memoria del programa en tiempo de ejecución se divide en:
 - Código: instrucciones del código máquina
 - Datos: variables estáticas y globales
 - Punteros (heap): variables dinámicas
 - Pila (stack): registro de activación del módulo (variables locales y parámetros)
- Llamada a una función
 1. Reserva del espacio en la pila para parámetros y variables locales
 2. Se guarda la dirección de las líneas de código que llama la función en la pila
 3. Se almacenan parámetros y valores en la pila
 4. Se libera la memoria de la pila
- Llamada a una función recursiva
 1. Se ejecuta como una función normal hasta la llamada recursiva
 2. Se crean en la pila nuevos parámetros y variables locales
 3. La nueva función se ejecuta
 4. Se crean más copias hasta llegar a los casos base, donde se resuelve el valor
 5. Se sale liberando la memoria hasta la primera llamada

Ej:

```
int potencia (int base, int expo) {
    if (expo == 0) {
        return 1;
    }
    else {
        return base * potencia (base, expo - 1);
    }
}
```

Ej: Suma recursiva

$$\text{suma}(a, b) = a + \text{suma}(a, b-1)$$

$$\text{suma}(a, 0) = a$$

```
int suma ( int a, int b ) {  
    if (b == 0) {  
        return a;  
    }  
    else {  
        return 1 + suma (a, b-1);  
    }  
}
```

Ej: Producto recursivo

$$\text{producto}(a, b) = a + \text{producto}(a, b-1)$$

$$\text{producto}(a, 0) = 0$$

```
int producto ( int a, int b ) {  
    if (b == 0) {  
        return 0;  
    }  
    else {  
        return a + producto (a, b-1)  
    }  
}
```

Ej: int sumaDigitos (int num) {

```
if (num < 10) {
    return num;
}
else {
    return (num % 10 + sumaDigitos (num / 10));
}
```

Ej: ¿Es potencia de una base?

```
int esPotencia (int num, int base) {
    if (num == 1) {
        return 1;
    }
    else if (num % base != 0) {
        return 0;
    }
    else {
        return esPotencia (num/base, base);
    }
}
```

Ej: Sucesión de Fibonacci

```
int fibonacci (int n){  
    if ((n == 1) || (n == 2)) {  
        return (1);  
    }  
    else {  
        return (fibonacci (n-1)+fibonacci (n-2)); Recursión doble  
    }  
}
```

Ej: Fibonacci iterativo (más eficiente)

```
int fibonacciIterativo (int n){  
    int ant1 = 1; Anterior  
    int ant2 = 1; Anterior del anterior  
    int actual = 0;  
    int i;  
    if ((n == 1) || (n == 2)) {  
        actual = 1;  
    }  
    else {  
        for (i = 3; i <= n; i++) {  
            actual = ant1 + ant2;  
            ant2 = ant1; } actualizar ant1 y ant2  
            ant1 = actual;  
        }  
    }  
    return (actual);  
}
```

Ej: Suma recursiva de los elementos de un vector

```
int sumaV (int *v, int n) {  
    if (n == 0) {  
        return v[0];  
    }  
    else {  
        return v[n] + sumaV (v, n - 1);  
    }  
}
```

Ej: Escribir un vector

```
void escribirVector (int *v, int posicion) {  
    if (posicion >= 0) {  
        escribirVector (v, posicion - 1);  
        printf ("\n\tV[%i] = %i", posicion, v[posicion]);  
    }  
}
```

Ej: Invertir vector

```
void escribirVectorInvertido (int *v, int posicion) {  
    if (posicion >= 0) {  
        printf ("\n\tVinvertido[%i] = %i", posicion, v[posicion]);  
        escribirVectorInvertido (v, posicion - 1);  
    }  
}
```

Ej: Busqueda lineal recursiva

```
int busquedaLineal (int *v, int posicion, int elemento) {
    if (posicion < 0) {
        return (0); primer caso base
    }
    else {
        if (v[posicion] == elemento) {
            return (1); segundo caso base
        }
        else {
            return (busquedaLineal (v, posicion-1, elemento));
        }
    }
}
```

Ej: elemento mayor de un vector

```
int mayor1 (int *v, int n) {
    int aux;
    if (n==0) {
        return v[0];
    }
    else {
        aux = mayor1 (v, n-1);
        if (v[n] > aux) {
            return v[n];
        }
        else {
            return aux;
        }
    }
}
```

Ej: Los dos elementos mayores de un vector

```
void dosMayores(int *v, int posicion, int *a, int *b){  
    if (posicion == 1) {  
        if (v[0] > v[1]) {  
            *a = v[0];  
            *b = v[1];  
        }  
        else {  
            *a = v[1];  
            *b = v[0];  
        }  
    }  
    else {  
        dosMayores(v, posicion - 1, a, b);  
        if (v[posicion] > *a) { Caso 1: -v[pos] --a --b  
            *b = *a;  
            *a = v[posicion];  
        }  
        else if (v[posicion] > *b) { Caso 2: a-- v[pos] --b  
            *b = v[posicion];  
        }  
    }  
}
```

Ej: Invertir un vector

```
void invertirVector (int *v, int izda, int dcha) {  
    int aux;  
    if (izda <= dcha) {  
        aux = v[izda];  
        v[izda] = v[dcha];  
        v[dcha] = aux;  
        invertirVector (v, izda+1, dcha+1);  
    }  
}
```

Ej: Determina si es palíndromo

```
int palindromo (char *cad, int izda, int dcha) {  
    if (izda > dcha) {  
        return (1);  
    }  
    else {  
        if (cad[izda] == cad[dcha]) {  
            return (palindromo (cad, izda+1, dcha-1));  
        }  
        else {  
            return (0);  
        }  
    }  
}
```

Ej: Pasar decimal a binario

```
int decimalTo2 ( int numero, char *res ) {  
    if (numero == 1) {  
        sprintf(res, "%d", 1);  
    }  
    else {  
        if (numero == 0) {  
            sprintf(res, "%d", 0);  
        }  
        else {  
            decimalTo2 ( numero / 2, res );  
            sprintf(res, "%5%d", res, numero % 2 );  
        }  
    }  
}
```

Paso de parámetros en recursión

- Hay que tener avisoado con el paso de parámetros en las módulos recursivos, un paso indeizado ocasionara errores
- Si x se pasa por referencia, cualquier modificación en una llamada modificara el resto

Conclusión

- Evitar recursividad si se puede resolver fácilmente de forma iterativa
 - Implementación más rápida
 - Algoritmo recursivo necesita memoria para almacenar llamadas
 - Las llamadas de subprogramas requiere tiempo de ejecución adicional
 - La iteración evita evaluar dos veces el mismo valor
- Utilizar recursividad si hay problemas que se resuelven recursivamente de forma inmediata y la iterativa más compleja

Ficheros de texto