

Programación I

Punteros

Iván Cantador

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Contenidos

1

- Definición y manejo de punteros
- Punteros y arrays
- Punteros a estructuras
- Punteros como argumentos de funciones
- Punteros y memoria dinámica

Contenidos

2

- Definición y manejo de punteros
- Punteros y arrays
- Punteros a estructuras
- Punteros como argumentos de funciones
- Punteros y memoria dinámica

Definición y manejo de punteros (I)

3

- Un **puntero** es una variable donde se almacena la dirección de memoria de otra variable de cierto tipo

```
int n = 5; /* Asigna el valor 5 a la posición de memoria
            identificada como n y que tiene espacio para
            guardar un número entero */
```

```
int *p = NULL;
p = &n; /* Guarda en p la dirección de n */
        /* "p apunta a n" */
```

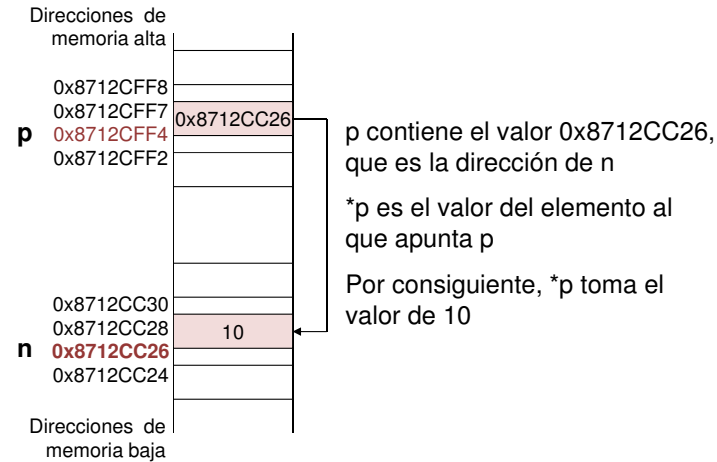
```
printf("%x\n", &n); /* Imprime la dirección de n */
printf("%x\n", p);  /* Imprime la dirección de n */
```

```
*p = 10; /* Asigna el valor 10 a la posición de memoria
          apuntada por p (es decir, a la de n) */
```



Definición y manejo de punteros (II)

4



Definición y manejo de punteros (III)

5

• Declaración

```
<tipo_dato> *<nombre_variable>;
```

```
int *pi = NULL;    /* Puntero a int */
long *pl = NULL;   /* Puntero a long */
float *pf = NULL;  /* Puntero a float */
double *pd = NULL; /* Puntero a double */
char *pc = NULL;   /* Puntero a char (cadena de caracteres) */

void *pv = NULL;   /* Puntero a void */

int **ppi = NULL;  /* Puntero a puntero a entero (matriz) */

Libro *ps = NULL;  /* Puntero a tipo definido (estructura) */
```

Definición y manejo de punteros (IV)

6

• Asignación

```
p = &<nombre_variable>;
```

```
int n;
int *pi = NULL;

pi = &n; /* p apunta a n */

*pi = 10; /* Equivalente a n = 10 */

/* CUIDADO CON LOS "PUNTEROS LOCOS" */
char c = 'A';
char *pc;

*pc = c; /* ¡DESASTRE! (¿a qué posición apuntaba pc?) */
```

2 usos diferentes de '*'
(aparte del operador aritmético de multiplicación)

Definición y manejo de punteros (V)

7

• Punteros nulos: aquellos que apuntan a **NULL** (que es un valor, 0, definido en **stdio.h**)

```
p = NULL;
```

• Comprobación de puntero nulo

```
if( p ) ... /* Equivale a if( p != NULL ) ... */
if( !p ) ... /* Equivale a if( p == NULL ) ... */
```

• Verificación de tipos

- En C, las variables puntero han de direccionar realmente a variables del mismo tipo de dato ligado a los punteros en sus declaraciones

```
int i;
long l;
long *p = NULL;      /* Puntero a long */

p = &l;               /* Correcto */
p = &i;               /* ¡Incorrecto! */
```

• Punteros a tipo no definido

- En C, los punteros a **void** pueden apuntar a variables de cualquier tipo → C es un lenguaje débilmente tipado

```
int i;
long l;
void *p = NULL;      /* Puntero a void */

p = &l;               /* Correcto */
p = &i;               /* Correcto */
```

• Compatibilidad en la asignación entre punteros

- Se pueden asignar punteros del mismo tipo

```
int *p1;
int *p2;

p1 = p2;
```

- Se pueden asignar punteros a void a cualquier otro

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

```
p1 = p2;
p2 = p1;
```

- Se pueden asignar punteros de distintos tipos con *casting* explícito

```
float *p1;
int *p2;

p1 = (float *) p2; /* No se pierde información */
p2 = (int *) p1;  /* Cuidado: se puede perder información */
```



• Aritmética de punteros

- En C, a un puntero P de tipo T se le puede sumar/restar un número entero N, dando como resultado la dirección de memoria que se corresponde con $\text{direccion}(P) + N * \text{sizeof}(T)$

```
char c[10];
int i[10];
float f[10];
```

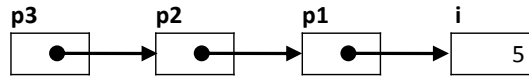
```
c + 2; /* Dirección de c + 2 Bytes, si sizeof(char)=1 Byte */
i + 2; /* Dirección de i + 4 Bytes, si sizeof(int)=2 Bytes */
f + 2; /* Dirección de f + 8 Bytes, si sizeof(float)=4 Bytes */
```

- Cuidado: en C no se comprueban límites de arrays al compilar

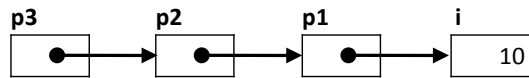
```
float f[10];
...f + 500...; /* 500 >> 10: compila, pero provoca desbordamiento */
```

• Punteros a puntero

```
int i = 5;
int *p1 = &i;
int **p2 = &p1;
int ***p3 = &p2;
```



```
*p1 = 10;
**p2 = 10;
***p3 = 10;
```



- Definición y manejo de punteros
- Punteros y arrays**
- Punteros a estructuras
- Punteros como argumentos de funciones
- Punteros y memoria dinámica

Punteros y arrays (I)

• Punteros y arrays

- El nombre de un array es el nombre simbólico de la dirección del primer byte del array => “apunta” a una dirección de memoria fija (no es un puntero que pueda variar)

- Un array de tipo T es “equivalente” a un puntero a tipo T

```
int e[10], *pe = NULL;
```

```
*e = 5;           /* equivale a e[0] = 5 */
*(e+4) = 5;       /* equivale a e[4] = 5 */
pe = e + 2;       /* equivale a pe = &e[2] */
```

- Se puede asignar un array a un puntero, pero no al revés**

```
pe = e;           /* e = pe da un error de compilación */
```

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

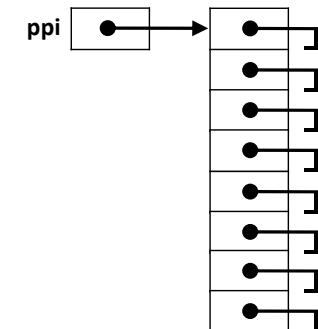
```
pe[2] = 5;
pe[4] = *(pe + 1); /* equivale a pe[4] = pe[1] */
```

Punteros y arrays (II)

• Array de punteros

```
int *ppi[8];      /* Array de 8 punteros a entero */
int i;
```

```
for( i=0; i<8; i++ ) {
    ppi[i] = NULL;
}
```



- A distinguir...

```
int *ptr          /* puntero a int */
int ptr[]         /* array de int = puntero a int */
int *ptr[]        /* array de punteros a int */
int (*ptr)[]      /* puntero a array de int */
int *(*ptr)[]     /* puntero a array de punteros a int */
...
```



- Definición y manejo de punteros
- Punteros y arrays
- **Punteros a estructuras**
- Punteros como argumentos de funciones
- Punteros y memoria dinámica

Punteros a estructuras (I)

- Ejemplo: estructura de número complejo

```
typedef struct {
    double re;
    double im;
} Complejo;
```

Punteros a estructuras (II)

- Punteros a estructuras

- El acceso a los atributos de una estructura a través de un puntero a ella se realiza normalmente con ' \rightarrow ', aunque se puede hacer mediante ' \cdot '

```
Complejo c;
Complejo *pc = NULL;
```

```
pc = &c;
```

```
/* Correctas */
printf("%f + %fi\n", pc->re, pc->im);
printf("%f + %fi\n", (*pc).re, (*pc).im);
```

```
/* Incorrectas */
printf("%f + %fi\n", *pc.re, *pc.im);
printf("%f + %fi\n", *(pc.re), *(pc.im));
```



• Punteros a estructuras

- Se suelen usar al invocar una función que recibe como argumentos de entrada estructuras => mejora en la eficiencia

```
/* Preferible */
void imprimirComplejo(Complejo *c) {
    printf("%f %fi", c->re, c->im);
}

/* No preferible */
void imprimirComplejo(Complejo c) {
    printf("%f %fi", c.re, c.im);
}
```

- Definición y manejo de punteros
- Punteros y arrays
- Punteros a estructuras
- Punteros como argumentos de funciones**
- Punteros y memoria dinámica

Punteros como argumentos de funciones (I) 22

• Paso de argumentos a una función

- Por valor** (mecanismo de invocación a una función por defecto en C)
 - La función recibe como argumento **copia(s)** de la(s) variable(s) de entrada
 - Las copias de variable se guardan de forma temporal (durante la ejecución de la función) en la PILA DEL PROGRAMA
- Por referencia**
 - La función recibe como argumento **puntero(s)** a la(s) variables de entrada

Punteros como argumentos de funciones (II) 23

• Paso de argumentos a una función



Por valor

```
void swap(int a, int b) {
    int t;

    t = a;
    a = b;
    b = t;
}
```

/* MAL */

Por referencia

```
void swap(int *a, int *b) {
    int t;

    t = *a;
    *a = *b;
    *b = t;
}
```

/* BIEN */
/* Y recomendable cuando los argumentos de entrada son estructuras */

- Definición y manejo de punteros
- Punteros y arrays
- Punteros a estructuras
- Punteros como argumentos de funciones
- **Punteros y memoria dinámica**

- En C se puede reservar memoria “dinámicamente”, es decir en tiempo de ejecución
- El manejo de la memoria dinámica se realiza mediante punteros


```
char *linea = NULL;
int **matriz = NULL;
```
- Toda memoria reservada de forma dinámica ha de liberarse antes de la finalización de la ejecución
- Las funciones para manejo de memoria se encuentran definidas en la librería **<stdlib.h>**
 - **malloc, calloc, realloc**
 - **free**

- Reserva de memoria: **malloc**

```
void *malloc(size_t numBytes)
```
- Liberación de memoria: **free**

```
void free(void *puntero)
```

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

int main(int argc, char *argv[]) {
    char *nombre = NULL;

    nombre = (char *) malloc(256 * sizeof(char));
    if( !nombre ) {
        fprintf(stderr, "Error en main: reserva de memoria.\n");
        return 1;
    }

    printf("Introduzca su nombre: ");
    gets(nombre);
    printf("Hola %s!\n", nombre);

    if( nombre ) free(nombre);

    return 0;
}
```

Punteros y memoria dinámica (IV)

28



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

int main(int argc, char *argv[]) {
    int **matriz = NULL, filas=2, columnas=3, i, j;

    /* Reserva de memoria para una matriz */
    matriz = (int **) malloc(filas * sizeof(int *));
    if( !matriz ) {
        return 1;
    }
    for( i=0; i<filas; i++ ) {
        matriz[i] = (int *) malloc(columnas * sizeof(int));
        if( !matriz[i] ) {
            for( j=0; j<i; j++ )
                free(matriz[j]);
            free(matriz);
            return 1;
        }
    }
    /* Liberacion de memoria de una matriz */
    for( i=0; i<filas; i++ )
        free(matriz[i]);
    free(matriz);

    return 0;
}
```

Punteros y memoria dinámica (V)

29

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

int main(int argc, char *argv[]) {
    Complejo *c = NULL;

    /* Reserva de memoria para un tipo de dato definido */
    c = (Complejo *) malloc(sizeof(Complejo));
    if( !c ) {
        fprintf(stderr, "Error en main: reserva de memoria.\n");
        return 1;
    }

    printf("Introduzca parte real: ");
    scanf("%f", &c->re);
    scanf("%f", &c->im);
    printf("Numero complejo creado: %f + %fi\n", c->re, c->im);

    if( c ) free(c);

    return 0;
}
```

Punteros y memoria dinámica (VI)

30

- Reserva de memoria inicializada a 0: **calloc**

```
void *calloc(size_t numElementos, size_t numBytesElemento)
```

- Reasignación de memoria: **realloc**

```
void *realloc(void *puntero, size_t numBytes)
```