Tema 2. Punteros

FUNDAMENTOS DE PROGRAMACIÓN II

Profesor: Baldomero Imbernón Tudela

Escuela Politécnica Superior
Grado en Ingeniería Informática

Contenidos

- Definición
- Declaración de punteros
- Operadores
  - El operador referencia: &
  - El operador indirección: *
- Asignación de punteros
- Punteros como argumentos
- Devolver punteros

Contenidos

- **Definición**
- Declaración de punteros
- Operadores
  - El operador referencia: &
  - El operador indirección: *
- Asignación de punteros
- Punteros como argumentos
- Devolver punteros

Definición (I)

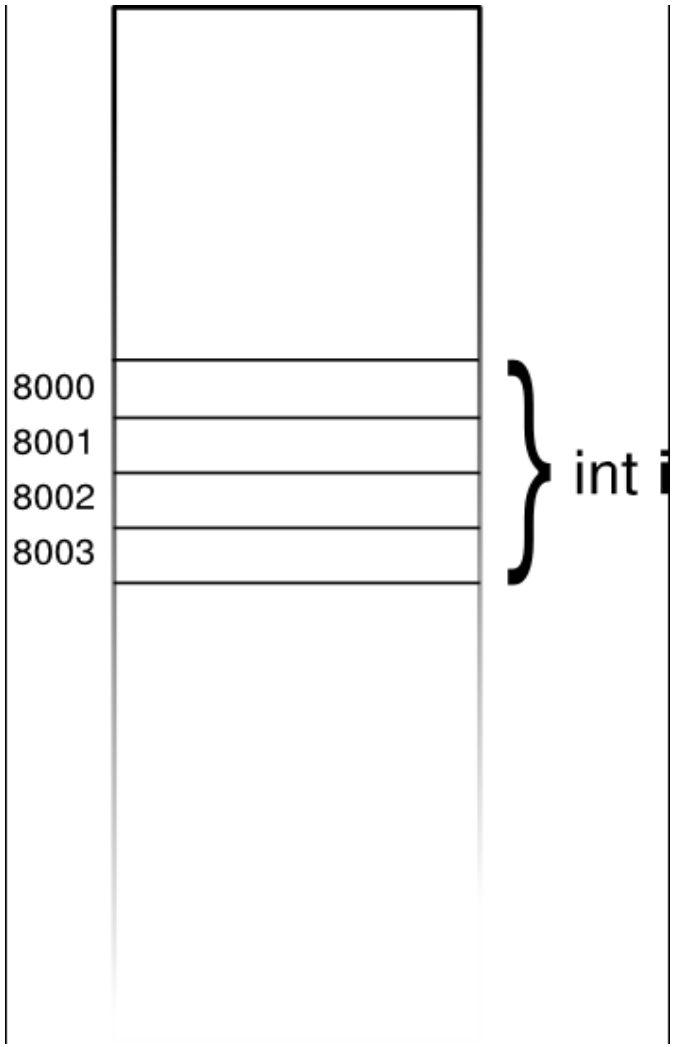
- Para entender bien el concepto de puntero hay que visualizar qué representa a nivel máquina.
- Cualquier ordenador moderno tiene una memoria principal dividida en bytes.
  - Cada byte de memoria tiene una dirección única que lo identifica.
  - Si una memoria tiene una capacidad de n bytes, el rango de direcciones irá desde 0 hasta n-1.
- Un ejecutable consiste en: **código** y **datos**.
  - Los datos son las variables de nuestros programas en C.
  - Cada variable en el programa ocupará uno o más bytes de memoria.

Definición (II)

Memoria RAM

Dirección	Contenido
0	01100010
1	10011000
2	11110111
3	00011110
4	00001111
5	00000000
6	11111010
7	00101010
8	10010110
9	10001110
10	10011010
11	00000001
...	

Definición (III)



- Supongamos una variable **i** que ocupe las posiciones de memoria desde la 8000 a la 8003...

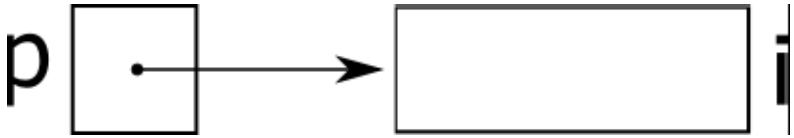
- Aunque las direcciones se representen por números, el rango puede no ser entero y necesitar de otro tipo.

- Por eso surgen las variables puntero.

- Un **puntero** es una variable capaz de representar una dirección de memoria.

Definición (IV)

- Como normal general, no aparecerán las direcciones.
- Para representar el hecho de que una variable **`p`**, almacena la dirección de una variable **`i`**, se utilizará la siguiente notación:



- Si la variable **`i`**, estuviera en la dirección de memoria 8000, el valor de la variable **`p`** sería su dirección, es decir 8000.

Contenidos

- Definición
- **Declaración de punteros**
- Operadores
  - El operador referencia: &
  - El operador indirección: *
- Asignación de punteros
- Punteros como argumentos
- Devolver punteros

Declaración de punteros (I)

- Una variable puntero (un puntero) se declara como cualquier otra variable.
  - Tan solo hay preceder al nombre de la variable con un asterisco:

```
int *p;
```

  - La anterior declaración hace que **p** sea un puntero capaz de apuntar a objetos de tipo **int**.
    - Se ha utilizado el concepto objeto, en vez de variable, porque el puntero **p** es capaz de apuntar a zonas de memoria que no pertenezcan a ninguna variable.
  - Los punteros pueden ser declarados junto a otras variables:

```
int i, j, a[10], *p, *q;
```

Declaración de punteros (II)

- C requiere que cada puntero apunte solo a objetos de un tipo determinado:

```
int *p; // p apunta solo a enteros
double *q; // q apunta solo a doubles
char *r; // r apunta solo a caracteres
```

- No hay ninguna restricción en el tipo referenciado, incluso un puntero puede apuntar a una variable que sea a su vez un puntero.

Contenidos

- Definición
- Declaración de punteros
- **Operadores**
  - El operador referencia: &
  - El operador indirección: *
- Asignación de punteros
- Punteros como argumentos
- Devolver punteros

Operadores

- Para el uso de punteros, C proporciona un par de operadores.
  - Operador **&** (**REFERENCIA**).
    - Permite determinar la dirección de una variable.
    - Si **x** es una variable, **&x** es la dirección de **x** en memoria.
  - Operador ***** (**INDIRECCIÓN**).
    - Permite obtener acceso al objeto apuntado por el puntero.
    - Si p es un puntero, *p es el objeto al que apunta.

Contenidos

- Definición
- Declaración de punteros
- Operadores
  - **El operador referencia: &**
  - El operador indirección: *
- Asignación de punteros
- Punteros como argumentos
- Devolver punteros

El operador referencia (I)

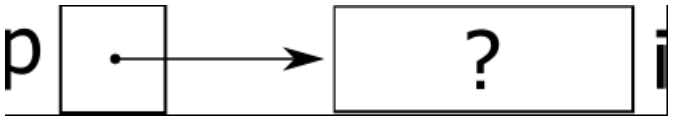
- Declarar un puntero reserva espacio para el puntero, pero no lo hace apuntar a ningún objeto:

```
int *p; // no apunta a un lugar en concreto
```

- Antes de utilizar un puntero, es imprescindible inicializarlo.
  - Se le puede asignar la dirección de cualquier variable, un l-value, usando el operador &.

```
int i, *p;
...
p = &i;
```

- Esta asignación hace que **p** obtenga como valor la dirección de **i**. Ahora **p** apunta a **i**:



El operador referencia (II)

- Se puede inicializar un puntero al mismo tiempo que se declara:

```
int i;
int *p = &i;
```

- ... o incluso en una misma línea, siempre que declaremos antes la variable i:

```
int i, *p = &i;
```

Contenidos

- Definición
- Declaración de punteros
- Operadores
  - El operador referencia: &
  - **El operador indirección: ***
- Asignación de punteros
- Punteros como argumentos
- Devolver punteros

El operador indirección (I)

- Una vez que un puntero apunta a un objeto, podemos utilizar el operador ***** (indirección) para acceder a él.
- Si **p** apunta a **i**, podemos mostrar el valor de la variable **i** de esta forma:

```
printf("%d", *p);
// printf mostrará el valor de i
```

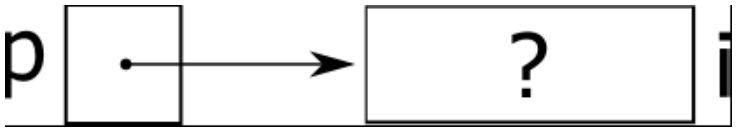
- Podemos ver ambos operadores, **&** y *****, como inversos.
  - Si aplicamos **&** sobre una variable obtenemos un puntero, al aplicarle ***** volveremos al valor de la variable:

```
j = *&i;    // Equivale a j = i;
```

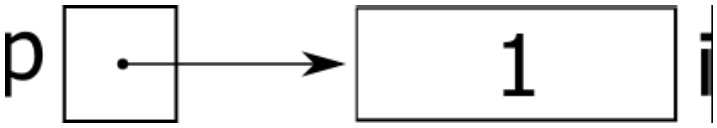
El operador indirección (II)

- Si **p** apunta a **i**, ***p** es un alias de **i**.
- A partir de ese momento ***p** e **i** tienen el mismo valor, modificando uno se modifica el otro.
  - ***p** es un l-valor, se le pueden asignar valores.

```
p = &i;
```



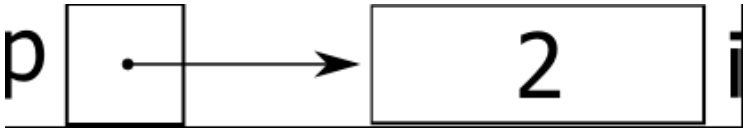

```
i = 1;
```



El operador indirección (III)

```
printf("%d\n", i);  // Muestra 1
printf("%d\n", *p); // Muestra 1
```

```
*p = 2;
```




```
printf("%d\n", i);  // Muestra 2
printf("%d\n", *p); // Muestra 2
```

El operador indirección (IV)

- **NO aplicar el operador * a un puntero no inicializado.**
- Hacerlo puede llevar a un comportamiento completamente indefinido,
- Por ejemplo, en la siguiente llamada a printf el programa puede mostrar basura, provocar un fallo de segmentación o cualquier otro efecto:

```
int *p;
printf("%d", *p); // MAL
```

- **Asignar un valor a *p (cuando p no es inicializado) es especialmente peligroso**, si p contiene una dirección válida, se sobrescribirá los datos de esa dirección, si es una zona de memoria del programa, el programa fallará, si la zona es del sistema operativo, el programa se colgará o provocará un fallo de segmentación (el compilador suele avisar de estas condiciones con un warning).

```
int *p;
*p = 1;  // MAL
```

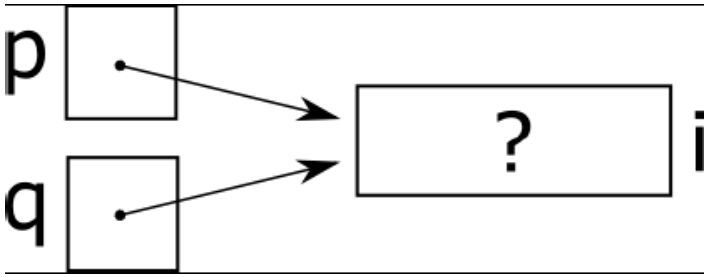
Contenidos

- Definición
- Declaración de punteros
- Operadores
  - El operador referencia: &
  - El operador indirección: *
- **Asignación de punteros**
- Punteros como argumentos
- Devolver punteros

Asignación de punteros (I)

- C permite el uso del operador asignación para copiar punteros, siempre que sean del mismo tipo.

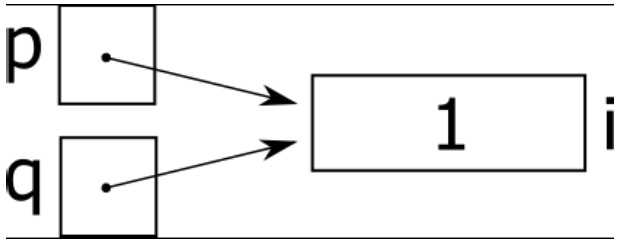
```
int i, j, *p, *q;
p = &i;
q = p;
```



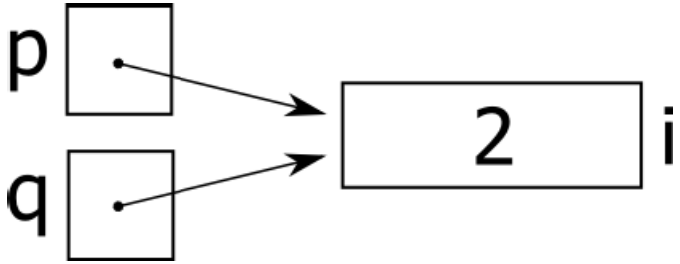
  - Ahora **p** y **q** apuntan a **i**. Podemos modificar el valor de **i** a través de ***p** y a través de ***q**

Asignación de punteros (II)

```
*p = 1;
```




```
*q = 2;
```



- Cualquier número de punteros pueden apuntar al mismo objeto.

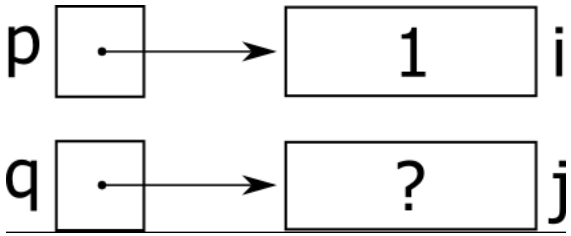
Asignación de punteros (III)

- No hay que confundir estas expresiones:

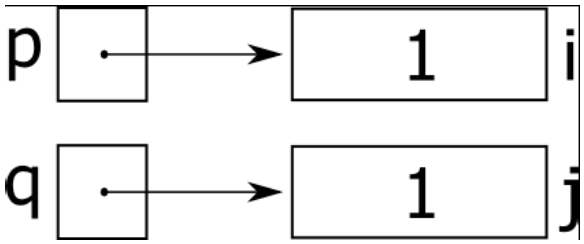
```
q = p; *q = *p;
```

- La primera es una asignación de punteros, después de la asignación, ambos apuntarán al mismo objeto.

```
p = &i;
q = &j;
i = 1;
```




```
*q = *p;
```



Contenidos

- Definición
- Declaración de punteros
- Operadores
  - El operador referencia: &
  - El operador indirección: *
- Asignación de punteros
- **Punteros como argumentos**
- Devolver punteros

Punteros como argumentos (I)

- El paso de parámetros a funciones se puede realizar de dos formas:
- **Paso de parámetros por valor.**
  - Se produce una copia de los argumentos a los parámetros.
  - Cualquier cambio producido en al parámetro no afecta al argumento original.
  - C siempre pasa los parámetros por valor.
- **Paso de parámetros por referencia.**
  - Permite que la modificación de un parámetro afecte al argumento original.
  - C no permite el paso de parámetros por referencia, pero sí permite pasar punteros como parámetros.

Punteros como argumentos (II)

- ¿Cómo podríamos escribir una función que intercambie el valor de dos variables?

```
#include <stdio.h>
void swap(int n1, int n2);
int main()
{
    int a = 5, b = 16;
    printf("Antes:   a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("Después: a = %d, b = %d\n", a, b);
    return (0);
}
void swap(int n1, int n2)
{
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Este ejemplo no funciona, ya que en la llamada a swap los argumentos se copian.

Los valores originales no se ven modificados.

Punteros como argumentos (III)

- En vez de pasar los valores, pasamos punteros:

```
#include <stdio.h>
void swap(int *n1, int *n2);
int main()
{
    int a = 5, b = 16;
    printf("Antes:   a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("Después: a = %d, b = %d\n", a, b);
    return (0);
}
void swap(int *n1, int *n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

Punteros como argumentos (IV)

```
#include <stdio.h>
#include <stdlib.h>
void descomponer(double x, long *parte_entera,
                 double *parte_fracc);
int main()
{
    long i;
    double d;
    descomponer(3.1415926L, &i, &d);
    printf("3.1415926 = %ld + %g\n", i, d);
    return (EXIT_SUCCESS);
}
void descomponer(double x, long *parte_entera,
                 double *parte_fracc)
{
    *parte_entera = (long) x;
    *parte_fracc = x - *parte_entera;
}
```

Punteros como argumentos (V)

- El uso de punteros como parámetros no es nada nuevo, los estamos utilizando desde que se estudiamos **`scanf`**:

```
int i;
...
scanf(“%d”, &i);
```

- scanf necesita un puntero para saber dónde poner el valor que lea de la entrada estándar.
- No siempre se tiene que utilizar el operador &:

```
int i, *p;
...
p = &i;
scanf(“%d”, p);
```

Ejemplo (I)

(encontrar el valor máximo y mínimo de un array)

```
#include <stdio.h>
#include <stdlib.h>
#define TAM 10

void max_min(int a[], int n, int *max,
             int *min);

int main(void)
{
    int a[TAM], i, minimo, maximo;
    printf("Introduzca %d números: ", TAM);
    for (i = 0; i < TAM; i++)
        scanf("%d", &a[i]);
    max_min(a, TAM, &maximo, &minimo);
    printf("El número máximo es: %d\n", maximo);
    printf("El número mínimo es: %d\n", minimo);
    return (EXIT_SUCCESS);
}
```

```
void max_min(int a[], int n,
             int *max,
             int *min)
{
    int i;
    *max = *min = a[0];
    for (i = 1; i < n; i++) {
        if (a[i] > *max)
            *max = a[i];
        else if (a[i] < *min)
            *min = a[i];
    }
}
```

Contenidos

- Definición
- Declaración de punteros
- Operadores
  - El operador referencia: &
  - El operador indirección: *
- Asignación de punteros
- Punteros como argumentos
- **Devolver punteros**

Devolver punteros (I)

- Se pueden utilizar punteros tanto como argumentos en las llamadas como valores devueltos.
  - Ejemplos: strcpy, strcat...
- Ejemplo:

```
int *max(int *a, int *b)
{
   if (*a > *b)
      return a;
   else
      return b;
}
```

- La función devolverá un puntero al mayor entero:

```
int *p, i, j;
p = max(&i, &j);
```

Devolver punteros (II)

- Nunca hay que devolver un puntero a una variable local de una función.
  - Las variables locales no existen una vez que la función termina:

```
int *funcion(void)
{
    int i;
    ...
    return &i;
}
```

Bibliografía

- King, K.N. **C Programming. A modern approach.** 2ªed. Ed. W.W. Norton & Company. Inc. 2008. Chapter 11.
- Khamtane Ashok**. Programming in C**. Ed. Pearson. 2012.
- Ferraris Llanos, R. D. **Fundamentos de la Informática y Programación en C**. Ed. Paraninfo. 2010.