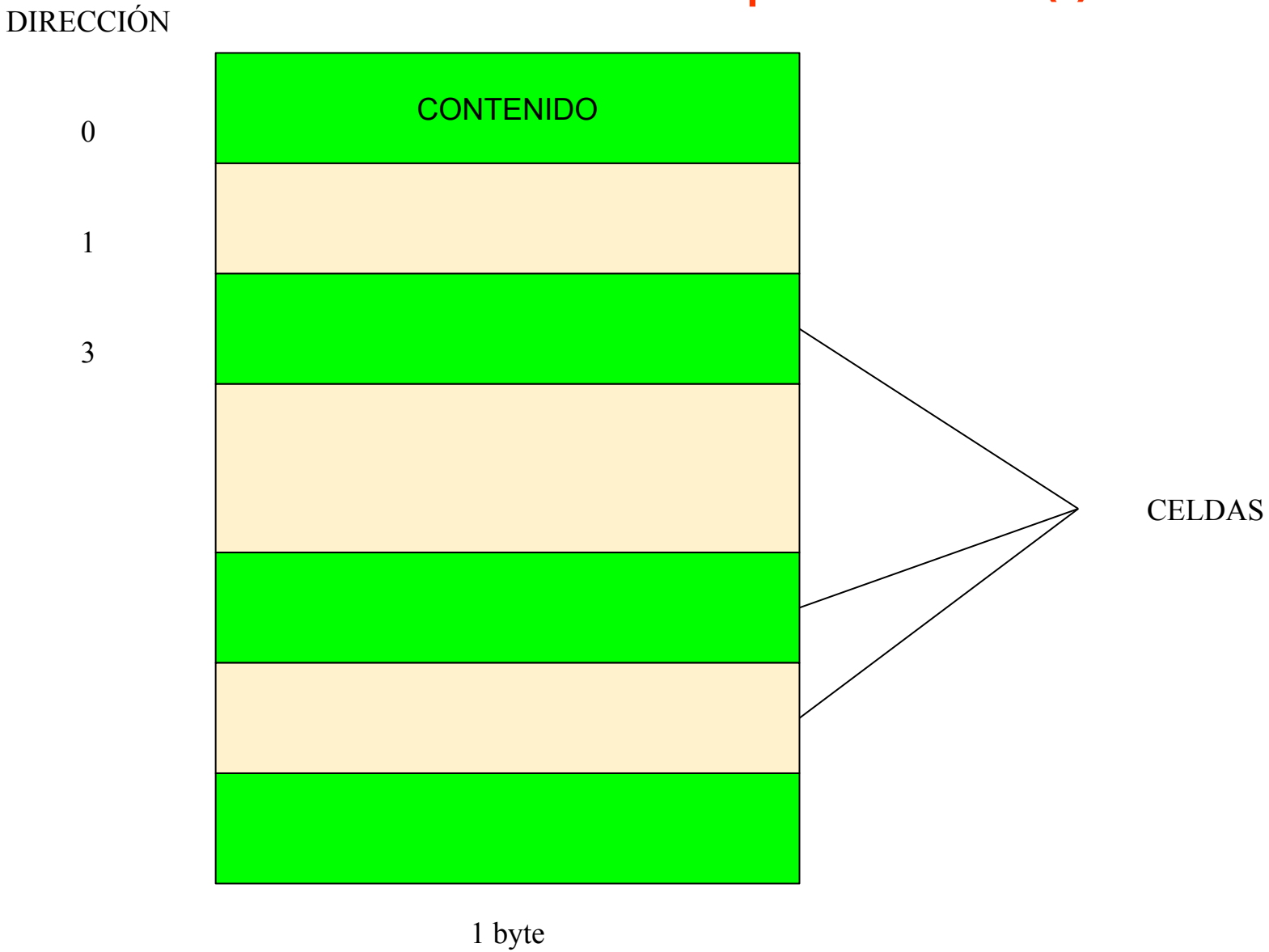
Cap. 7 - Punteros

7.0 Memoria del Computador

- Lugar donde se almacenan datos y instrucciones
- Se compone de celdas de tamaño 1 byte (8 bits)
- Cada celda tiene una dirección
- Con esa dirección se puede acceder a la celda y obtener el valor almacenado
- Una palabra es un número de bits que se leen/escriben de una sola vez.

7.0 Memoria del Computador (I)



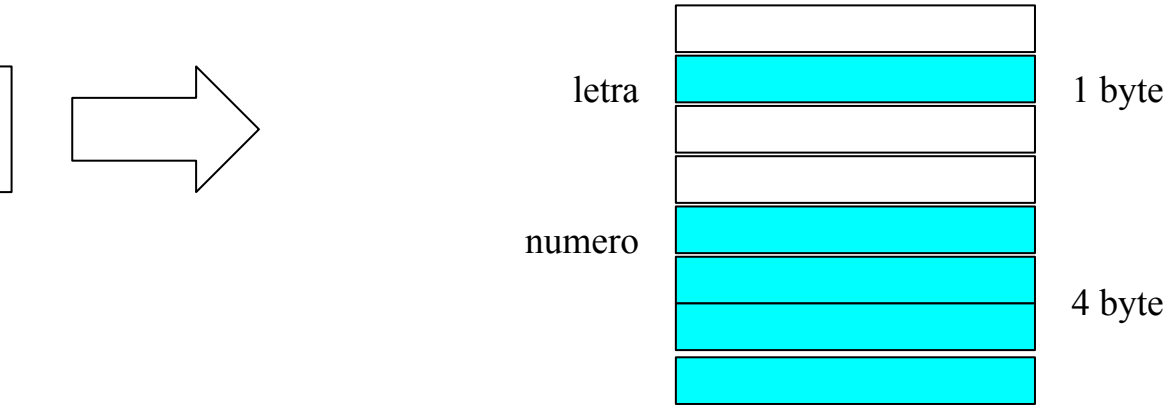
7.0 Memoria del Computador (II)

- Una variables equivale a una zona de la memoria reservada para almacenar un valor concreto que pertenece a un tipo de dato
- Cualquier variable que se define debe ocupar un número entero de bytes contiguos
- Para acceder a una variable el compilador necesita de:
  - Número de bytes que componen la variable
  - Dirección inicial de la variable

7.0 Memoria del Computador (III)

- El nombre de la variable es sustituido por la dirección del byte inicial y el número de bytes está definido por el tipo de datos.

```
char letra;
long numero;
```

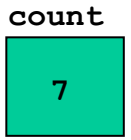


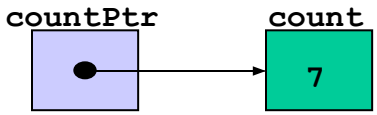
7.1 Introducción

- Punteros
  - Poderoso, pero difícil de dominar
  - Simular llamada por referencia
  - La estrecha relación con los arreglos y las cadenas

7.2 Declaraciones de Punteros e Inicialización

- Punteros variables
  - Contienen direcciones de memoria como sus valores
  - Las variables normales contienen un valor específico (referencia directa)
  - Los punteros contienen la dirección de una variable que tiene un valor específico (*referencia indirecta*)
  - Indirección - referencia a un valor de puntero





7.2 Declaraciones de Punteros e Inicialización (II)

- Declaración de Punteros
  - **`*`** utilizado con variables punteros

    `int *myPtr;`
  - Declaro un puntero a **`int`** (puntero de tipo **`int *`**)
  - Multiples punteros, multiple **`*`**

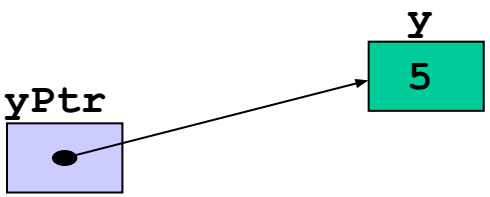
    `int *myPtr1, *myPtr2;`
  - *Se puede declarar punteros a cualquier tipo de dato*
  - Iniliziar punteros **`0`**, **`NULL`**, or una dirección
    - **`0`** o **`NULL`** - apunta a nada (**`NULL`** preferido)

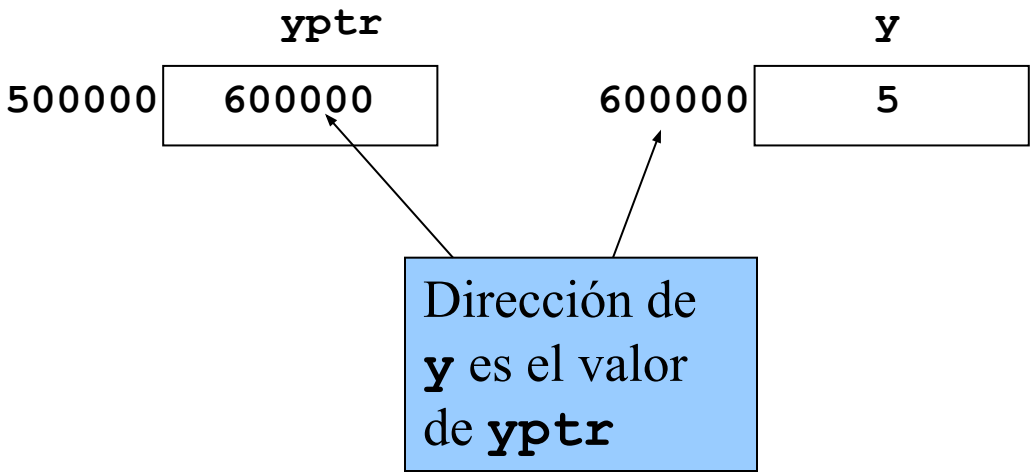
7.3 Operador Puntero

- `&` (operador de direccionamiento)
  - Retorna la dirección del operando

```
int y = 5;
int *yPtr;
yPtr = &y;   //yPtr consigue la direcciónde y
```

- `yPtr` "apunta a" `y`





7.3 Operador Puntero (II)

- * (operador de indirección/desreferenciación)
  - Retorna un sinónimo/alias de lo que su operando *apunta* a

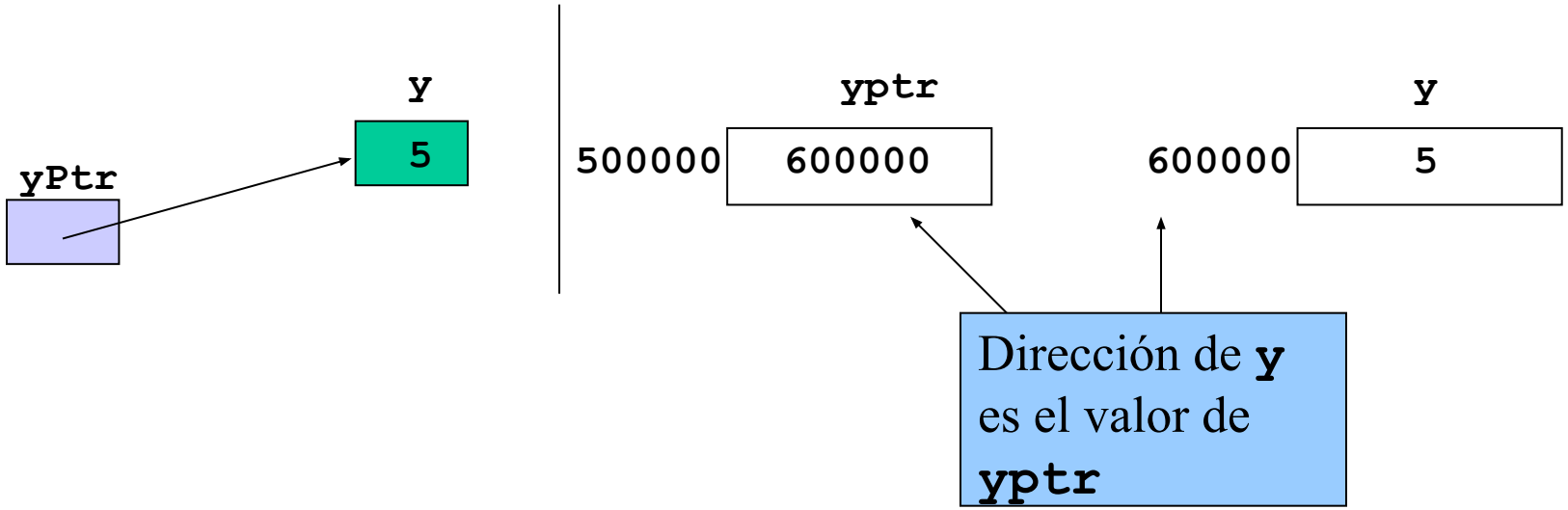
    ***yptr** retorna **y** (porque **yptr** apunta a **y**)
  - * puede ser utilizado para asignamiento
    - Retorna un alias a un objeto

    ```
    *yptr = 7; // changes y to 7
    ```
  - Puntero Dereferenciado (operando de *) debe ser un *lvalue* (no constante)

7.3 Operador Puntero (III)

- * y & son inversos
  - Ellos se cancelan uno a otro

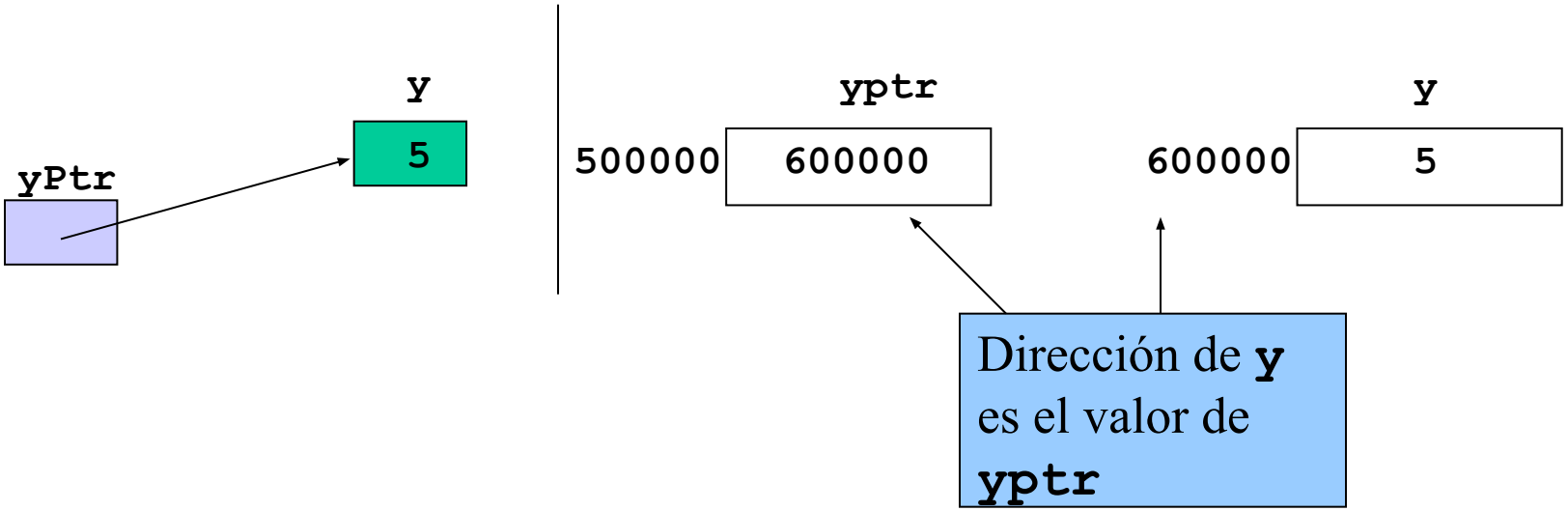
```
*&yptr  ->  * (&yptr) -> * (dirección de yptr)->
  retorna alias de lo que operador apunta a -> yptr
*&yptr  ->  * (&yptr) -> * (5000000) -> 600000
```



7.3 Operador Puntero (IV)

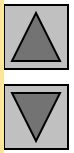
- * y & son inversos
  - Ellos se cancelan uno a otro

```
&*yptr -> &(*yptr) -> &(y) -> retorna dirección de
  y, el cual es yptr -> yptr
&*yptr -> &(*yptr) -> &(y) -> 600000
```



```c
/* Fig. 7.4: fig07_04.c
   Uando los operadores & y * */
#include <stdio.h>

int main()
{
   int a;        /* a es un entero */
   int *aPtr;    /* aPtr es un apuntador a entero */

   a = 7;
   aPtr = &a;    /* aPtr apunta dirección de a */

   printf( "La dirección de a es %p"
           "\nEl valor de aPtr es %p", &a, aPtr );

   printf( "\n\nEl valor de a %d"
           "\nEl valor de *aPtr es %d", a, *aPtr );

   printf( "\n\nMuestra que * y & son inversos"
           "uno de otro.\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0;
}
```

```
La dirección de a es 0012FF88
El valor de aPtr es 0012FF88

El valor de a es 7
El valor de *aPtr es 7
Muestra que * y & son complementos uno de otro.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```



Outline

1. Declarción de variables

2 Initialición de variables

3. Impresión

Salida del programa

```c
/* Fig. 7.4: fig07_04.c
   Uando los operadores & y * */
#include <stdio.h>

int main()
{
   int a;        /* a es un entero */
   int *aPtr;    /* aPtr es un apuntador a entero *

   a = 7;
   aPtr = &a;    /* aPtr apunta dirección de a */

   printf( "La dirección de a es %p"
           "\nEl valor de aPtr es %p", &a, aPtr );

   printf( "\n\nEl valor de a %d"
           "\nEl valor de *aPtr es %d", a, *aPtr );

   printf( "\n\nMuestra que * y & son inversos"
           "uno de otro.\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0;
}
```

La dirección de **a** es el valor de **aPtr**.

El operador * retorna un alias a lo que su operador apunta. **aPtr** apunta a **a**, asi ***aPtr** retorna **a**.

Note como * y & son inversos

Outline

1. Declarción de
   le variables
3. Impresión

Salida del programa

```
La dirección de a es 0012FF88
El valor de aPtr es 0012FF88

El valor de a es 7
El valor de *aPtr es 7
Muestra que * y & son complementos uno de otro.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

7.4 Llamada a Funciones por Referencia

- LLamada por referencia con argumento punteros
  - Pase la dirección del argumento usando & operador
  - Le permite cambiar la ubicación real en la memoria
  - Los arreglos no se pasan con & porque el nombre del arreglo ya es un puntero

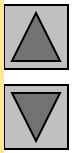
- * operador
  - Usado como alias/apodo para la variable dentro de la función

```
void double(int *number)
{
   *number = 2 * (*number);
}
```

   `*number` usado como apodo por la variable pasada

```c
/* Fig. 7.7: fig07_07.c
   Cubo una variable usando llamada por referencia
   con con un argumento puntero */

#include <stdio.h>

void cubeByReference( int * );   /* prototype */

int main()
{
   int numero = 5;

   printf( "El valor original del numero es %d", numero );
   cubeByReference( &numero );
   printf( "\nEl nuevo valor del numero es %d\n", numero );

   return 0;
}

void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  // el cubo del number
}                                  // en el main
```

```
El valor original de numero es 5
El nuevo valor de numero es 125
```



Outline

1. Función prototipo - toma un puntero an `int`.

1.1 Inicializa variables

2. Llamada a función

3. Define función

Salida de Programa

```
1   /* Fig. 7.7: fig07_07.c
2      Cubo una variable usando llamada por referencia
3      con con un argumento puntero */
4
5   #include <stdio.h>
6
7   void cubeByReference( int * );
8
9   int main()
10  {
11     int number = 5;
12
13     printf( "El valor original del numero es %d", number );
14     cubeByReference( &number );
15     printf( "\nEl nuevo valor del numero es %d\n", number );
16
17     return 0;
18  }
19
20  void cubeByReference( int *nPtr )
21  {
22     *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
23  }
```

Note como la dirección de **number** es dado- **cubeByReference** expectera un puntero (una dirección de la variable).

Dentro de **cubeByReference**, ***nPtr** es usado (***nPtr** es **number**).

Outline

ión prototipo -
n puntero an int.

ializa variables

2. Llamada a función

3. Define función

Salida de Programa

```
The original value of number is 5
The new value of number is 125
```

7.5 Usando el calificador de la Const con punteros

- **`const`** Calificador- variable no puede ser modificada
  - Es buena idea tener **const** si la función no necesita cambiar una variable
  - Intentar cambiar una **const** es un error del compilador
- **`const`** Punteros - apuntan a la misma ubicación de la memoria
  - Debe ser inicializado cuando se declara

```
int *const myPtr = &x;
```

    - Tipe `int *const` - puntero constante a un `int`

```
const int *myPtr = &x;
```

    - Un puntero regular para un `const int`

```
const int *const Ptr = &x;
```

    - `const` apunta a `const int`
    - `x` puede ser cambiada, pero no `*Ptr`

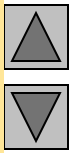
```
1   /* Fig. 7.13: fig07_13.c
2      Intenta modificar un puntero constanta
3      a dato no constante */
4
5   #include <stdio.h>
6
7   int main()
8   {
9      int x, y;
10
11     int * const ptr = &x; /* ptr es un puntero constante a
12                              entero. Un entero puede ser modificado
13                              por ptr, pero ptr siempre apunta
14                              a la misma dirección de memoria. */
15     *ptr = 7;
16     ptr = &y;
17
18     return 0;
19  }
```

Cambiar ***ptr** se permite - **x** no es constante.

Cambiar **ptr** es un error - **ptr** un puntero constante.

```
FIG07_13.c:
Error E2024 FIG07_13.c 16: Cannot modify a const object in
function main
*** 1 errors in Compile ***
```

Salida de Programa



Outline

1. Declara variables

1.1 Declara `const` pointer a un `int`.

2. Cambia `*ptr` (el cual es `x`).

2.1 Intenta cambiar `ptr`.

3. Salida

7.6 Ordenamiento de Burbuja usando llamada por Referencia

- Implementando `bubblesort` usando punteros
  - Intercambiando dos elementos (Swap)
  - **`swap`** debe recibir la dirección (usando **&**) del arreglo
    - Los elementos del arreglo tienen llamada por valor por defecto
  - Usando punteros y el operador *****, **`swap`** puede conmutar elementos del arreglo

- Pseudocodigo

Inicializar arreglo
    imprima datos en el orden original
Llamar a la función `bubblesort`
    imprima arreglo ordenado
Define `bubblesort`

7.6 Ordenamiento de Burbuja usando llamada por Referencia (II)

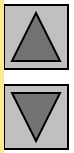
- **`sizeof`**
  - Retorna tamaño del operando en bytes
  - Para arreglos: tamaño de 1 elemento * numero de elementos
  - Si **`sizeof(int) = 4 bytes`**, entonces

```
int myArray[10];
printf( "%d", sizeof( myArray ) );
```

    va imprimir 40

- **`sizeof`** puede ser usado con
  - Nombre de Variable
  - Tipe de nombre
  - Valores Constantes

```c
/* Fig. 7.15: fig07 15.c
   Este programa pone los valores en un arreglo, ordena los
   orden ascendente, e imprime el arreglo resultante. */
#include <stdio.h>
#define SIZE 10
void bubbleSort( int *, const int );

int main()
{

   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
   int i;

   printf( "Datos en orden original\n" );

   for ( i = 0; i < SIZE; i++ )
      printf( "%4d", a[ i ] );

   bubbleSort( a, SIZE );          /* ordenamiento*/
   printf( "\nDatos en orden ascendente\n" );

   for ( i = 0; i < SIZE; i++ )
      printf( "%4d", a[ i ] );

   printf( "\n" );

   return 0;
}

void bubbleSort( int *array, const int size )
{
   void swap( int *, int * );
```



Outline

1. Inicializa el arreglo

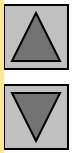
1.1 Declara variables

2. Imprime arreglos

2.1 Llama `bubbleSort`

2.2 Imprime arreglo

A `Bubblesort` se le pasa la dirección de los elementos del arreglo (punteros). El nombre de un arreglo es un puntero.



Outline

```
33      int pass, j;
34      for ( pass = 0; pass < size - 1; pass++ )
35
36         for ( j = 0; j < size - 1; j++ )
37
38            if ( array[ j ] > array[ j + 1 ] )
39               swap( &array[ j ], &array[ j + 1 ] );
40   }
41
42   void swap( int *element1Ptr, int *element2Ptr )
43   {
44      int hold = *element1Ptr;
45      *element1Ptr = *element2Ptr;
46      *element2Ptr = hold;
47   }
```

3. Definición de la Función

```
Dato en orden original
   2    6    4    8   10   12   89   68   45   37
Dato en orden ascendente
   2    4    6    8   10   12   37   45
```

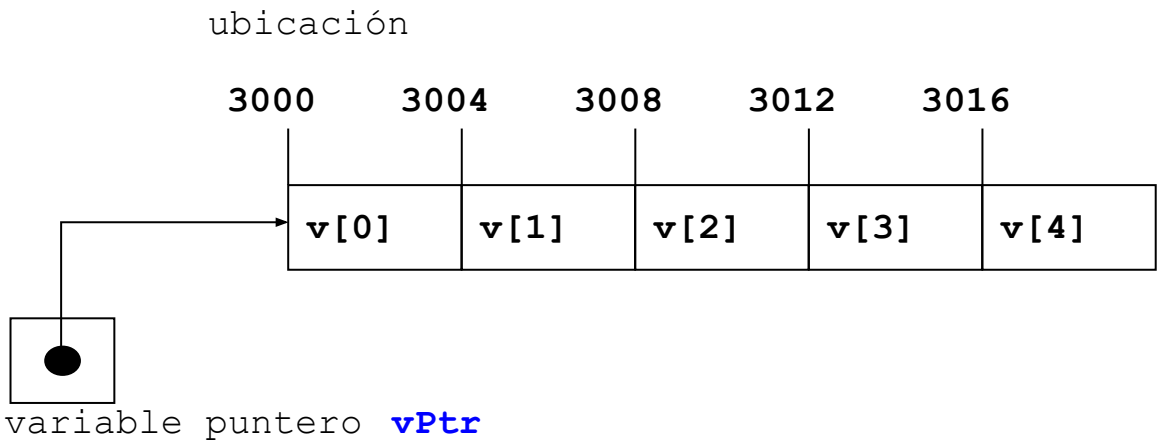
Salida del Programa

7.7 Expresiones de puntero y aritmética de puntero

- Las operaciones aritméticas pueden realizarse en punteros
  - Puntero de incremento/decremento (++ o --)
  - Añade un número entero a un puntero (+ o += , - o -=)
  - Los punteros pueden restarse entre sí
  - Las operaciones no tienen sentido a menos que se realicen en un arreglo

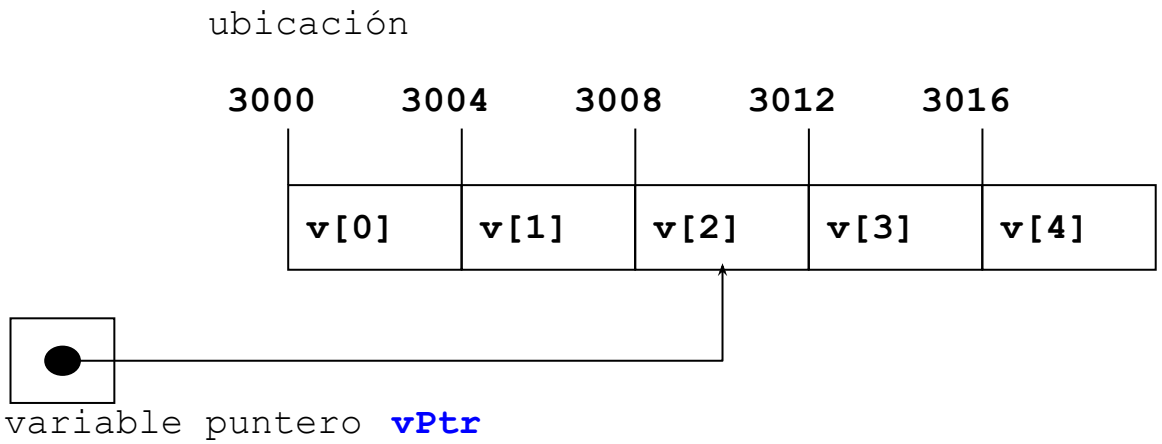
7.7 Expresiones de puntero y aritmética de puntero (II)

- Arreglo de 5 elementos **`int`** en una máquina con 4 byte por **`int`**
  - **`vPtr`** apunta al primer elemento de **`v[0]`** en la dirección **`3000`**.
    (**`vPtr = 3000`**)



7.7 Expresiones de puntero y aritmética de puntero (II)

- Arreglo de 5 elementos **`int`** en una máquina con 4 byte por **`int`**
  - **`vPtr`** apunta al primer elemento de **`v[0]`** en la dirección **`3000`**.
    (**`vPtr = 3000`**)
  - **`vPtr +=2;`** configura **`vPtr`** a **`3008`**
    - **`vPtr`** apunta a **`v[2]`** (incrementado por 2), pero la máquina tiene 4 byte por **`int`**.



7.7 Expresiones de puntero y aritmética de puntero (III)

- Sustracción de punteros
  - Retorna números de elementos de uno a otro

```
vPtr2 = v[2];
vPtr = v[0];
vPtr2 - vPtr == 2.
```

- Comparación de Punteros (**<**, **==** , **>**)
  - Observa qué puntero apunta al elemento del arreglo de mayor número
  - También, observa si un puntero apunta a **0**

7.7 Expresiones de puntero y aritmética de puntero (IV)

- Punteros del mismo tipo pueden ser asignados a otros
  - Si no son del mismo tipo, un operador **`cast`** debe ser usado
  - Excepción: puntero a `void` (tipo `void *`)
    - Puntero genérico, representa cualquier tipo
    - No es necesario hacer un `casting` para convertir un puntero en un puntero `void`
    - No casting needed to convert a pointer to **`void`** pointer
    - punteros **`void`** no pueden ser desreferenciado

7.8 La relación entre Punteros y Arreglos

- Los arreglos y los punteros están estrechamente relacionados
  - El nombre del arreglo es como un puntero constante
  - Los punteros pueden hacer operaciones de suscripción de arreglos
- Declare un arreglo **`b[5]`** y un puntero **`bPtr`**

```
bPtr = b;
```

El nombre del arreglo es en realidad una dirección del primer elemento

o

```
bPtr = &b[0]
```

Asigna explícitamente a bPtr la dirección del primer elemento

7.8 La relación entre Punteros y Arreglos (II)

- Elemento **`b[n]`**
  - puede ser accesado por **`*( bPtr + n )`**
  - **`n`** - desplazamientos (puntero/notación desplazamiento)
  - Arreglo mismo puede usar aritmética de punteros.

    `b[3]` igual como **`*(b + 3)`**
  - Punteros pueden ser suscrito (puntero/notación índice)

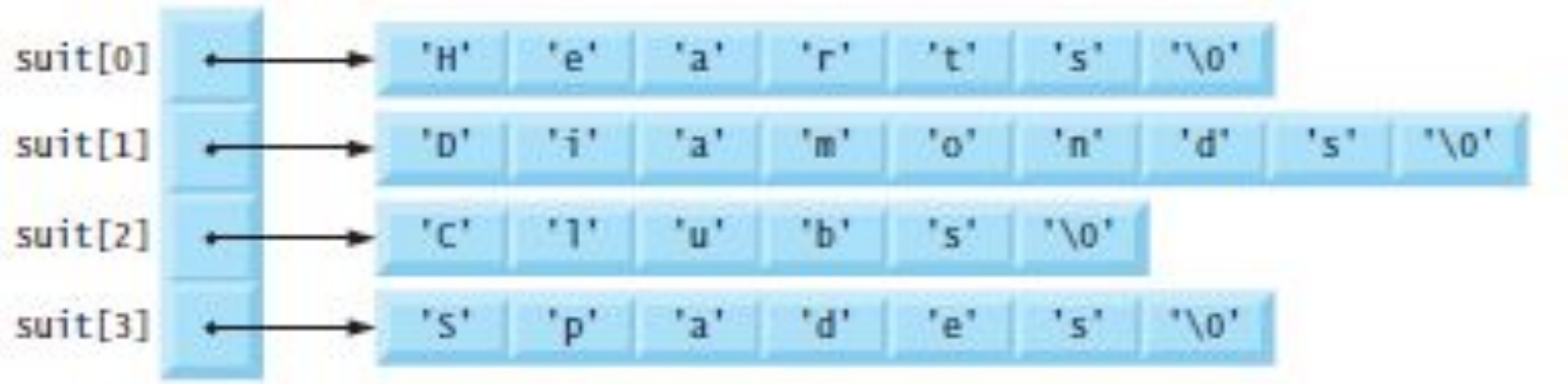
    `bPtr[3]` igual como **`b[3]`**

7.9 Arreglo de Punteros

- Los arreglos pueden contener punteros - arreglos de cadenas

```
char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades" };
```

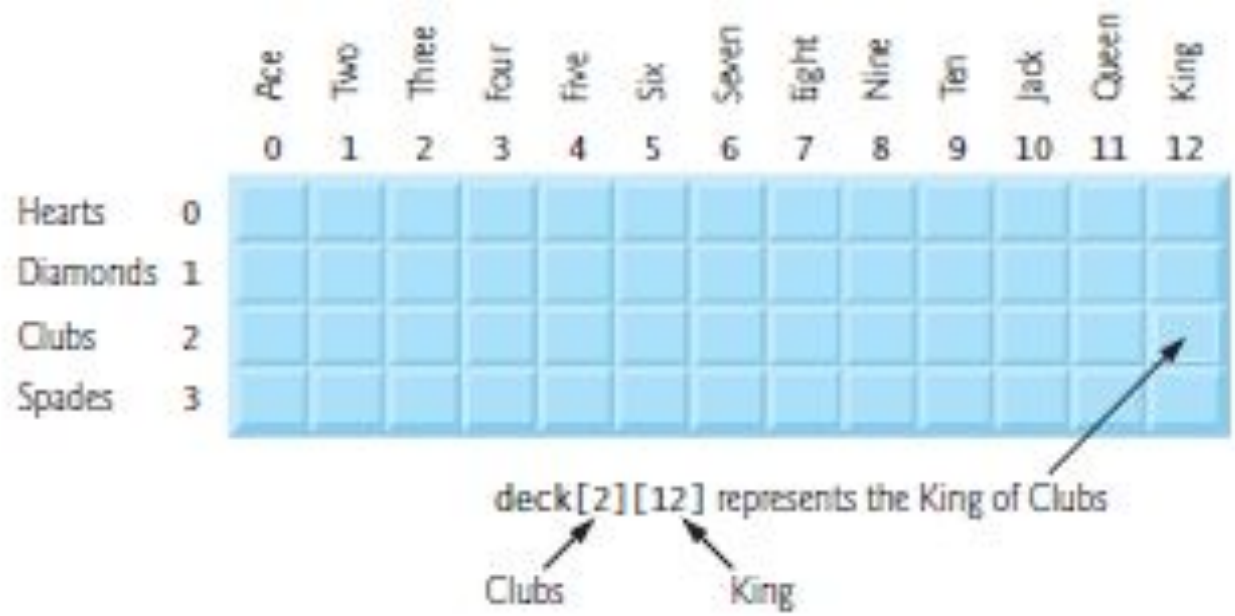
  - Cadena: puntero al primer caracter
  - **`char *`** - cada elemento de **`suit`** es un puntero a **`char`**
  - Cadenas no están en el arreglo - solo los *punteros* a cadenas están en el arreglo



- **`suit`** tiene tamaño fijo, pero las cadenas pueden ser de distinto tamaños.

7.10 Caso de Estudio: Un simulacro de barajado y reparto de cartas

- Programa de barajado de Cartas
  - Usar una serie de punteros para las cadenas
  - Usar un arreglo de doble índice (palo, cara)



  - Los números de 1 a 52 van en el arreglo - este es el orden en el que se tratan

7.10 Caso de Estudio: Un simulacro de barajado y reparto de cartas

- Pseudocodigo - Top level: *Barajar y repartir 52 cartas*

Primer Refinamiento

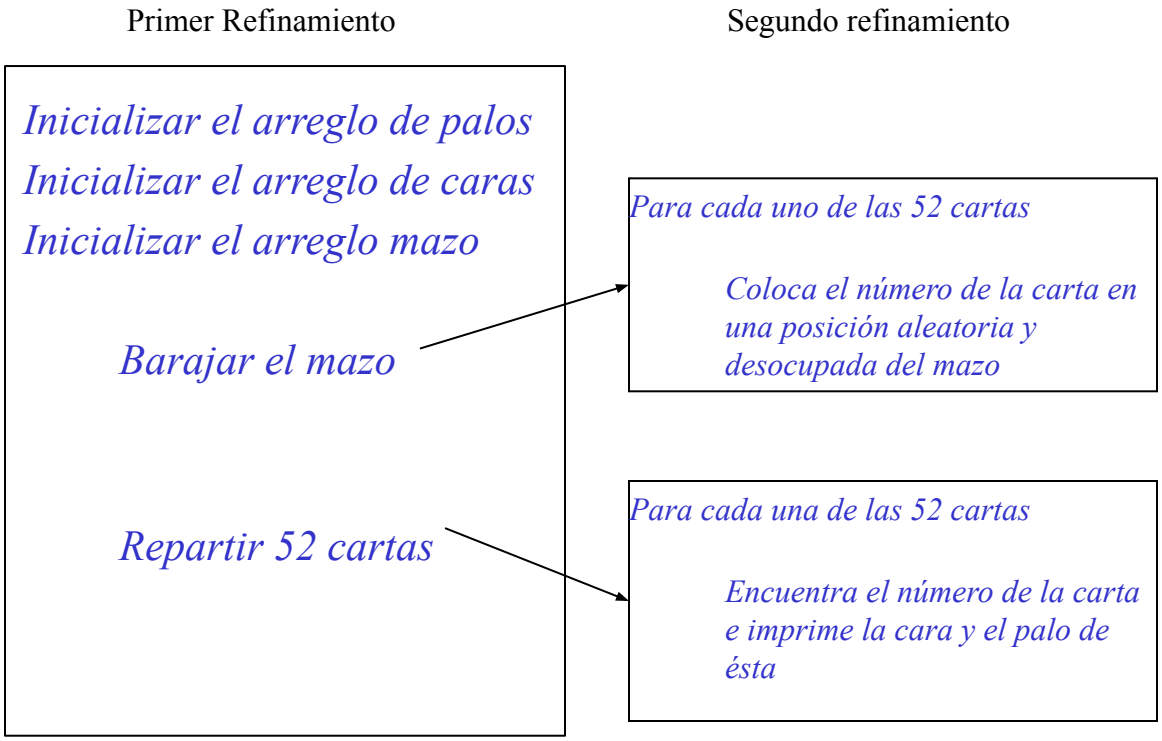
Inicializar el arreglo de palos
Inicializar el arreglo de caras
Inicializar el arreglo mazo

Barajar el mazo

Repartir 52 cartas

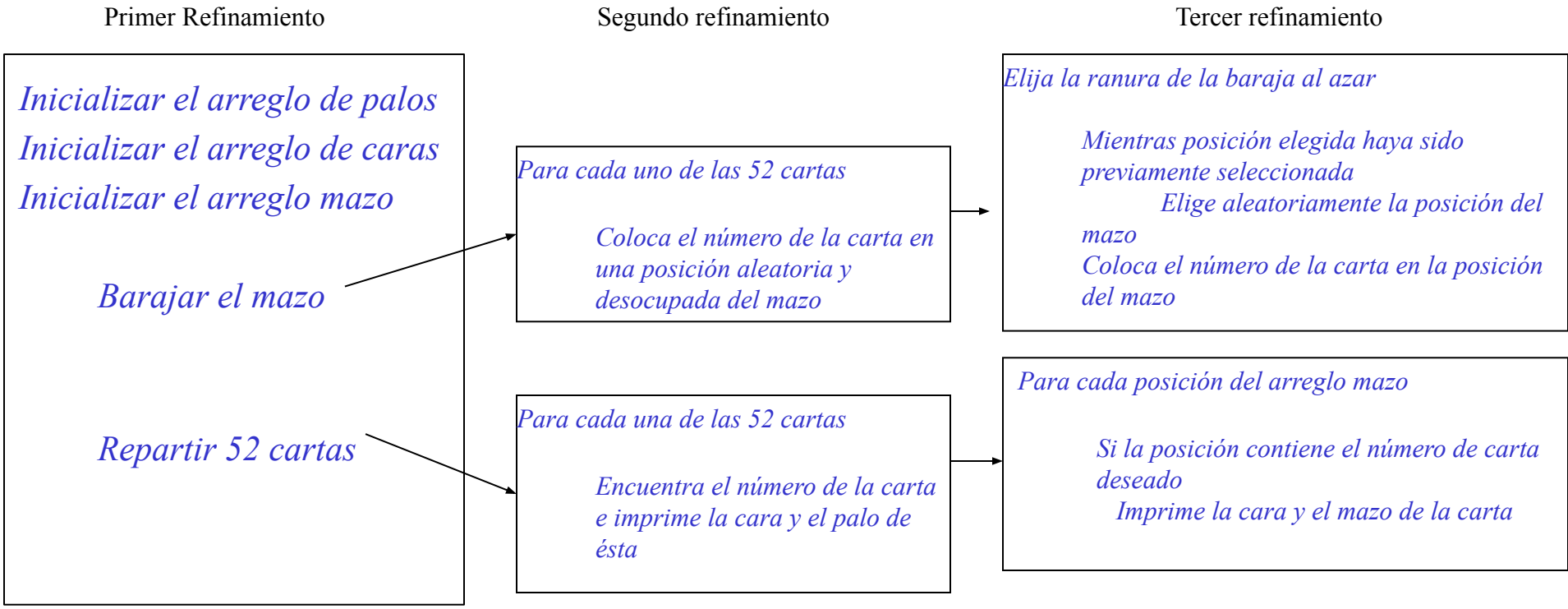
7.10 Caso de Estudio: Un simulacro de barajado y reparto de cartas

- Pseudocodigo - Top level: *Barajar y repartir 52 cartas*

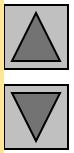


7.10 Caso de Estudio: Un simulacro de barajado y reparto de cartas

- Pseudocodigo - Top level: *Barajar y repartir 52 cartas*



```
1   /* Fig. 7.24: fig07 24.c
2      Programa para barajar y repartir cartas */
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <time.h>
6
7   void shuffle( int [][ 13 ] );
8   void deal( const int [][ 13 ], const char *[], const char *[]
9
10  int main()
11  {
12     const char *suit[ 4 ] =   /*inicializa el arreglo palo*/
13        { "Hearts", "Diamonds", "Clubs", "Spades" };
14     const char *face[ 13 ] =  /*inicializa el arreglo cara*/
15        { "Ace", "Deuce", "Three", "Four",
16          "Five", "Six", "Seven", "Eight",
17          "Nine", "Ten", "Jack", "Queen", "King" };
18     int deck[ 4 ][ 13 ] = { 0 }; /*inicializa el arreglo mazo*/
19
20     srand( time( 0 ) );//semilla del generador de números aleatorios
21
22     shuffle( deck );
23     deal( deck, face, suit );
24
25     return 0; /* indica terminación exitosa */
26  }
27  /* baraja las cartas del mazo */
28  void shuffle( int wDeck[][ 13 ] )
29  {
30     int row, column, card;//número de fila, columna, contador
31    /* elige aleatoriamente un espacio para cada carta */
32     for ( card = 1; card <= 52; card++ ) {
```



Outline

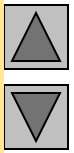
1. Inicializa `suit` y arreglo `face`

1.1 Inicializa arreglo `deck`

2. Llama a función `shuffle`

2.1 Llama a función `deal`

3. Define funcion



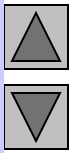
Outline

3. Define funciones

```
33          do {
34             row = rand() % 4;
35             column = rand() % 13;
36          } while( wDeck[ row ][ column ] != 0 );
37
38          wDeck[ row ][ column ] = card;
39       }
40    }
41    /* reparte las cartas del mazo */
42    void deal( const int wDeck[][ 13 ], const char *wFace[],
43               const char *wSuit[] )
44    {
45       int card, row, column;
46
47       for ( card = 1; card <= 52; card++ )
48
49          for ( row = 0; row <= 3; row++ )
50
51             for ( column = 0; column <= 12; column++ )
52
53                if ( wDeck[ row ][ column ] == card )
54                   printf( "%5s of %-8s%c",
55                   wFace[ column ], wSuit[ row ],
56                   card % 2 == 0 ? '\n' : '\t' );
57    }
```

Los números 1-52 son aleatoriamente ubicados dentro del arreglo **deck**.

Busca **deck** para el número **card**, entonces impime la **face** y **suit**.



Outline

Program Output

```
  Six of Clubs          Seven of Diamonds
  Ace of Spades           Ace of Diamonds
  Ace of Hearts         Queen of Diamonds
Queen of Clubs          Seven of Hearts
  Ten of Hearts         Deuce of Clubs
  Ten of Spades         Three of Spades
  Ten of Diamonds        Four of Spades
 Four of Diamonds         Ten of Clubs
  Six of Diamonds         Six of Spades
Eight of Hearts         Three of Diamonds
 Nine of Hearts         Three of Hearts
Deuce of Spades           Six of Hearts
 Five of Clubs          Eight of Clubs
Deuce of Diamonds       Eight of Spades
 Five of Spades          King of Clubs
 King of Diamonds        Jack of Spades
Deuce of Hearts         Queen of Hearts
  Ace of Clubs           King of Spades
Three of Clubs           King of Hearts
 Nine of Clubs           Nine of Spades
 Four of Hearts         Queen of Spades
Eight of Diamonds        Nine of Diamonds
 Jack of Diamonds       Seven of Clubs
 Five of Hearts          Five of Diamonds
 Four of Clubs           Jack of Hearts
 Jack of Clubs          Seven of Spades
```

7.11 Punteros a Funciones

- Puntero a función
  - Contiene la dirección de la función
  - De manera similar a como el nombre de arreglo es la dirección del primer elemento
  - El nombre de la función es la dirección inicial del código que define la función

- Los punteros funciones puede ser
  - Pasado a funciones
  - Contenidos en arreglos
  - Asignados a otros punteros funciones

7.11 Punteros a Funciones (II)

- Ejemplo: bubblesort
  - Función **bubble** toma un puntero función
    - **bubble** llama a this función auxiliar
    - este determina el orden ascendente o descendente
  - El argumento en **bubblesort** para la función puntero:

    ```
    bool ( *compare )( int, int )
    ```

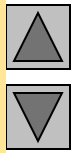
    dice **bubblesort** espera un puntero a función que toma dos **int**s y retorna un **bool**..
  - Si se omitieron los paréntesis:

    ```
    bool *compare( int, int )
    ```

    - Se declara una función que recibe dos **int**sy retorna un punto a **bool**

```c
/* Fig. 7.26: fig07 26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10
void bubble( int [], const int, int (*)( int, int ) );
int ascending( int, int );
int descending( int, int );

int main()
{

   int order,
       counter,
       a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "Enter 1 to sort in ascending order,\n"
           "Enter 2 to sort in descending order: " );
   scanf( "%d",  &order );
   printf( "\nData items in original order\n" );

   for ( counter = 0; counter < SIZE; counter++ )
      printf( "%5d", a[ counter ] );

   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      printf( "\nData items in ascending order\n" );
   }
   else {
      bubble( a, SIZE, descending );
      printf( "\nData items in descending order\n" );
   }

```

Note los parámetros de la función



Outline

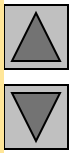
1. Inicializa arreglo.

2. Solicitud de ordenación ascendente o descendente.

2.1 Coloca el puntero de función apropiado en bubblesort.

2.2 Lalla a `bubble`.

3. Imprime resultados.



Outline

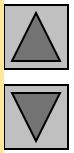
3.1 Define functions.

```
33      for ( counter = 0; counter < SIZE; counter++ )
34         printf( "%5d", a[ counter ] );
35
36      printf( "\n" );
37
38      return 0;
39   }
40
41   void bubble( int work[], const int size,
42                int (*compare)( int, int ) )
43   {
44      int pass, count;
45
46      void swap( int *, int * );
47
48      for ( pass = 1; pass < size; pass++ )
49
50         for ( count = 0; count < size - 1; count++ )
51
52            if ( (*compare)( work[ count ], work[ count + 1 ] ) )
53               swap( &work[ count ], &work[ count + 1 ] );
54   }
55
56   void swap( int *element1Ptr, int *element2Ptr )
57   {
58      int temp;
59
60      temp = *element1Ptr;
61      *element1Ptr = *element2Ptr;
62      *element2Ptr = temp;
63   }
64
```

ascending y **descending** retorna **true** o **false**. **bubble** llama a **swap** si la función llamada retorna **true**.

Observe cómo se llaman los punteros de la función usando el operador de desreferenciación. El * no es necesario, pero hace hincapié en que "comparar" es un puntero de función y no una función.

```
65   int ascending( int a, int b )
66   {
67      return b < a;   /* swap if b is less than a */
68   }
69
70   int descending( int a, int b )
71   {
72      return b > a;   /* swap if b is greater than a */
73   }
```



Outline

3.1 Define funciones.

Salida de Programa

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in ascending order
   2   4   6   8  10  12  37  45  68  89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2   6   4   8  10  12  89  68  45  37
Data items in descending order
  89  68  45  37  12  10   8   6   4   2
```

Cap. 7 - Punteros

Aguije!!!
Preguntas