ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

Lecture 05

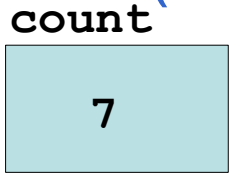
POINTERS

Pointers

- Powerful, but difficult to master
- Simulate call-by-reference
- Close relationship with arrays and strings

Pointer Variable vs. Normal Variable

- Normal variables contain a specific value (direct reference)
- Pointer variables
  - Contain memory addresses as their values
  - Pointers contain *address* of a variable that has a specific value (indirect reference)
  - Indirection - referencing a pointer value



Declarations and Initialization

- Pointer declarations
  - `*` used with pointer variables

    ```
    int *myPtr;
    ```
  - Declares a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers, multiple `*`

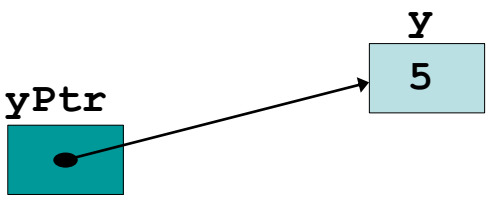
    ```
    int *myPtr1, *myPtr2;
    ```
  - Can declare pointers to any data type
- Initialize pointers to `0`, `NULL`, or an address
  - `0` or `NULL` - points to nothing (`NULL` preferred)

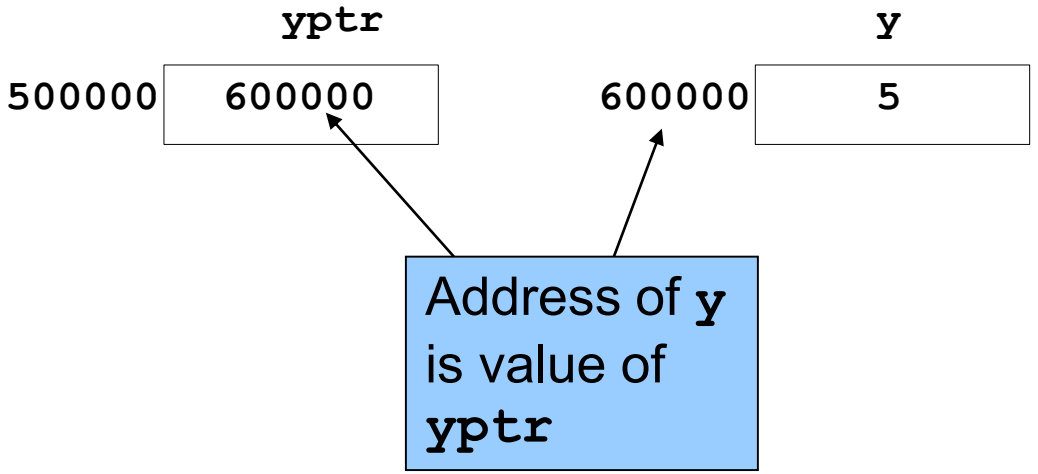
Pointer Operators

- `&` (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;  //yPtr gets address of y
```

  - `yPtr` "points to" `y`





Pointer Operators

- `*` (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand *points* to
    `*yptr` returns `y` (because `yptr` points to `y`)
  - `*` can be used for assignment
    - Returns alias to an object

```
*yptr = 7; // changes y to 7
```

Pointer Operators

- * and & are inverses
  - They cancel each other out

```
*&yptr  ->  * (&yptr) -> * (address of yptr)-> returns
  alias of what operand points to -> yptr

&*yptr -> &(*yptr) -> &(y) -> returns address of y,
  which is yptr -> yptr
```

```c
/* Fig. 7.4: fig07_04.c
   Using the & and * operators */
#include <stdio.h>

int main()
{
   int a;        /* a is an integer */
   int *aPtr;    /* aPtr is a pointer to an integer */

   a = 7;
   aPtr = &a;    /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr );

   printf( "\n\nShowing that * and & are inverses of "
           "each other.\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0;
}
```

The address of **a** is the value of **aPtr**.

The * operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how * and & are inverses

```
The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Showing that * and & are inverses of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

Calling Functions by Reference

- Call by reference with pointer arguments
  - Pass address of argument using **&** operator
  - Allows you to change actual location in memory
  - Arrays are not passed with **&** because the array name is already a pointer
- ***** operator
  - Used as alias/nickname for variable inside of function

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

***number** used as nickname for the variable passed

```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call-by-reference
3     with a pointer argument */
4
5  #include <stdio.h>
6
7  void cubeByReference( int * );    /* proto
8
9  int main()
10 {
11    int number = 5;
12
13    printf( "The original value of number is %d", number );
14    cubeByReference( &number );
15    printf( "\nThe new value of number is %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 }
```

Notice how the address of **number** is given - **cubeByReference** expects a pointer (an address of a variable).

Inside **cubeByReference**, ***nPtr** is used (***nPtr** is **number**).

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

Using the Const Qualifier with Pointers

- `const` qualifier - variable cannot be changed
  - Good idea to have `const` if function does not need to change a variable
  - Attempting to change a `const` is a compiler error
- `const` pointers - point to same memory location
  - Must be initialized when declared

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

  - Type `int *const` - constant pointer to an `int`

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

  - Regular pointer to a `const int`

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

  - `const` pointer to a `const int`

```c
/* Fig. 7.13: fig07_13.c
   Attempting to modify a constant pointer to
   non-constant data */

#include <stdio.h>

int main()
{
   int x, y;

   int * const ptr = &x; /* ptr is a constant pointer to an
                            integer. An integer can be modified
                            through ptr, but ptr always points
                            to the same memory location. */
   *ptr = 7;
   ptr = &y;

   return 0;
}
```

Changing ***ptr** is allowed - **x** is not a constant.

Changing **ptr** is an error - **ptr** is a constant pointer.

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

Swap function Using Call-by-reference

- Implement swap function
  - Swap two elements
  - **swap** function must receive address (using **&**) of array elements
    - Array elements have call-by-value default
  - Using pointers and the * operator, **swap** can switch array elements

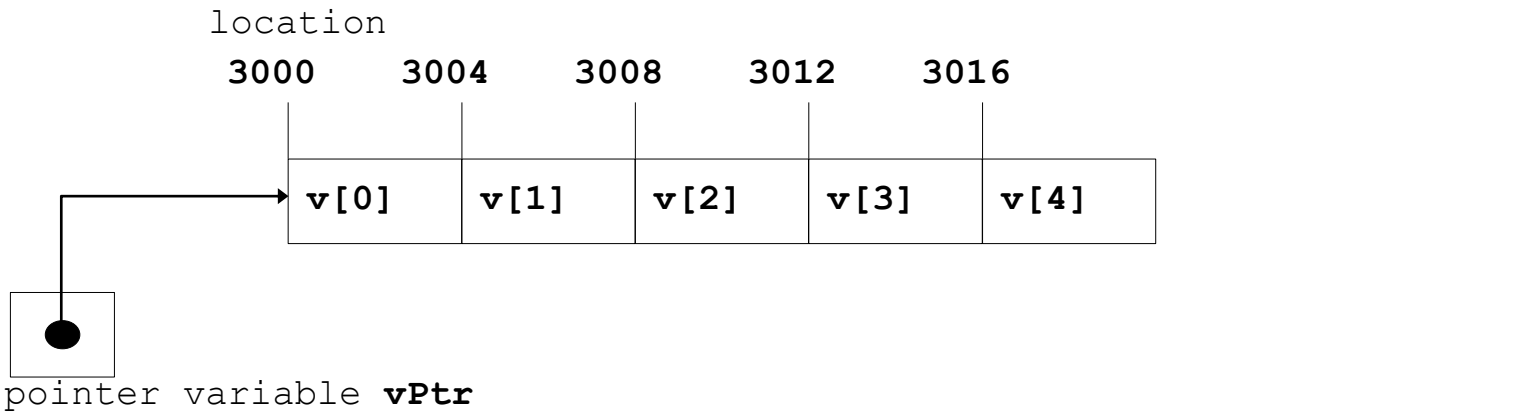
```
void swap( int *element1Ptr, int *element2Ptr )
{
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}
```

Pointer Expressions and Pointer Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or −−)
  - Add an integer to a pointer(+ or += , − or −=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Pointer Expressions and Pointer Arithmetic

- 5 element `int` array on machine with 4 byte `int`s
  - `vPtr` points to first element `v[0]`
    at location `3000`. (`vPtr = 3000`)
  - `vPtr +=2;` sets `vPtr` to `3008`
    - `vPtr` points to `v[2]` (incremented by 2), but machine has 4 byte `int`s.



Pointer Expressions and Pointer Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other.

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

- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to 0

Pointer Expressions and Pointer Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

The Relationship Between Pointers and Arrays

- Arrays and pointers closely related
  - Array name like a constant pointer
  - Pointers can do array subscripting operations

- Declare an array `b[5]` and a pointer `bPtr`

```
bPtr = b;
```

Array name actually a address of first element

OR

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

Explicitly assign `bPtr` to address of first element

The Relationship Between Pointers and Arrays

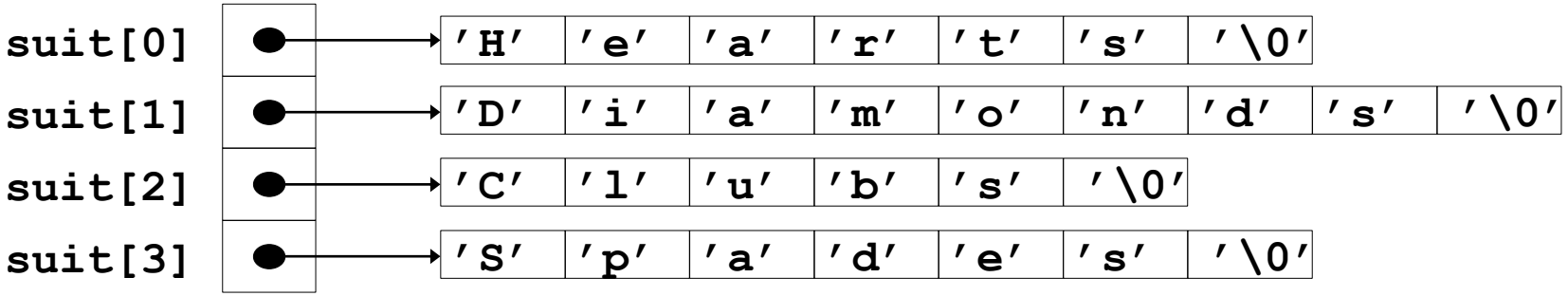
- Element `b[n]`
  - can be accessed by `*( bPtr + n )`
  - `n` - offset (pointer/offset notation)
  - Array itself can use pointer arithmetic.
    `b[3]` same as `*(b + 3)`
  - Pointers can be subscripted (pointer/subscript notation)
    `bPtr[3]` same as `b[3]`

Arrays of Pointers

- Arrays can contain pointers - array of strings

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

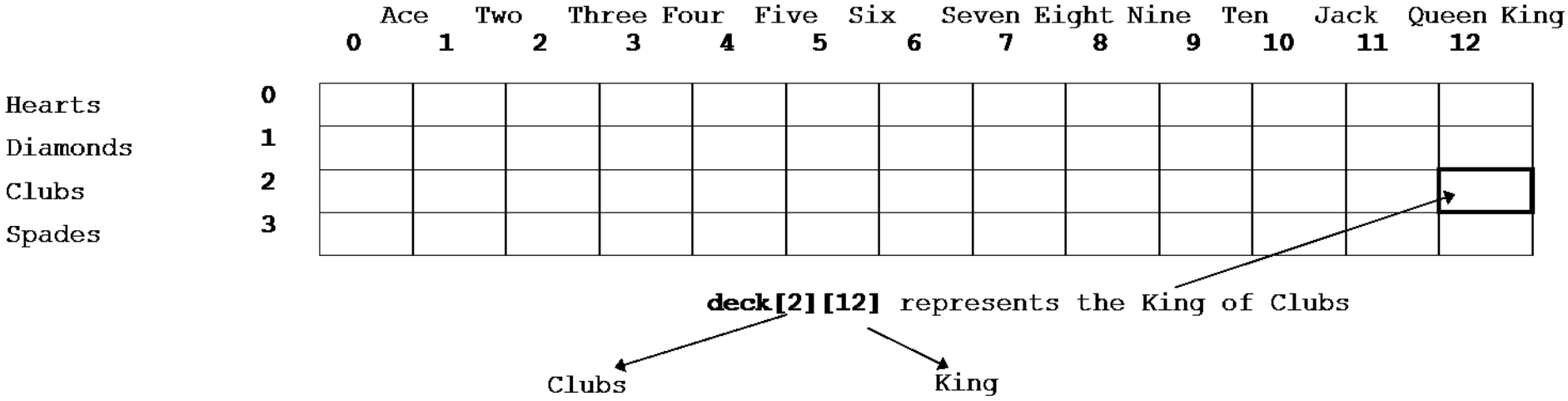
  - String: pointer to first character
  - **`char *`** - each element of **`suit`** is a pointer to a **`char`**
  - Strings not actually in array - only *pointers* to string in array



- **`suit`** array has a fixed size, but strings can be of any size.

Case Study: A Card Shuffling and Dealing Simulation

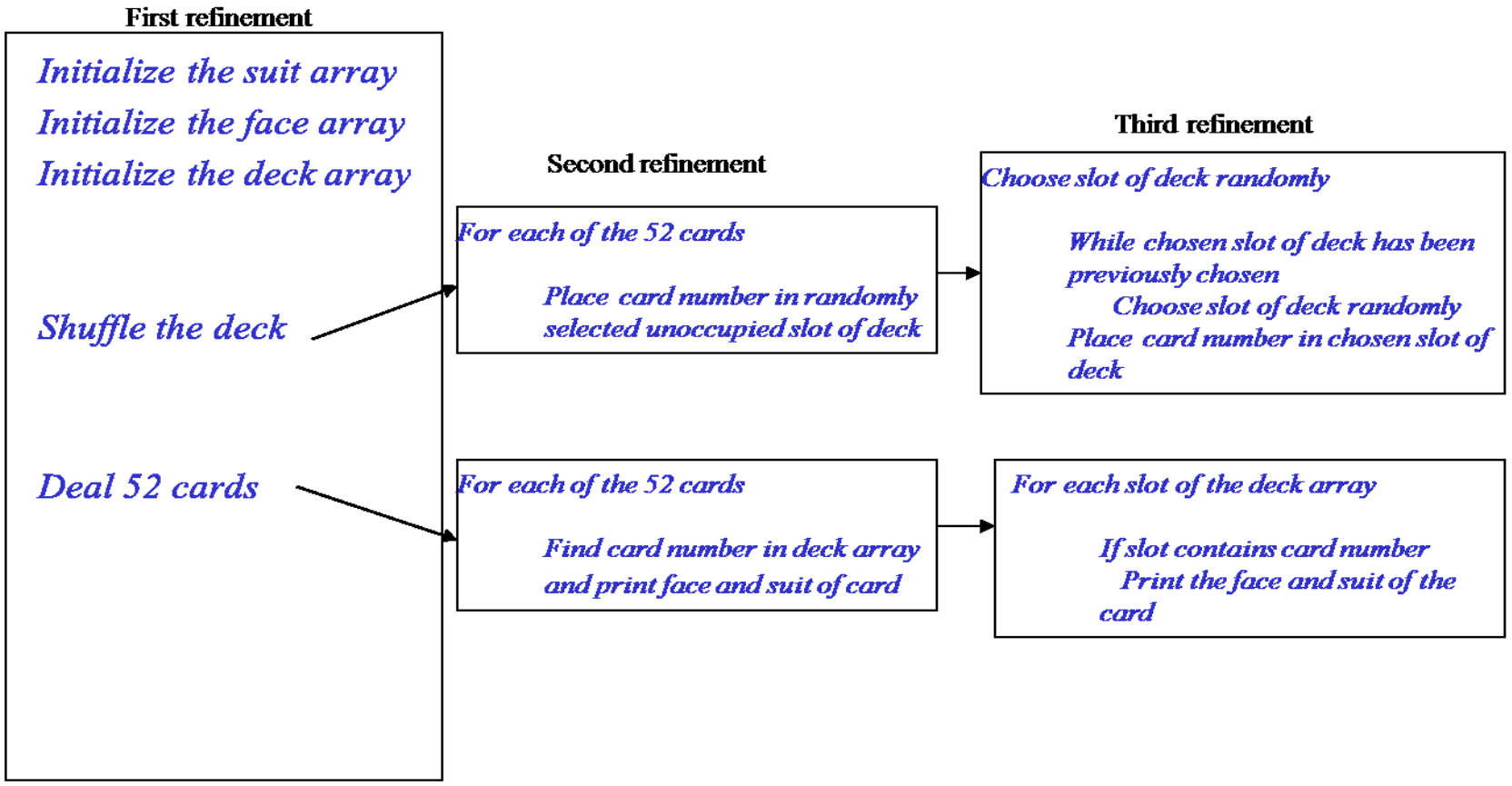
- Card shuffling program
  - Use array of pointers to strings
  - Use double scripted array (suit, face)



  - The numbers 1-52 go into the array - this is the order they are dealt

Case Study: A Card Shuffling and Dealing Simulation

- Pseudocode - Top level: *Shuffle and deal 52 cards*



```c
/* Fig. 7.24: fig07_24.c
   Card shuffling dealing program */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void shuffle( int [][ 13 ] );
void deal( const int [][ 13 ], const char *[], const char *[] );

int main()
{
   const char *suit[ 4 ] =
      { "Hearts", "Diamonds", "Clubs", "Spades" };
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };
   int deck[ 4 ][ 13 ] = { 0 };

   srand( time( 0 ) );

   shuffle( deck );
   deal( deck, face, suit );

   return 0;
}

void shuffle( int wDeck[][ 13 ] )
{
   int row, column, card;

   for ( card = 1; card <= 52; card++ ) {
```

```
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
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 }
```

The numbers 1-52 are randomly placed into the **deck** array.

Searches **deck** for the **card** number, then prints the **face** and **suit**.

Pointers to Functions

- Pointer to function
  - Contains address of function
  - Similar to how array name is address of first element
  - Function name is starting address of code that defines function

- Function pointers can be
  - Passed to functions
  - Stored in arrays
  - Assigned to other function pointers

Pointers to Functions

- Example: bubblesort
  - Function **`bubble`** takes a function pointer
    - **`bubble`** calls this helper function
    - this determines ascending or descending sorting
  - The argument in **`bubblesort`** for the function pointer:

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

    tells **`bubblesort`** to expect a pointer to a function that takes two **`int`**s and returns a **`bool`**.
  - If the parentheses were left out:

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

    - Declares a function that receives two integers and returns a pointer to 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" );
   }

```

Notice the function pointer parameter.

- 1. Initialize array.
- 2. Prompt for ascending or descending sorting.
- 2.1 Put appropriate function pointer into bubblesort.
- 2.2 Call `bubble`.
- 3. Print results.

```
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 and **descending** return **true** or **false**. **bubble** calls **swap** if the function call returns **true**.

Notice how function pointers are called using the dereferencing operator. The * is not required, but emphasizes that **compare** is a function pointer and not a function.

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

```
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
```