Chapter 7 - Pointers

Outline

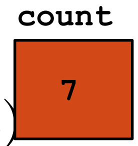
7.1 Introduction
7.2 Pointer Variable Declarations and Initialization
7.3 Pointer Operators
7.4 Calling Functions by Reference
7.5 Using the Const Qualifier with Pointers
7.6 Bubble Sort Using Call by Reference
7.7 Pointer Expressions and Pointer Arithmetic
7.8 The Relationship between Pointers and Arrays
7.9 Arrays of Pointers
7.10 Case Study: A Card Shuffling and Dealing Simulation
7.11 Pointers to Functions

7.1 Introduction

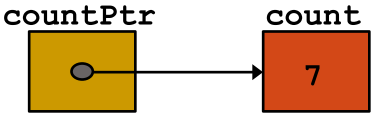
- Pointers
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

7.2 Pointer Variable Declarations and Initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain *address* of a variable that has a specific value (indirect reference)



  - Indirection - referencing a pointer value

7.2 Pointer Variable Declarations and Initialization (II)

- Pointer declarations
  - **`*`** used with pointer variables

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

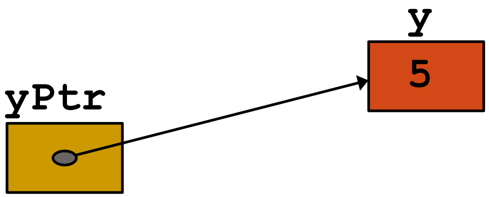
    ```
    int *myPtr1, *myPtr2;
    ```
  - Can declare pointers to any data type
  - Initialize pointers to **`0`**, **`NULL`**, or an address
    - **`0`** or **`NULL`** - points to nothing (**`NULL`** preferred)

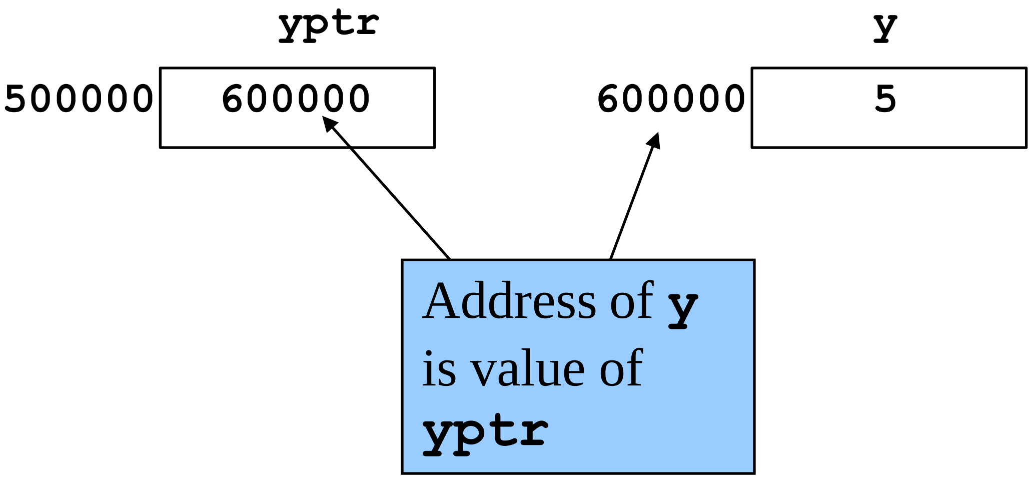
7.3 Pointer Operators

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

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

  - **yPtr** "points to" **y**





Address of **y** is value of **yptr**

7.3 Pointer Operators (II)

- **`*`** (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
    ```
  - Dereferenced pointer (operand of **`*`**) must be an *lvalue* (no constants)

7.3 Pointer Operators (III)

- * 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**.

Declare variables

ariables

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
Proving that * and & are complements of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

Program Output

7.4 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

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference
   with a pointer argument */

#include <stdio.h>

void cubeByReference( int * );    /* p

int main()
{
   int number = 5;

   printf( "The original value of number is %d", number );
   cubeByReference( &number );
   printf( "\nThe new value of number is %d\n", number );

   return 0;
}

void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
}
```

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**).

1. Function prototype - takes a pointer to an int.

1.1 Initialize variables

2. Call function

ne function

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

Program Output

7.5 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`
  - `x` can be changed, but not `*Ptr`

```
 1  /* Fig. 7.13: fig07_13.c
 2     Attempting to modify a constant pointer to
 3     non-constant data */
 4
 5  #include <stdio.h>
 6
 7  int main()
 8  {
 9     int x, y;
10
11     int * const ptr = &x; /* ptr is a constant pointer to an
12                              integer. An integer can be modified
13                              through ptr, but ptr always points
14                              to the same memory location. */
15     *ptr = 7;
16     ptr = &y;
17
18     return 0;
19  }
```

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

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

1. Declare variables

1.1 Declare `const` pointer to an `int`.

2. Change `*ptr` (which is `x`).

2.1 Attempt to change `ptr`.

3. Output

Program Output

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

7.6 Bubble Sort Using Call-by-reference

- Implement bubblesort using pointers
  - 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

- Psuedocode

```
Initialize array
    print data in original order
Call function bubblesort
    print sorted array
Define bubblesort
```

7.6 Bubble Sort Using Call-by-reference (II)

- **`sizeof`**
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if **`sizeof(int) = 4 bytes, then`**

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

    will print 40
- **`sizeof`** can be used with
  - Variable names
  - Type name
  - Constant values

```c
/* Fig. 7.15: fig07_15.c
   This program puts values into an array, sorts the values into
   ascending order, and prints the resulting array. */
#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( "Data items in original

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

   bubbleSort( a, SIZE );          /* sort the array */
   printf( "\nData items in ascending order\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 * );
```

`Bubblesort` gets passed the address of array elements (pointers). The name of an array is a pointer.

1. Initialize array

1.1 Declare variables

2. Print array

2.1 Call `bubbleSort`

2.2 Print array

```
   int pass, j;
   for ( pass = 0; pass < size - 1; pass++ )

      for ( j = 0; j < size - 1; j++ )

         if ( array[ j ] > array[ j + 1 ] )
            swap( &array[ j ], &array[ j + 1 ] );
}

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

3. Function definitions

```
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
```

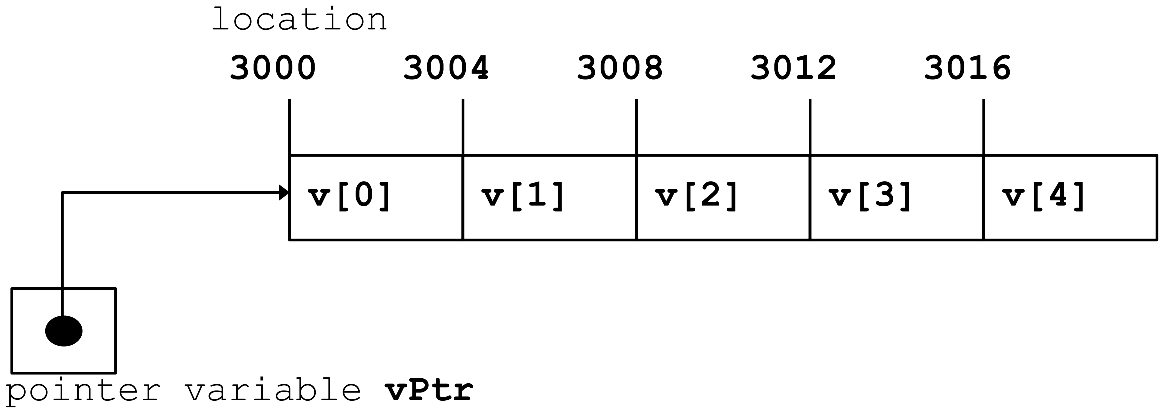
Program Output

7.7 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

7.7 Pointer Expressions and Pointer Arithmetic (II)

- 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.



7.7 Pointer Expressions and Pointer Arithmetic (III)

- 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**

7.7 Pointer Expressions and Pointer Arithmetic (IV)

- 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

7.8 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

7.8 The Relationship Between Pointers and Arrays (II)

- 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)

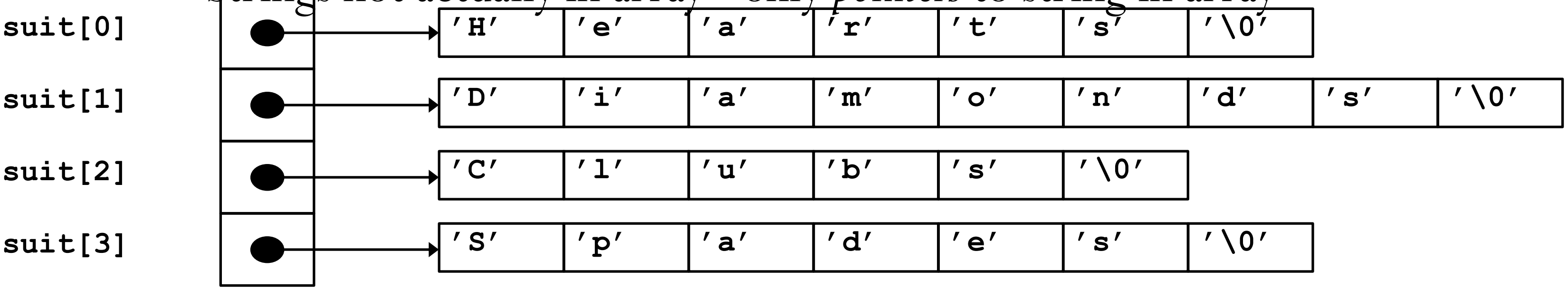
    `bPtr[3]` same as **`b[3]`**

7.9 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[0] 'H' 'e' 'a' 'r' 't' 's' '\0'

suit[1] 'D' 'i' 'a' 'm' 'o' 'n' 'd' 's' '\0'

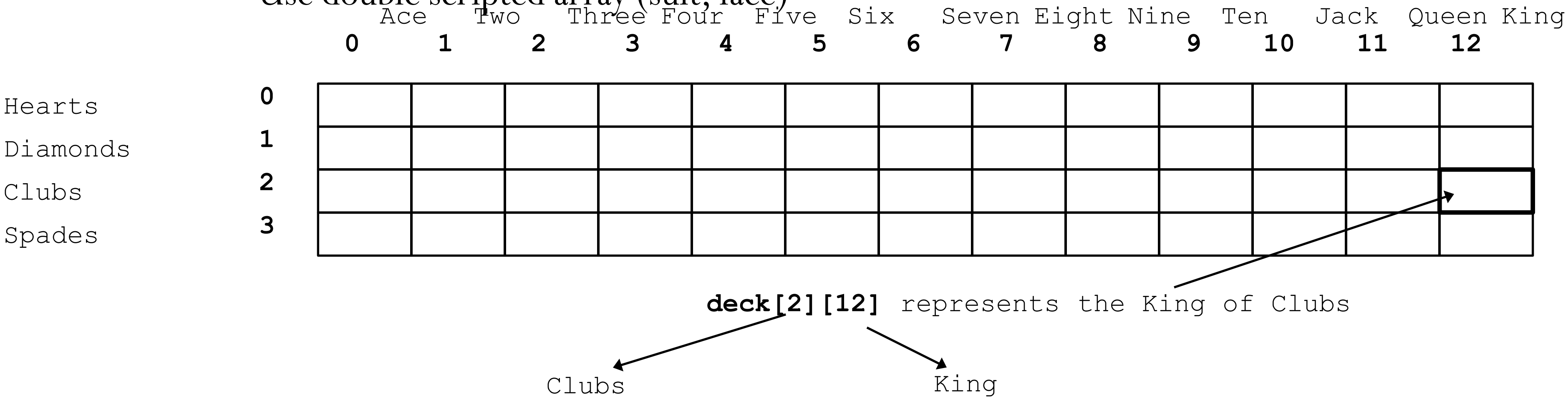
suit[2] 'C' 'l' 'u' 'b' 's' '\0'

suit[3] 'S' 'p' 'a' 'd' 'e' 's' '\0'

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

7.10 Case Study: A Card Shuffling and Dealing Simulation

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



Ace Two Three Four Five Six Seven Eight Nine Ten Jack Queen King

0 1 2 3 4 5 6 7 8 9 10 11 12

Hearts 0

Diamonds 1

Clubs 2

Spades 3

deck[2][12] represents the King of Clubs

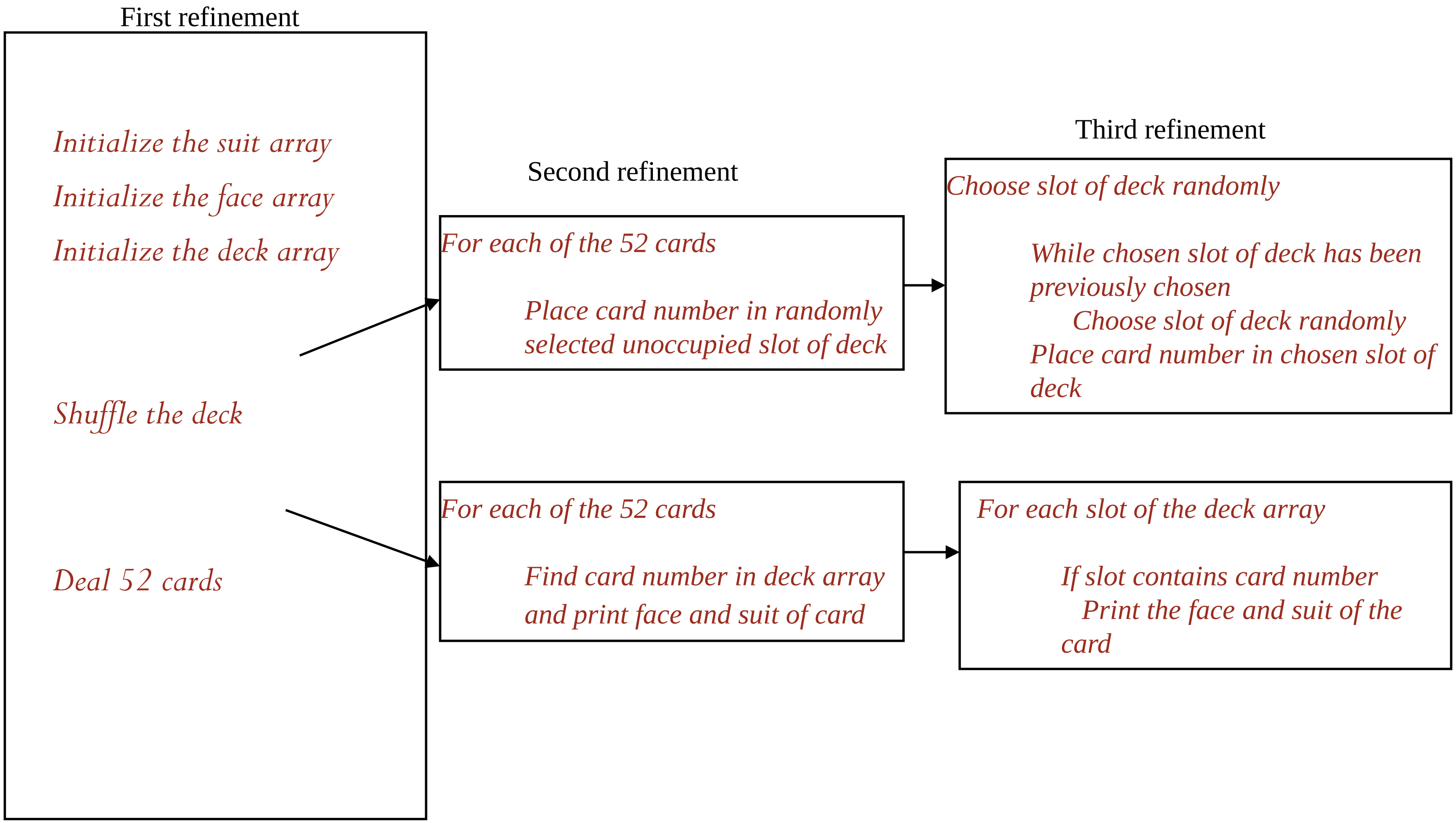
Clubs

King

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

7.10 Case Study: A Card Shuffling and Dealing Simulation

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



First refinement

```
Initialize the suit array
Initialize the face array
Initialize the deck array

Shuffle the deck

Deal 52 cards
```

Second refinement

```
For each of the 52 cards
    Place card number in randomly selected unoccupied slot of deck
```

```
For each of the 52 cards
    Find card number in deck array and print face and suit of card
```

Third refinement

```
Choose slot of deck randomly

    While chosen slot of deck has been previously chosen
        Choose slot of deck randomly
    Place card number in chosen slot of deck
```

```
For each slot of the deck array

    If slot contains card number
        Print the face and suit of the card
```

```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++ ) {
```

1. Initialize `suit` and `face` arrays

1.1 Initialize `deck` array

2. Call function `shuffle`

2.1 Call function `deal`

3. Define functions

```c
      do {
         row = rand() % 4;
         column = rand() % 13;
      } while( wDeck[ row ][ column ] != 0 );

      wDeck[ row ][ column ] = card;
   }
}

void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] )
{
   int card, row, column;

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

      for ( row = 0; row <= 3; row++ )

         for ( column = 0; column <= 12; column++ )

            if ( wDeck[ row ][ column ] == card )
               printf( "%5s of %-8s%c",
               wFace[ column ], wSuit[ row ],
               card % 2 == 0 ? '\n' : '\t' );
}
```

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

3. Define functions

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

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 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

7.11 Pointers to Functions (II)

- 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.

pt for ascending or descending sorting.

2.1 Put appropriate function pointer into bubblesort.

2.2 Call `bubble`.

3. Print results.

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

   printf( "\n" );

   return 0;
}

void bubble( int work[], const int size,
             int (*compare)( int, int ) )
{
   int pass, count;

   void swap( int *, int * );

   for ( pass = 1; pass < size; pass++ )

      for ( count = 0; count < size - 1; count++ )

         if ( (*compare)( work[ count ], work[ count + 1 ] ) )
            swap( &work[ count ], &work[ count + 1 ] );
}

void swap( int *element1Ptr, int *element2Ptr )
{
   int temp;

   temp = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = temp;
}

```

3.1 Define functions.

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.

```c
int ascending( int a, int b )
{
   return b < a;   /* swap if b is less than a */
}

int descending( int a, int b )
{
   return b > a;   /* swap if b is greater than a */
}
```

3.1 Define functions.

```
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
```

Program Output

```
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
```