Programming Languages -1 (Introduction to C)

pointers

Instructor: M.Fatih AMASYALI
E-mail:mfatih@ce.yildiz.edu.tr

Pointers

- A variable in a program is stored in a certain number of bytes at a particular memory location, or address, in the machine.
- Pointers allow us to manipulate these addresses explicitly.
- Two unary operators:
  - & operator – “address of”. Can be applied to any variable.
  - * operator – “information at”. Can be applied only to pointers.
- Pointer when declared points to an invalid location usually; so you must make it point to a valid one.

```
int a = 1, b = 2, *p;
char *char_p;
p = &a;
b = *p;
```

- An assignment like `char_p = &a;` is illegal, as the types do not match.

Constructs *not* to be pointed at

- Do not point at constants:
  - `int  *ptr;`
  - `*ptr = 3;  /* OK */`
  - `ptr = &3;  /* illegal */`
- Do not point at arrays
  - `int a[77];`
  - `void *ptr;`
  - `ptr = a;  /* OK */`
  - `ptr = &a;  /* illegal */`
- Do not point at expressions that are not variables.
  - `int k = 1, *ptr;`
  - `*ptr = k + 99;  /* OK */`
  - `ptr = &(k + 99);  /* illegal */`

“Call by reference” (not really)

- Pointers allow us to perform something similar to call-by-reference (we are passing pointers/references by value)
- “call-by-reference” allows a function to make changes to a variable that persist

```
void set_int_to_3( int *p )
{
      *p = 3;
}
```

```
void swap( int *p, int *q )
{
      int temp;
      temp = *p;
      *p = *q;
      *q = temp;
}
```

Example: Arrays as Function Arguments

(Array passed by reference, so changes to array persist)

```
int change_and_sum( int a[], int size )
{
      int i, sum = 0;
      a[0] = 100;
      for( i = 0; i < size; i++ )
            sum += a[i];
      return sum;
}

int main()
{
      int a[5] = {0, 1, 2, 3, 4};
      printf( "sum of a: %d\n",
            change_and_sum( a, 5 ));
      printf( "value of a[0]: %d\n", a[0] );
}
```

Arrays are not passed with & because the array name is already a pointer

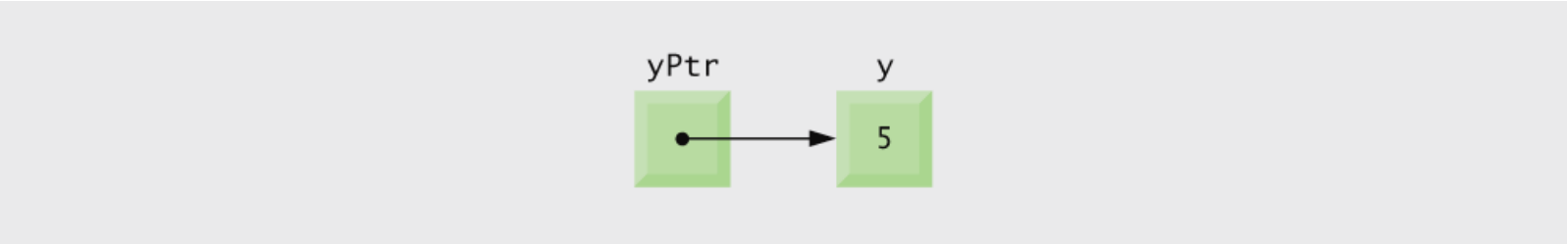
- Pointer definitions
  - Multiple pointers require using a * before each variable definition

```
int *myPtr1, *myPtr2;
```

  - Can define pointers to any data type

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;   /* yPtr gets address of y */
yPtr “points to” y
```



Graphical representation of a pointer pointing to an integer variable in memory.

Representation of **`y`** and **`yPtr`** in memory.

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

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

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

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

int main( void )
{
   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 complements of "
           "each other\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   return 0; /* indicates successful termination */

} /* end main */
```

If **aPtr** points to **a**, then **&a** and **aPtr** have the same value.

a and ***aPtr** have the same value

&*aPtr and ***&aPtr** have the same value

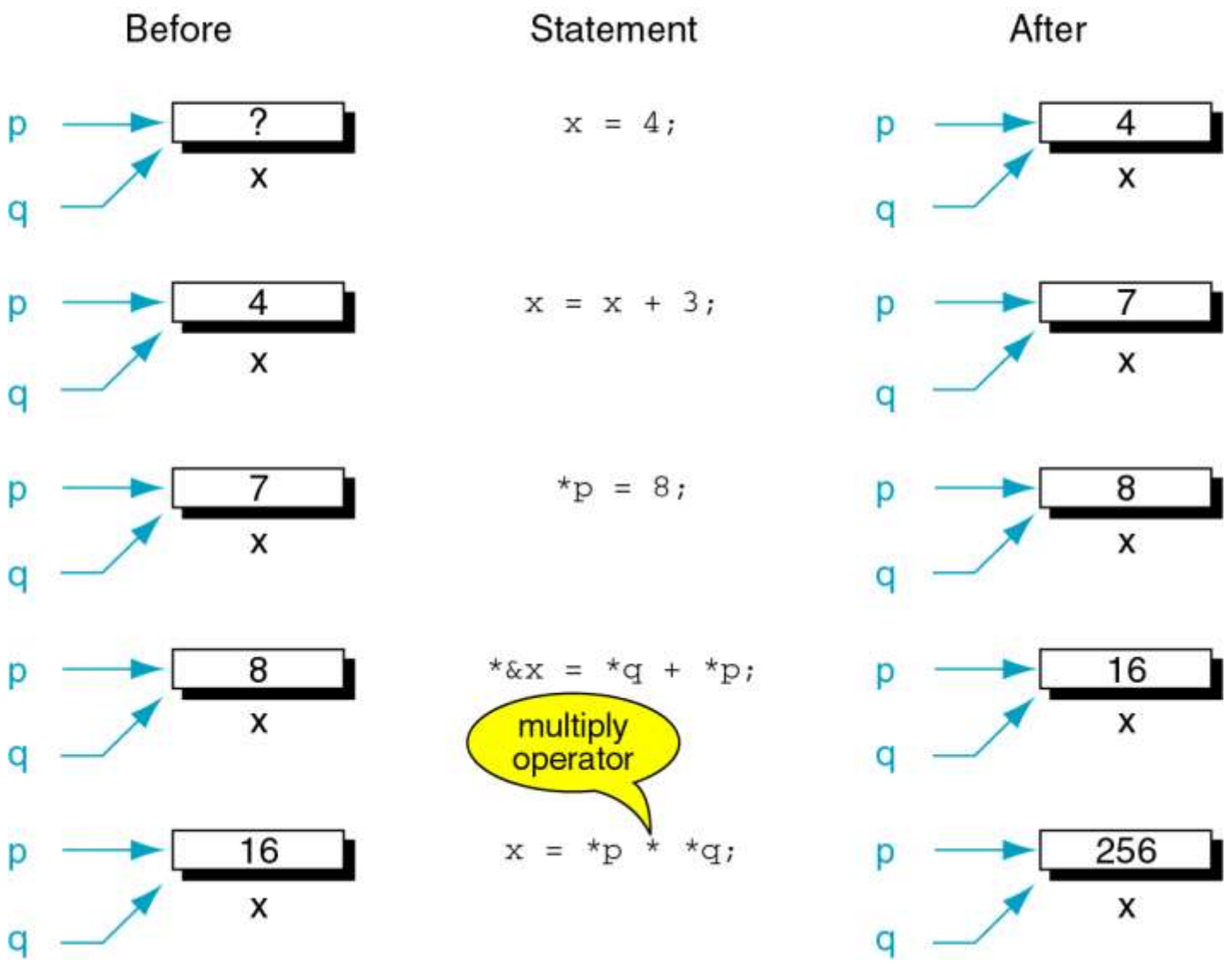
```
The address of a is 0012FF7C
The value of aPtr is 0012FF7C

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other.
&*aPtr = 0012FF7C
*&aPtr = 0012FF7C
```

örnek hafıza değişimleri

1) int a,*ptr,b,c,*d
2) a=25;
3) ptr=&a;
4) b=a;
5) c=*ptr
6) d=ptr;

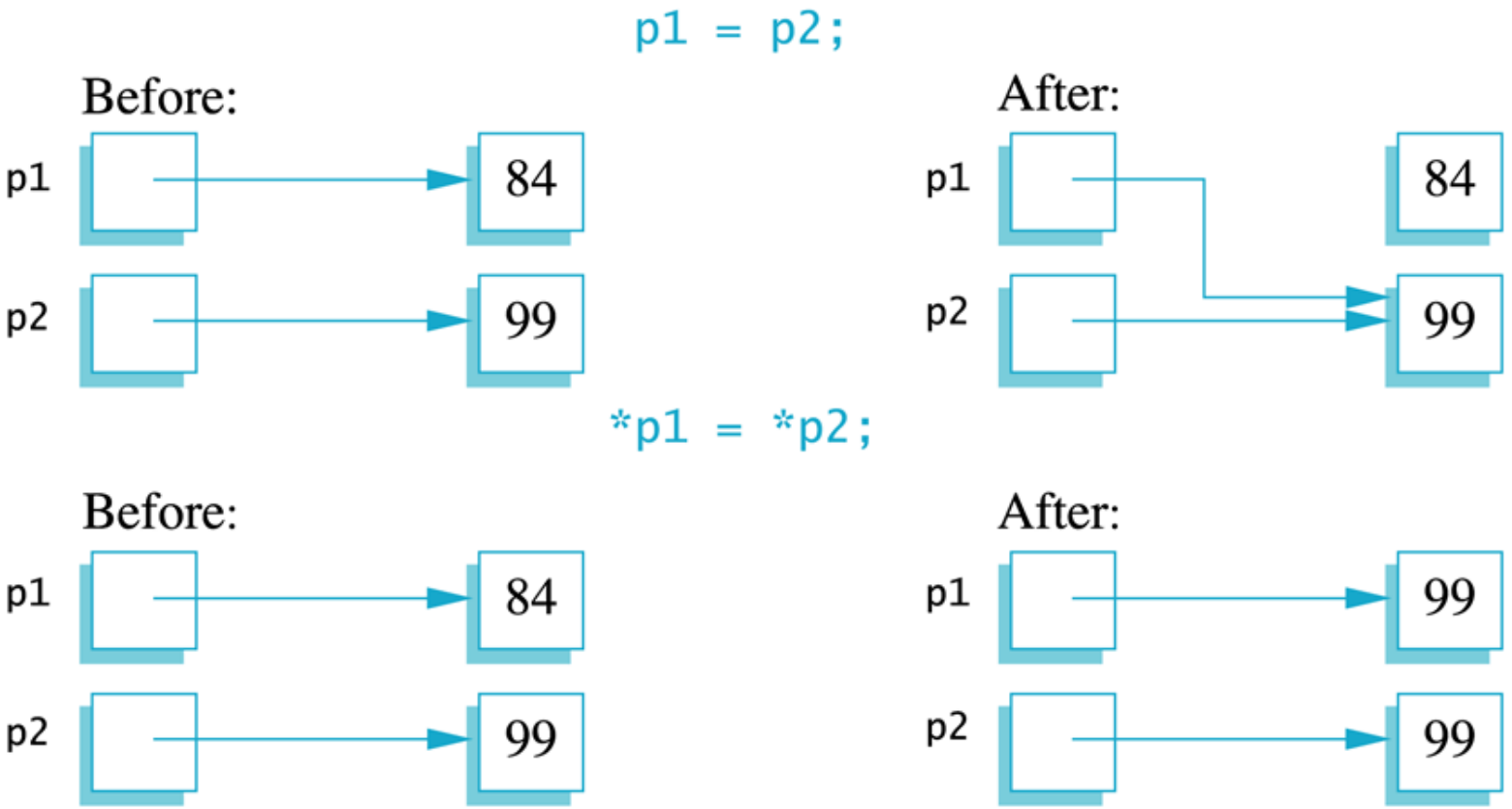
adres	**değişkenler**	**1**	**2**	**3**	**4**	**5**	**6**
1000	a	Boş	25	25	25	25	25
2000	ptr	Boş	Boş	1000	1000	1000	1000
3000	b	Boş	Boş	Boş	25	25	25
4000	c	Boş	Boş	Boş	Boş	25	25
5000	d	Boş	Boş	Boş	Boş	Boş	1000

```
int x;
int *p=&x;
int *q=&x;
```



Caution! Pointer assignments

Uses of the Assignment Operator



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

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

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

Portability Tip

- The number of bytes used to store a particular data type may vary between systems. When writing programs that depend on data type sizes and that will run on several computer systems, use `sizeof` to determine the number of bytes used to store the data types.

```c
/* Fig. 7.17: fig07_17.c
   Demonstrating the sizeof operator */
#include <stdio.h>

int main( void )
{
   char c;
   short s;
   int i;
   long l;
   float f;
   double d;
   long double ld;
   int array[ 20 ];  /* create array of 20 int elements */
   int *ptr = array; /* create pointer to array */

```

```c
   printf( "     sizeof c = %d\tsizeof(char)  = %d"
           "\n     sizeof s = %d\tsizeof(short) = %d"
           "\n     sizeof i = %d\tsizeof(int) = %d"
           "\n     sizeof l = %d\tsizeof(long) = %d"
           "\n     sizeof f = %d\tsizeof(float) = %d"
           "\n     sizeof d = %d\tsizeof(double) = %d"
           "\n    sizeof ld = %d\tsizeof(long double) = %d"
           "\n sizeof array = %d"
           "\n   sizeof ptr = %d\n",
          sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
          sizeof( int ), sizeof l, sizeof( long ), sizeof f,
          sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
          sizeof( long double ), sizeof array, sizeof ptr );

   return 0; /* indicates successful termination */

} /* end main */
```

```
     sizeof c = 1       sizeof(char)  = 1
     sizeof s = 2       sizeof(short) = 2
     sizeof i = 4       sizeof(int) = 4
     sizeof l = 4       sizeof(long) = 4
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 8       sizeof(long double) = 8
 sizeof array = 80
   sizeof ptr = 4
```

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

```
#include <stdio.h>
int main ()
{
double a=122.5,*c;
int b=10,*d;
c=&a; d=&b;
printf("a b c d degerleri   %f %d %p %p \n",a,b,c,d);
c++; d--;
printf("a b c d degerleri   %f %d %p %p \n",a,b,c,d);
}
```

```
a b c d degerleri   122.500000        10        0012FF84        0012FF7C
a b c d degerleri   122.500000        10        0012FF8C        0012FF78
```

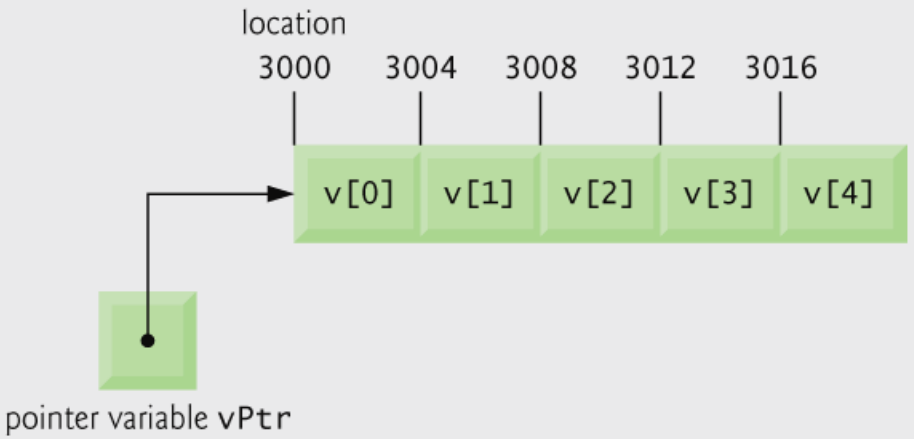
// integer i gösteren pointer 4 azalmış, double gösteren pointer 8 artmıştır.

Pointer ları tamsayılarla toplamak pointer ın gösterdiği değişken türüne göre bir artım sağlar. 3 ile toplanırsa 3*(tututuğu tipin boyutu) kadar artar. Örneğin yukarıdaki programda c++ yerine c+=3 denseydi 8*3=24 artardı ve 0012FF9C olurdu.

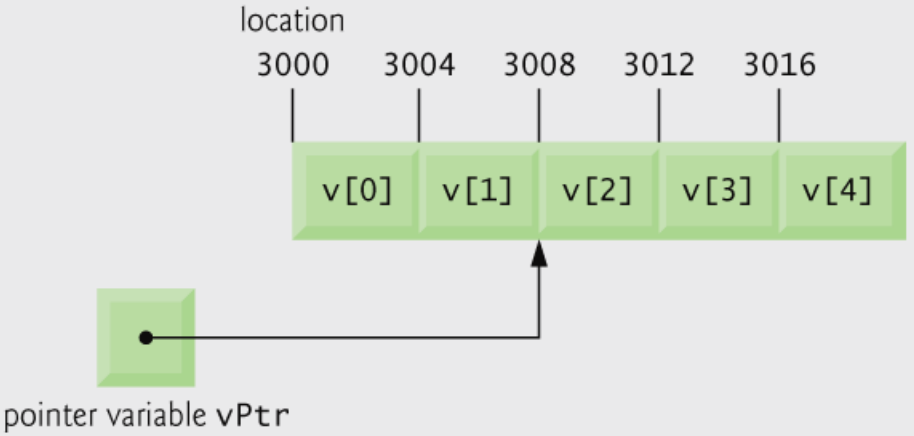
Pointer Expressions and Pointer Arithmetic

- 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 the machine has 4 byte `int`s, so it points to address 3008

Array **v** and a pointer variable **vPtr** that points to **v**.



The pointer **vPtr** after pointer arithmetic.



Pointer Expressions and Pointer Arithmetic

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

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

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element

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

Arrays and Pointers (a difference)

- An array is similar to a pointer
- However:

```
int  i, a[10], *p;
p=a;          /* equiv. to p=&a[0]; */
p++;          /* valid */
a++;          /* error! */
```

- The name of an array is not a variable, so the only operator you can apply to it is []
  - E.g. `a[i+3]`

Arrays and Pointers

- Assume `int i, a[10], *p;`
  - The type of `a` is "`int *`".
  - `a` is equivalent to `&a[0]`
  - `a + i` is equivalent to `&a[i]`
- Correspondingly,
  - `a[i]` is equivalent to `*(a + i)`
- In fact,
  - `p[i]` is equivalent to `*(p + i)`

```
for( p = a; p < &a[10]; p++ )
      sum += *p;
```

```
for( i = 0; i < 10; i++ )
      sum += *(a + i);
```

```
p = a;
for( i = 0; i < 10; i++ )
      sum += p[i];
```

```c
/* Fig. 7.20: fig07_20.cpp
   Using subscripting and pointer notations with arrays */

#include <stdio.h>

int main( void )
{
   int b[] = { 10, 20, 30, 40 }; /* initialize array b */
   int *bPtr = b;                /* set bPtr to point to array b */
   int i;                        /* counter */
   int offset;                   /* counter */

   /* output array b using array subscript notation */
   printf( "Array b printed with:\nArray subscript notation\n" );

   /* loop through array b */
   for ( i = 0; i < 4; i++ ) {
      printf( "b[ %d ] = %d\n", i, b[ i ] );
   } /* end for */

   /* output array b using array name and pointer/offset notation */
   printf( "\nPointer/offset notation where\n"
           "the pointer is the array name\n" );

   /* loop through array b */
   for ( offset = 0; offset < 4; offset++ ) {
      printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
   } /* end for */

```

Array subscript notation

Pointer/offset notation

```
30    /* output array b using bPtr and array subscript notation */
31    printf( "\nPointer subscript notation\n" );
32
33    /* loop through array b */
34    for ( i = 0; i < 4; i++ ) {
35       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
36    } /* end for */
37
38    /* output array b using bPtr and pointer/offset notation */
39    printf( "\nPointer/offset notation\n" );
40
41    /* loop through array b */
42    for ( offset = 0; offset < 4; offset++ ) {
43       printf( "*( bPtr + %d ) = %d\n", offset, *( bPtr + offset ) );
44    } /* end for */
45
46    return 0; /* indicates successful termination */
47
48 } /* end main */
```

Pointer subscript notation

Pointer offset notation

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40
```

(continued on next slide…)

(continued from previous slide…)

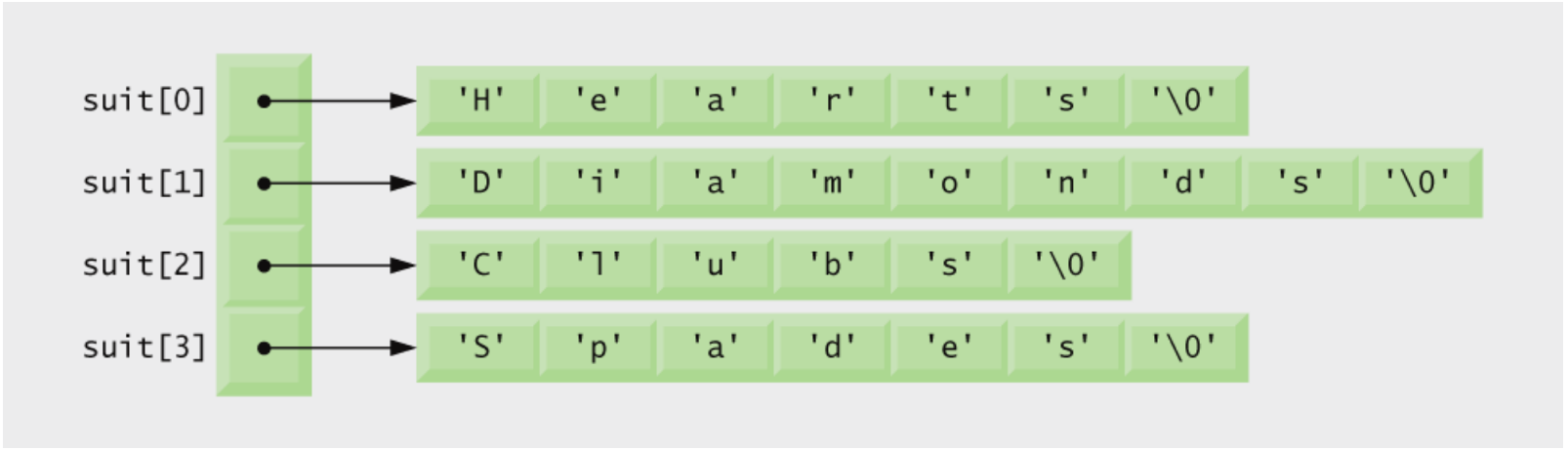
```
Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

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

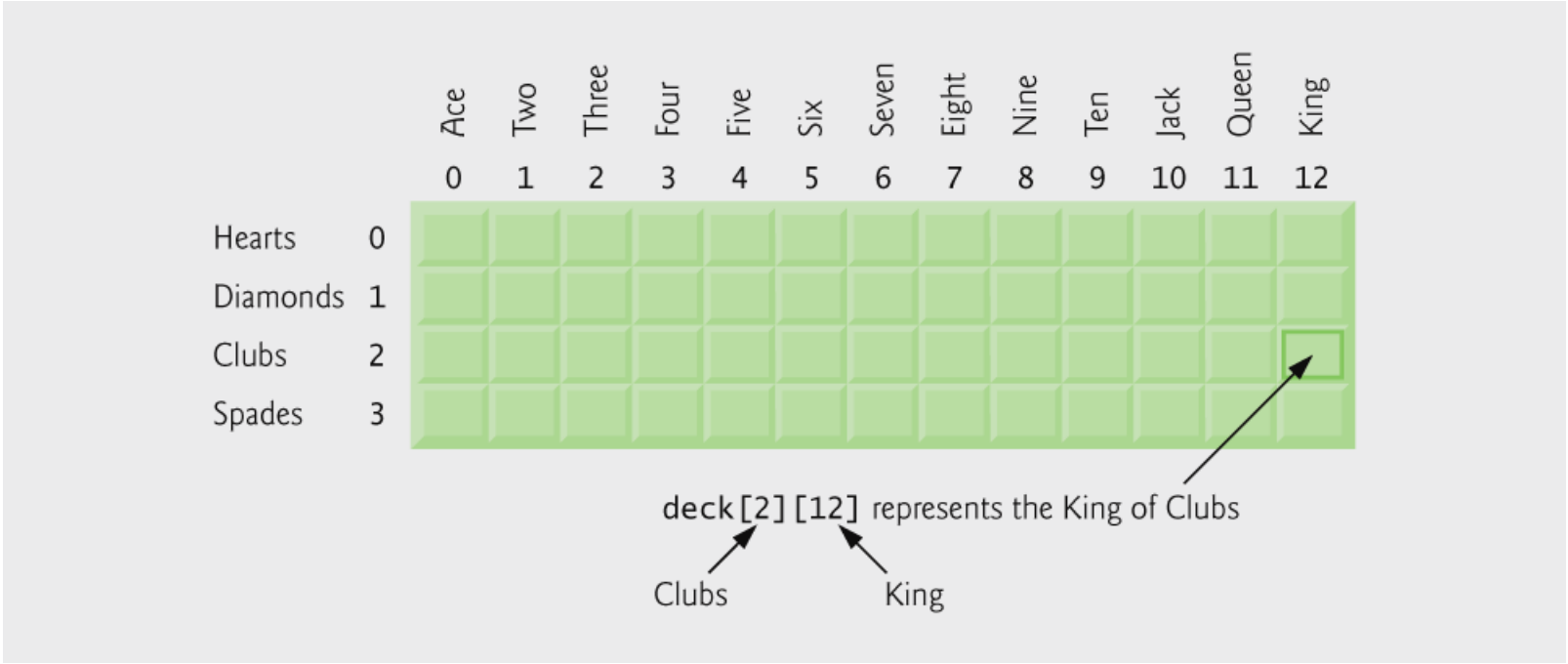
  - Strings are pointers to the first character
  - `char *` – each element of `suit` is a pointer to a `char`
  - The strings are not actually stored in the array `suit`, only pointers to the strings are stored



Graphical representation of the **`suit`** array.

Case Study: Card Shuffling and Dealing Simulation

- Card shuffling program
  - Use array of pointers to strings
  - Use double subscripted array (suit, face)
  - The numbers 1-52 go into the array
    - Representing the order in which the cards are dealt



Double-subscripted array representation of a deck of cards.

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

/* prototypes */
void shuffle( int wDeck[][ 13 ] );
void deal( const int wDeck[][ 13 ], const char *wFace[],
           const char *wSuit[] );

int main( void )
{
   /* initialize suit array */
   const char *suit[ 4 ] = { "Hearts", "Diamonds", "Clubs", "Spades" };

   /* initialize face array */
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };

```

`suit` and **`face`** arrays are arrays of pointers

```
23    /* initialize deck array */
24    int deck[ 4 ][ 13 ] = { 0 };
25
26    srand( time( NULL ) ); /* seed random-number generator */
27
28    shuffle( deck );
29    deal( deck, face, suit );
30
31    return 0; /* indicates successful termination */
32
33 } /* end main */
34
35 /* shuffle cards in deck */
36 void shuffle( int wDeck[][ 13 ] )
37 {
38    int row;    /* row number */
39    int column; /* column number */
40    int card;   /* counter */
41
42    /* for each of the 52 cards, choose slot of deck randomly */
43    for ( card = 1; card <= 52; card++ ) {
44
45       /* choose new random location until unoccupied slot found */
46       do {
47          row = rand() % 4;
48          column = rand() % 13;
49       } while( wDeck[ row ][ column ] != 0 ); /* end do...while */
50
```

do…while loop selects a random spot for each card

```
51       /* place card number in chosen slot of deck */
52       wDeck[ row ][ column ] = card;
53    } /* end for */
54
55 } /* end function shuffle */
56
57 /* deal cards in deck */
58 void deal( const int wDeck[][ 13 ], const char *wFace[],
59            const char *wSuit[] )
60 {
61    int card;   /* card counter */
62    int row;    /* row counter */
63    int column; /* column counter */
64
65    /* deal each of the 52 cards */
66    for ( card = 1; card <= 52; card++ ) {
67       /* loop through rows of wDeck */
68
69       for ( row = 0; row <= 3; row++ ) {
70
71          /* loop through columns of wDeck for current row */
72          for ( column = 0; column <= 12; column++ ) {
```

```
73
74             /* if slot contains current card, display card */
75             if ( wDeck[ row ][ column ] == card ) {
76                printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
77                   card % 2 == 0 ? '\n' : '\t' );
78             } /* end if */
79
80          } /* end for */
81
82       } /* end for */
83
84    } /* end for */
85
86 } /* end function deal */
```

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

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:

```
int ( *compare )( int a, int b )
```

    tells `bubblesort` to expect a pointer to a function that takes two `ints` and returns an `int`
  - If the parentheses were left out:

```
int *compare( int a, int b )
```

    - Defines a function that receives two integers and returns a pointer to a `int`

```c
/* Fig. 7.26: fig07_26.c
   Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

/* prototypes */
void bubble( int work[], const int size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order;   /* 1 for ascending order or 2 for descending order */
   int counter; /* counter */

   /* initialize array a */
   int 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" );

   /* output original array */
   for ( counter = 0; counter < SIZE; counter++ ) {
      printf( "%5d", a[ counter ] );
   } /* end for */

```

bubble function takes a function pointer as an argument

```
30    /* sort array in ascending order; pass function ascending as an
31       argument to specify ascending sorting order */
32    if ( order == 1 ) {
33       bubble( a, SIZE, ascending );
34       printf( "\nData items in ascending order\n" );
35    } /* end if */
36    else { /* pass function descending */
37       bubble( a, SIZE, descending );
38       printf( "\nData items in descending order\n" );
39    } /* end else */
40
41    /* output sorted array */
42    for ( counter = 0; counter < SIZE; counter++ ) {
43       printf( "%5d", a[ counter ] );
44    } /* end for */
45
46    printf( "\n" );
47
48    return 0; /* indicates successful termination */
49
50 } /* end main */
51
```

depending on the user's choice, the **`bubble`** function uses either the **`ascending`** or **`descending`** function to sort the array

```
52 /* multipurpose bubble sort; parameter compare is a pointer to
53    the comparison function that determines sorting order */
54 void bubble( int work[], const int size, int (*compare)( int a, int b ) )
55 {
56    int pass;  /* pass counter */
57    int count; /* comparison counter */
58
59    void swap( int *element1Ptr, int *element2ptr ); /* prototype */
60
61    /* loop to control passes */
62    for ( pass = 1; pass < size; pass++ ) {
63
64       /* loop to control number of comparisons per pass */
65       for ( count = 0; count < size - 1; count++ ) {
66
67          /* if adjacent elements are out of order, swap them */
68          if ( (*compare)( work[ count ], work[ count + 1 ] ) ) {
69             swap( &work[ count ], &work[ count + 1 ] );
70          } /* end if */
71
72       } /* end for */
73
74    } /* end for */
75
76 } /* end function bubble */
77
```

Note that what the program considers "out of order" is dependent on the function pointer that was passed to the **`bubble`** function

```
78 /* swap values at memory locations to which element1Ptr and
79    element2Ptr point */
80 void swap( int *element1Ptr, int *element2Ptr )
81 {
82    int hold; /* temporary holding variable */
83
84    hold = *element1Ptr;
85    *element1Ptr = *element2Ptr;
86    *element2Ptr = hold;
87 } /* end function swap */
88
89 /* determine whether elements are out of order for an ascending
90    order sort */
91 int ascending( int a, int b )
92 {
93    return b < a; /* swap if b is less than a */
94
95 } /* end function ascending */
96
97 /* determine whether elements are out of order for a descending
98    order sort */
99 int descending( int a, int b )
100 {
101    return b > a; /* swap if b is greater than a */
102
103 } /* end function descending */
```

Passing the **bubble** function **ascending** will point the program here

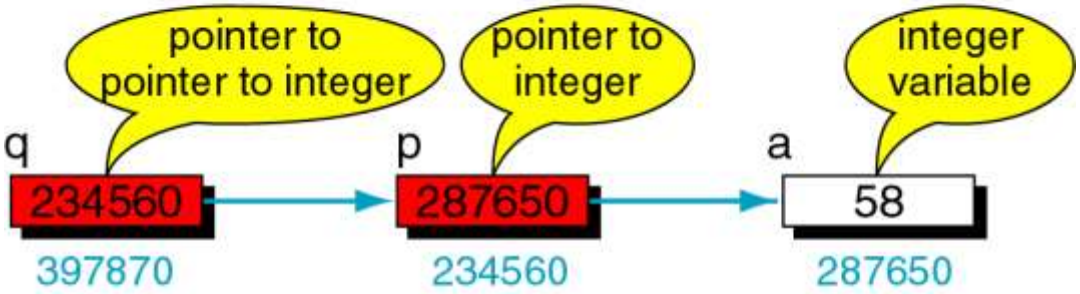
Passing the **bubble** function **descending** will point the program here

```
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
```

Pointers to Pointers

```
/* Local Declarations */
int       a;
int      *p;
int     **q;
```




```
/* Statements */
a = 58;
p = &a;
q = &p;
printf(" %3d",   a);
printf(" %3d",  *p);
printf(" %3d", **q);
```

Output: **58**

Output: **58**

Output: **58**

SORU: Aşağıdaki programın çıkışını yazınız.

```
int main(int argc, char* argv[])
{
int k[5]={50,80,60,14,32};
int *p,**pp;
p=k;
pp=&p;
printf("%d \n",*p);
p++;
printf("%d \n",*p);
printf("%d \n",**pp);
*p=43;
printf("%d \n",k[1]);
p+=2;
**pp+=32;
printf("%d \n",k[3]);
printf("%d \n",k[4]);
*pp-=1;
*p+=10;
printf("%d \n",*p);
printf("%d \n",**pp);
printf("%d \n",k[1]);
printf("%d \n",k[2]);
}
```

p=k → k nın başlangıç adresini p ye atar.
pp=&p; → p nin adresini pp ye atar
p++ → p nin değerini arttırır
*p=3 → p nin gösterdiği adresin içeriğini 3 yapar
*pp-=5 → pp nin gösterdiği adresin içeriğini 5 azaltır
**pp-=2 → pp nin gösterdiği adresin gösterdiği adresin içeriğini 2 azaltır.

CEVAP:

50
80
80
43
46
32
70
70
43
70

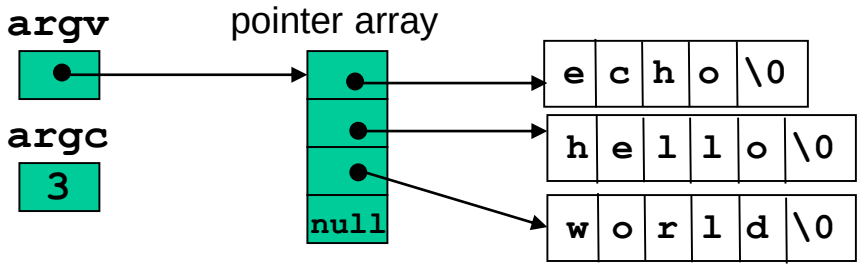
Command-Line Arguments

- `argc` and `argv`
  - In environments those support C, there is a way to pass *command-line arguments* or parameters to a program when it begin executing.
  - When main is called to begin execution, it is called with two arguments – **`argc`** and **`argv`**
    - **`argc`** : The first (conventionally called ***`argc`***) is the number of command-line arguments the program was invoked with
    - **`argv`** : The second (conventionally called ***`argv`***) is a pointer to an array of character strings that contain the arguments, one per string.

Example:

if **`echo`** is a program and executed on unix prompt, such as

```
10 <user:/home/droberts> echo hello world
```



Command-Line Arguments

Example: print out the arguments. ex: `hello world`

```
main (int argc, char *argv[])
{
  int i;
  for (i = 1; i < argc; i++)
    printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n');
}
```

```
main (int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s%c", *++argv, (argc > 1) ? ' ' : '\n');
}
```

What if

"*++argv" ← "*argv++"

Referance

- Ioannis A. Vetsikas, Lecture notes
- Dale Roberts, Lecture notes