Lecture#3

`sizeof` Operator

Pointer Expressions and Pointer Arithmetic

Relationship between Pointers and Arrays

Arrays of Pointers

CENG 102- Algorithms and Programming II,

2024-2025, Spring

Contains materials from:
P. Deitel, H. Deitel, ‘’C How to Program with an Introduction to C++’’, 8th edition, Pearson

Lecture 4.1

`sizeof` Operator

7.7 `sizeof` Operator

- C provides the special unary operator `sizeof` to determine the **size in bytes** of an array (or any other data type) .
- When applied to a single variable, the `sizeof` operator **returns the total number of bytes** to store that variable.
  - float x;
  - printf("%u", sizeof(x)); //outputs **4** (may vary in different systems)
- When applied to the **name of an array**, the `sizeof` operator **returns the total number of bytes** in the array **as type `size_t`**.
  - float arr[10];
  - pintf("%u", sizeof(arr)); //outputs **40** (4 x 10)

7.7 sizeof Operator

```c
#include <stdio.h>

int main()
{
    float x=0;
    printf("%u", sizeof(x));
}
```

```
main.c:7:14: warning: format ‘%u’ expects argument of type ‘unsigned int’, but argument 2 has type ‘long unsigned int’ [-Wformat=]
    7 |     printf("%u", sizeof(x));
      |             ~^   ~~~~~~~~~
      |              |   |
      |              |   long unsigned int
      |             unsigned int
      |             %lu
4
```

```c
#include <stdio.h>

int main()
{
    float x=0;
    printf("%lu", sizeof(x));
}
```

```
4
```

```c
#include <stdio.h>

int main()
{
    //sizeof single variable and array
    float x;
    float arr[10];
    printf("sizeof x: %u\n", sizeof(x)); //single variable
    printf("sizeof arr: %u\n", sizeof(arr)); //whole array
    printf("sizeof arr[0]: %u\n", sizeof(arr[0])); //single element of the arr

    return 0;
}
```

```
sizeof x: 4
sizeof arr: 40
sizeof arr[0]: 4

Process returned 0 (0x0)   execution time : 0.000 s
Press any key to continue.
```

7.7 `sizeof` Operator (Cont.)

- The number of elements in an array also can be determined with `sizeof`.
- For example, consider the following array definition:
  - `double real[22];`
- Variables of type **`double`** normally are stored in **8 bytes** of memory.
- Thus, array `real` contains a total of 176 bytes.
- To determine the number of elements in the array, the following expression can be used:
  - `sizeof(real) / sizeof(real[0])`

7.7 `sizeof` Operator (Cont.)

- The expression determines the number of bytes in array `real` and divides that value by the number of bytes used in memory to store the first element of array `real` (a `double` value).

7.7 `sizeof` Operator (Cont.)

- When you use `sizeof` **with a pointer**, it returns the ***size of the pointer***, not the size of the item to which it points.
- The size of a pointer on our system is 8 bytes. It does not matter if it points to char, float, int, or double. Every pointer has the same size.
- Therefore, the calculation shown above for determining the number of array elements using `sizeof` works only when using the **actual array**, not when using a pointer to the array.

```c
#include <stdio.h>

int main()
{
    int a;
    int *b;

    float x;
    float *y;

    printf("size of int: %u\n", sizeof(a));
    printf("size of int pointer: %u\n", sizeof(b));
    printf("size of float: %u\n", sizeof(x));
    printf("size of float pointer: %u\n", sizeof(y));

    return 0;
}

```

```
size of int: 4
size of int pointer: 8
size of float: 4
size of float pointer: 8
```

```
#include <stdio.h>

int main()
{
    double arr[10];
    double *arrPtr = arr;

    printf("There are %u elements in the arr.\n",
           sizeof(arr) / sizeof(arr[0]));

    printf("\nUsing pointer of the array instead of its actual name:\n");
    printf("The number of elements is incorrectly calculated as:%u\n",
           sizeof(arrPtr) / sizeof(arr[0]));

    return 0;
}
```

```
There are 10 elements in the arr.

Using pointer of the array instead of its actual name:
The number of elements is incorrectly calculated as:1
```

Using data types directly in sizeof

- We can also learn the size of a data type without even declaring a variable:
  - printf("%u", sizeof(**char**));
  - printf("%u", sizeof(**short int**));
  - printf("%u", sizeof(**float**));
  - …

Sizeof without parenthesis

- We can use size of without parenthesis if we use variable or constant in it. This will not work if we use data types directly as shown in the previous slide.

```c
#include <stdio.h>
#define PI 3.14

int main()
{
	char c;
	short int s;
	printf("%u", sizeof c);
	printf("\n");
	printf("%u", sizeof s);
	printf("\n");
	printf("%u", sizeof PI);

	return 0;
}
```

```
1
2
8
```

7.7 `sizeof` Operator (Cont.)

Determining the Sizes of the Standard Types, an Array and a Pointer

- The following code example calculates the number of bytes used to store each of the standard data types.

- *The results of this program are implementation dependent and often differ across platforms (OS) and sometimes across different compilers on the same platform.*

```c
// Fig. 7.17: fig07_17.c
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

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

   printf("     sizeof c = %u\tsizeof(char)  = %u"
          "\n     sizeof s = %u\tsizeof(short) = %u"
          "\n     sizeof i = %u\tsizeof(int) = %u"
          "\n     sizeof l = %u\tsizeof(long) = %u"
          "\n    sizeof ll = %u\tsizeof(long long) = %u"
          "\n     sizeof f = %u\tsizeof(float) = %u"
```

Fig. 7.17 | Using operator `sizeof` to determine standard data type sizes. (Part 1 of 2.)

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

```
     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 ll = 8       sizeof(long long) = 8
     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
```

```
     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 ll = 8       sizeof(long long) = 8
     sizeof f = 4       sizeof(float) = 4
     sizeof d = 8       sizeof(double) = 8
    sizeof ld = 16      sizeof(long double) = 16
 sizeof array = 80
   sizeof ptr = 8
```

Fig. 7.17 | Using operator `sizeof` to determine standard data type sizes. (Part 2 of 2.)

Portability Tip 7.1

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.*

7.7 `sizeof` Operator (Cont.)

- Operator `sizeof` can be applied to any variable name, type or value (including the value of an expression).
  - int x = 2;
  - printf("%u", sizeof(**x**));
  - printf("%u", sizeof(**6**));
  - printf("%u", sizeof(**x*x**));

- The parentheses are required when a type is supplied as `size_of`'s operand.

Lecture 4.2

Pointer Expressions and Pointer Arithmetic

7.8 Pointer Expressions and Pointer Arithmetic

- Pointers are valid operands in arithmetic expressions, assignment expressions and comparison expressions.
- However, not all the operators normally used in expressions are valid with pointer variables.
- This section describes the operators that can have pointers as operands, and how these operators are used.

7.8 Pointer Expressions and Pointer Arithmetic

- A limited set of arithmetic operations may be performed on pointers.
- A pointer may be
  - *incremented* (++) or *decremented* (--),
  - an integer may be *added* to a pointer (+ or +=) or *subtracted* from a pointer (- or -=),
  - one pointer may be subtracted from another pointer.
    - this last operation is meaningful only when *both* pointers point to elements of the *same* array.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Assume that array `int v[5]` has been defined and its first element is at location `3000` in memory.
- Assume pointer `vPtr` has been initialized to point to `v[0]`—i.e., the value of `vPtr` is `3000`.
- Figure 7.18 illustrates this situation for a machine with 4-byte integers.
- Variable `vPtr` can be initialized to point to array `v` with either of the statements

Portability Tip 7.2

Because the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is machine and compiler dependent.

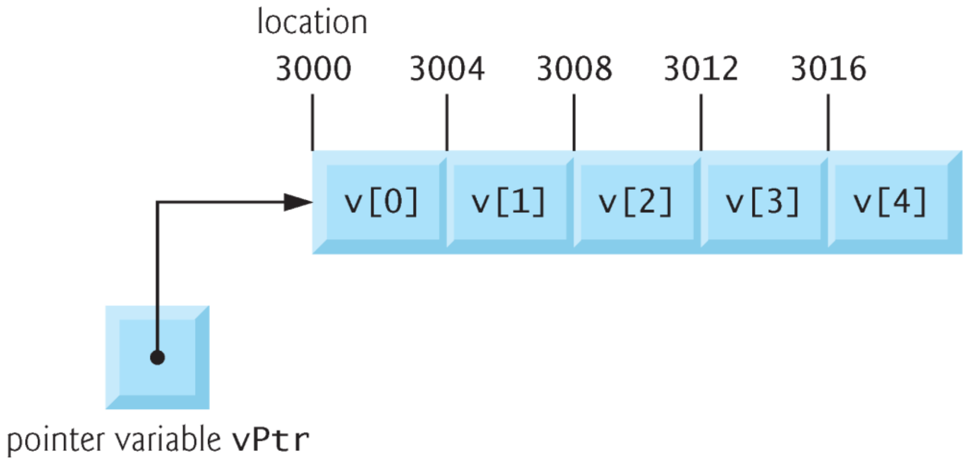

Fig. 7.18 | Array `v` and a pointer variable `vPtr` that points to `v`.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- In conventional arithmetic, `3000 + 2` yields the value `3002`.
- This is normally not the case with pointer arithmetic.
- When an integer is added to or subtracted from a pointer, the pointer is *not* incremented or decremented simply by that integer, but by that integer times the size of the object to which the pointer refers.
- The number of bytes depends on the object's data type.
- For example, the statement
    - `vPtr += 2;`

  would produce `3008` (`3000` + 2 * 4), assuming an integer is stored in 4 bytes of memory.

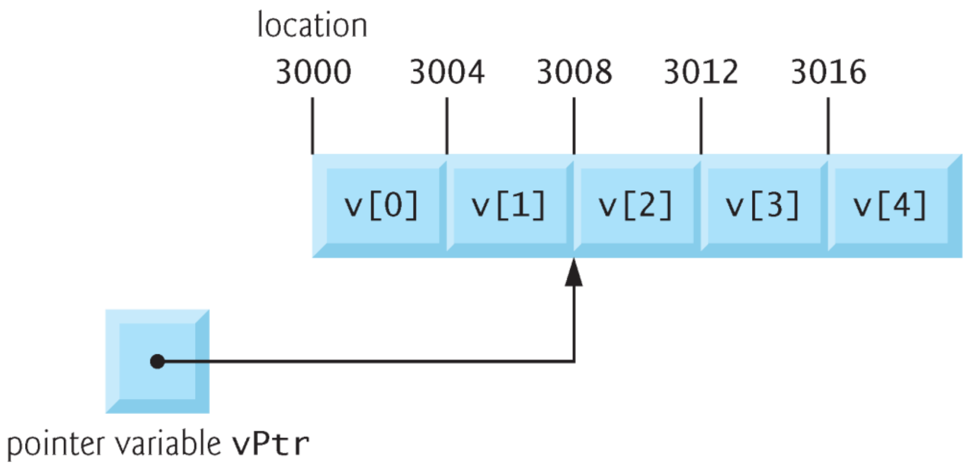

Fig. 7.19 | The pointer `vPtr` after pointer arithmetic.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- In the array `v`, `vPtr` would now point to `v[2]` (Fig. 7.19).
- If an integer was stored in 2 bytes of memory, then the preceding calculation would result in memory location `3004` (`3000 + 2 * 2`).
- If the array were of a different data type, the preceding statement would increment the pointer by twice the number of bytes that it takes to store an object of that data type.
  - i.e. 2 * 8 for double
- When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is 1 byte long.

Common Programming Error 7.4

Using pointer arithmetic on a pointer that does not refer to an element in an array.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- If `vPtr` had been incremented to `3016`, which points to `v[4]`, the statement
  - `vPtr -= 4;`

  would set `vPtr` back to `3000`—the beginning of the array.

- If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Either of the sEither of the statements
  - ```
    ++vPtr;
    vPtr++;
    ```

  increments the pointer to point to the *next* location in the array.
- tatements
  - ```
    --vPtr;
    vPtr--;
    ```

  decrements the pointer to point to the *previous* element of the array.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Pointer variables may be subtracted from one another.
- For example, if `vPtr` contains the location `3000`, and `v2Ptr` contains the address `3008`, the statement
    - `x = v2Ptr - vPtr;`

  would assign to `x` the *number of array elements* from `vPtr` to `v2Ptr`, in this case 2 (not 8).
- Pointer arithmetic is undefined unless performed on an array.
- We cannot assume that two variables of the same type are stored contiguously in memory if they are not adjacent elements of an array.

Common Programming Error 7.5

Running off either end of an array when using pointer arithmetic.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- A pointer can be assigned to another pointer if both have the same type.
- The exception to this rule is the pointer to void (i.e., `void *`), which is a generic pointer that can represent *any* pointer type.
- All pointer types can be assigned a pointer to `void`, and a pointer to `void` can be assigned a pointer of any type.
- A pointer to `void` *cannot* be dereferenced.

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Consider this: The compiler knows that a pointer to `int` refers to 4 bytes of memory on a machine with 4-byte integers, but a pointer to `void` simply contains a memory location for an *unknown* data type—the precise number of bytes to which the pointer refers is not known by the compiler.
- The compiler *must* know the data type to determine the number of bytes to be dereferenced for a particular pointer.

Common Programming Error 7.7

Assigning a pointer of one type to a pointer of another type if neither is of type `void *` *is a syntax error.*

Common Programming Error 7.8

Dereferencing a `void *` *pointer is a syntax error.*

7.8 Pointer Expressions and Pointer Arithmetic (Cont.)

- Pointers can be compared using equality and relational operators, but such comparisons are meaningless unless the pointers point to elements of the *same* array.
- Pointer comparisons compare the addresses stored in the pointers.
- A comparison of two pointers pointing to elements in the same array could show, for example, that one pointer points to a higher-indexed element of the array than the other pointer does.
- A common use of pointer comparison is determining whether a pointer is `NULL`.

Common Programming Error 7.9

Comparing two pointers that do not refer to elements in the same *array.*

Lecture 4.3

Relationship between Pointers and Arrays

7.9 Relationship between Pointers and Arrays

- Arrays and pointers often may be used interchangeably.
- An *array name* can be thought of as a constant pointer.
- Assume that integer array `b[5]` and integer pointer variable `bPtr` have been defined.
- Because the array name (without an index) is a pointer to the first element of the array, we can set `bPtr` equal to the address of the first element in array b with the statement
  - `bPtr = b;`

7.9 Relationship between Pointers and Arrays (Cont.)

- This statement is equivalent to taking the address of the array's first element as follows:
  - `bPtr = &b[0];`
- Array element `b[3]` can alternatively be referenced with the pointer expression
  - `*(bPtr + 3)`
- The 3 in the expression is the offset to the pointer.
- When the pointer points to the array's first element, the offset indicates which array element should be referenced, and the offset value is identical to the array index.
- This notation is referred to as pointer/offset notation.

7.9 Relationship between Pointers and Arrays (Cont.)

- The parentheses are necessary because the precedence of * is higher than the precedence of +.
- Without the parentheses, the above expression would add 3 to the value of the expression `*bPtr` (i.e., 3 would be added to `b[0]`, assuming `bPtr` points to the beginning of the array).
- Just as the array element can be referenced with a pointer expression, the address
  - `&b[3]`

  can be written with the pointer expression
  - `bPtr + 3`

7.9 Relationship between Pointers and Arrays (Cont.)

- The array itself can be treated as a pointer and used in pointer arithmetic.
- For example, the expression
    - `*(b + 3)`

  also refers to the array element `b[3]`.
- In this case, pointer/offset notation was used with the name of the array as a pointer.
- The preceding statement does not modify the array name in any way; `b` still points to the first element in the array.

7.9 Relationship between Pointers and Arrays (Cont.)

- Pointers can be indexed like arrays.
- If `bPtr` has the value `b`, the expression
  - `bPtr[1]`

  refers to the array element `b[1]`.
- This is referred to as pointer/index notation.
- Remember that an array name is essentially a constant pointer; it always points to the beginning of the array.
- Thus, the expression
  - `b += 3`

  is *invalid* because it attempts to modify the value of the array name with pointer arithmetic.

Common Programming Error 7.10

Attempting to modify the value of an array name with pointer arithmetic is a compilation error.

7.9 Relationship between Pointers and Arrays (Cont.)

- Figure 7.20 uses the four methods we've discussed for referring to array elements
  - array indexing
  - pointer/offset with the array name as a pointer
  - pointer indexing
  - pointer/offset with a pointer

  to print the four elements of the integer array b.

```c
// Fig. 7.20: fig07_20.cpp
// Using indexing and pointer notations with arrays.
#include <stdio.h>
#define ARRAY_SIZE 4

int main(void)
{
   int b[] = {10, 20, 30, 40}; // create and initialize array b
   int *bPtr = b; // create bPtr and point it to array b

   // output array b using array index notation
   puts("Array b printed with:\nArray index notation");

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

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

```

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 1 of 3.)

```c
        // loop through array b
        for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
           printf("*(b + %u) = %d\n", offset, *(b + offset));
        }

        // output array b using bPtr and array index notation
        puts("\nPointer index notation");

        // loop through array b
        for (size_t i = 0; i < ARRAY_SIZE; ++i) {
           printf("bPtr[%u] = %d\n", i, bPtr[i]);
        }

        // output array b using bPtr and pointer/offset notation
        puts("\nPointer/offset notation");

        // loop through array b
        for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
           printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
        }
    }
```

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 2 of 3.)

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

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

Pointer index 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
```

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 3 of 3.)

7.9 Relationship between Pointers and Arrays (Cont.)

String Copying with Arrays and Pointers

- To further illustrate the interchangeability of arrays and pointers, let's look at the two string-copying functions—`copy1` and `copy2`—in the program of Fig. 7.21.
- Both functions copy a string into a char array.
- They accomplish the same task; but they're implemented differently.

```c
// Fig. 7.21: fig07_21.c
// Copying a string using array notation and pointer notation.
#include <stdio.h>
#define SIZE 10

void copy1(char * const s1, const char * const s2); // prototype
void copy2(char *s1, const char *s2); // prototype

int main(void)
{
   char string1[SIZE]; // create array string1
   char *string2 = "Hello"; // create a pointer to a string

   copy1(string1, string2);
   printf("string1 = %s\n", string1);

   char string3[SIZE]; // create array string3
   char string4[] = "Good Bye"; // create an array containing a string

   copy2(string3, string4);
   printf("string3 = %s\n", string3);
}

```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 1 of 2.)

```
// copy s2 to s1 using array notation
void copy1(char * const s1, const char * const s2)
{
   // loop through strings
   for (size_t i = 0; (s1[i] = s2[i]) != '\0'; ++i) {
      ; // do nothing in body
   }
}

// copy s2 to s1 using pointer notation
void copy2(char *s1, const char *s2)
{
   // loop through strings
   for (; (*s1 = *s2) != '\0'; ++s1, ++s2) {
      ; // do nothing in body
   }
}
```

```
string1 = Hello
string3 = Good Bye
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 2 of 2.)

7.9 Relationship between Pointers and Arrays (Cont.)

- Function `copy1` uses *array index notation* to copy the string in `s2` to the character array `s1`.
- The function defines counter variable `i` as the array index.
- The `for` statement header performs the entire copy operation—its body is the empty statement.
- The header specifies that `i` is initialized to zero and incremented by one on each iteration of the loop.
- The expression `s1[i] = s2[i]` copies one character from `s2` to `s1`.
- When the null character is encountered in `s2`, it's assigned to `s1`, and the value of the assignment becomes the value assigned to the left operand (`s1`).

7.9 Relationship between Pointers and Arrays (Cont.)

- The loop terminates when the null character is assigned from `s1` to `s2` (false).
- Function `copy2` uses *pointers and pointer arithmetic* to copy the string in `s2` to the character array `s1`.
- Again, the `for` statement header performs the entire copy operation.
- The header does not include any variable initialization.
- As in function `copy1`, the expression (`*s1 = *s2`) performs the copy operation.
- Pointer `s2` is dereferenced, and the resulting character is assigned to the dereferenced pointer `*s1`.

7.9 Relationship between Pointers and Arrays (Cont.)

- After the assignment in the condition, the pointers are incremented to point to the next element of array `s1` and the next character of string `s2`, respectively.
- When the null character is encountered in `s2`, it's assigned to the dereferenced pointer `s1` and the loop terminates.
- *The first argument to both `copy1` and `copy2` must be an array large enough to hold the string in the second argument.*
- Otherwise, an error may occur when an attempt is made to write into a memory location that's not part of the array.
- Also, the second parameter of each function is declared as `const char *` (a constant string).

7.9 Relationship between Pointers and Arrays (Cont.)

- In both functions, the second argument is copied into the first argument—characters are read from it one at a time, but the characters are *never modified*.
- Therefore, the second parameter is declared to point to a constant value so that the *principle of least privilege* is enforced—neither function requires the capability of modifying the second argument, so neither function is provided with that capability.

Lecture 4.4

Arrays of Pointers

7.10 Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings, referred to simply as a string array.
- Each entry in the array is a string, but in C a string is essentially a pointer to its first character.
- So, each entry in an array of strings is actually a pointer to the first character of a string.
- Consider the definition of string array `suit`, which might be useful in representing a deck of cards.
    - `const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};`

7.10 Arrays of Pointers (Cont.)

- The `suit[4]` portion of the definition indicates an array of 4 elements.
- The `char *` portion of the declaration indicates that each element of array `suit` is of type "pointer to `char`."
- Qualifier `const` indicates that the strings pointed to by each element pointer will not be modified.
- The four values to be placed in the array are `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`.
- Each is stored in memory as a *null-terminated character string* that's one character longer than the number of characters between quotes.

7.10 Arrays of Pointers (Cont.)

- The four strings are 7, 9, 6 and 7 characters long, respectively.
- Although it appears as if these strings are being placed in the `suit` array, only pointers are actually stored in the array (Fig. 7.22).
- Each pointer points to the first character of its corresponding string.
- Thus, even though the `suit` array is *fixed* in size, it provides access to character strings of *any length*.

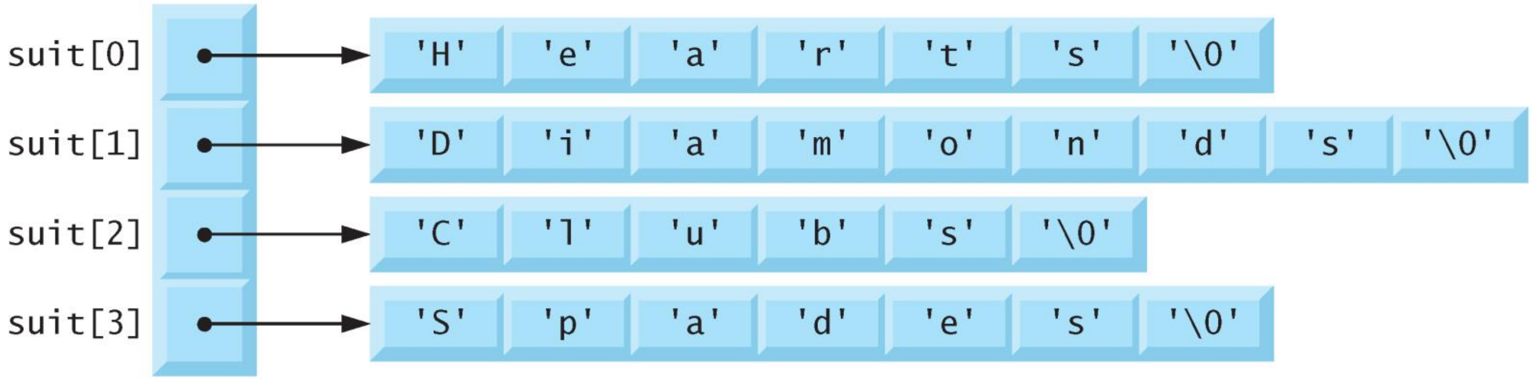

Fig. 7.22 | Graphical representation of the `suit` array.

7.10 Arrays of Pointers (Cont.)

- The suits could have been placed in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name.
- Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string.
- Therefore, considerable memory could be wasted when storing a large number of strings of which most were shorter than the longest string.