VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY

HOCHIMINH CITY INTERNATIONAL UNIVERSITY • HCM - IU •

SCHOOL OF ELECTRICAL ENGINEERING * IU - EE *

Programming for Engineers

Lecture 7: C Pointers

Course ID: EE057IU

Lecture Outline

- Introduction to C Pointer
- Initializing Pointer
- Passing Arguments in C
- Const Pointer
- sizeof
- Pointer and Array
- Function Pointer

Introduction to Pointers

► What is a Pointer?

- A **pointer** is a variable that stores the **memory address** of another variable.
- It “points” to the location of a value in memory.

► Key Characteristics

1. The pointers hold addresses, not actual values.
2. Declared using the `*` symbol.
   - Example: `int *ptr;` (Pointer to an integer)
3. Can be dereferenced to access or modify the value at the memory address.

Why Do We Need Pointers?

1. **Efficient Memory Access**
   - Direct access to memory for low-level operations.
2. **Dynamic Memory Management**
   - Allows allocating and freeing memory at runtime.
3. **Passing by Reference**
   - Modify variables directly via function arguments.
4. **Advanced Data Structures**
   - Essential for creating linked lists, trees, and graphs.
5. **Hardware Interaction**
   - Work with memory-mapped devices or system-level operations.
6. **Optimized Array and String Handling**
   - Enables efficient manipulation and iteration.

Memory

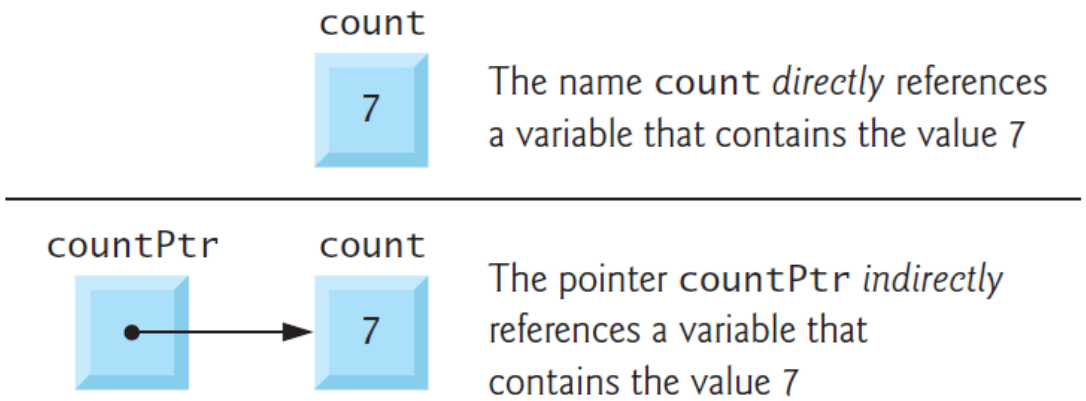
Passing & Data Structures

Arrays

Declaring Pointers

- Pointers must be defined before they can used
- The definition
  - `int *countPtr, count;`

  Specifis that variable countPtr is of type int * (i.e., a pointer to an integer)
- The variable count is defined to be an *int*, *not* a pointer to an *int*



Pointer Variable Naming

- **Consistent Naming for Clarity and Safety**
  - End with “Ptr”

```
int *countPtr;
char *namePtr;
```

  - Prefix with “p” or “_p”

```
int *pCount, *pName
int *p_count, *p_name
```

Initializing Pointers

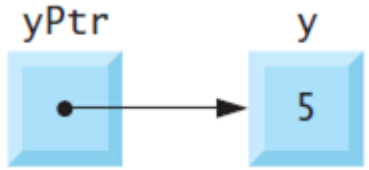
- Pointers should be initialized when:
  - They are defined
  - They can be assigned a value
- A pointer may be initialized to NULL, 0, or an address
- A pointer with the value NULL points to nothing
- Intializing a pointer to 0 is equivalent to initializing a pointer to NULL
- When 0 is assigned, it's first converted to a pointer of the appropriate type
- The value 0 is the only integer value that can be assigned directly to a pointer variable.

Pointer Operators

- **The Address (&) Operator**: return the address of its operand
- Example definition
  - **int y = 5;**
  - **int *yPtr;**

  the statement
  - yPtr = &y

  assigns the *address* of the variable y to pointer variable yPtr



Pointer Operators

- **Indirection (*) Operator**: return the *value* of the object to which its operant (i.e., pointer) point.
- Example:
  - printf(“%d”, *yPtr);

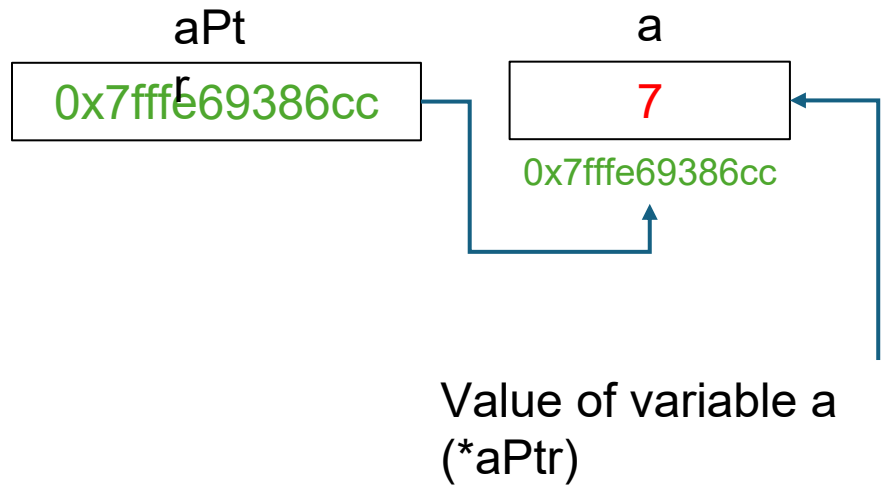
  prints the value of variable that yPtr is pointing to in this case it is y, whose value is 5.

- Using * in this manner is called dereferencing a pointer.

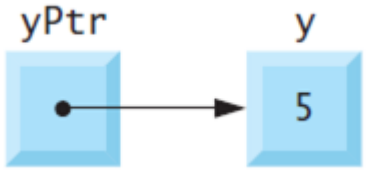
Demonstration the & and * Operators

```c
#include <stdio.h>

int main(void)
{
    int a = 7;
    int *aPtr = &a; // set aPtr to the 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;
}
```

```
Address of a is 0x7fffe69386cc
Value of aPtr is 0x7fffe69386cc

Value of a is 7
Value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0x7fffe69386cc
*&aPtr = 0x7fffe69386cc
```



Direct Initialization vs Two-step Initialization



Direct Initialization

```
int a = 5;
int *aPtr = &a;
```

Two-step initialization

```
int a = 5;
int *aPtr;
aPtr = &a;
```

In-class example

```c
#include <stdio.h>

int main() {
    double d = 123.45;      // Declare and initialize the variable
    double *dPtr;           // Declare a pointer to a double

    dPtr = &d;              // Assign the address of d to the pointer dPtr

    // Print the initial value and address of d
    printf("Initial value of d: %.2f\n", d);
    printf("The address of d is: %p\n", &d);
    printf("The value pointed to by dPtr is: %.2f\n", *dPtr);
    printf("The address stored in dPtr is: %p\n", dPtr);

    // Change the value of d using the pointer
    *dPtr = 456.78;         // Modify the value of d through the pointer

    // Print the modified value and address of d
    printf("\nAfter modifying through dPtr:\n");
    printf("Modified value of d: %.2f\n", d);
    printf("The value pointed to by dPtr is: %.2f\n", *dPtr);

    return 0;
}
```

```
Initial value of d: 123.45
The address of d is: 0x7ffcceb48340
The value pointed to by dPtr is: 123.45
The address stored in dPtr is: 0x7ffcceb48340

After modifying through dPtr:
Modified value of d: 456.78
The value pointed to by dPtr is: 456.78
```

Passing Arguments in C

- **There are two ways to pass arguments to a function:**
  - **Pass-by-Value:** Copies the variable's value to the function. Changes inside the function do not affect the caller.
  - **Pass-by-Reference:** Uses pointers to pass the variable's address, allowing the function to modify the caller's variable.

- **Key Benefits of Pass-by-Reference**:
  1. Modify variables in the caller.
  2. Avoid copying large data objects, improving efficiency.
  3. Simulate returning multiple values by modifying the caller's variables.

Passing Arguments – by Value

```c
// fig07_02.c
// Cube a variable using pass-by-value.
#include <stdio.h>

int cubeByValue(int n); // prototype

int main(void) {
    int number = 5; // initialize number

    printf("The original value of number is %d\n", number);
    number = cubeByValue(number); // pass number by value to cubeByValue
    printf("\nThe new value of number is %d\n", number);

    return 0;
}

// calculate and return cube of integer argument
int cubeByValue(int n) {
    return n * n * n; // cube local variable n and return result
}
```

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

Passing Arguments – by Reference

```c
// fig07_03.c
// Cube a variable using pass-by-reference with a pointer argument.
#include <stdio.h>

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

int main(void) {
    int number = 5; // initialize number

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

    return 0;
}

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

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

Passing Arguments – by Value (Visualization)

Step 1: Before `main` calls `cubeByValue`:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

number: 5

Step 2: After `cubeByValue` receives the call:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

number: 5

```
int cubeByValue( int n ) {

   return n * n * n;

}
```

n: 5

Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

number: 5

```
int cubeByValue(int n) {

   return n * n * n;

}
```

125

n: 5

Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:

```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

125

number: 5

Step 5: After `main` completes the assignment to `number`:

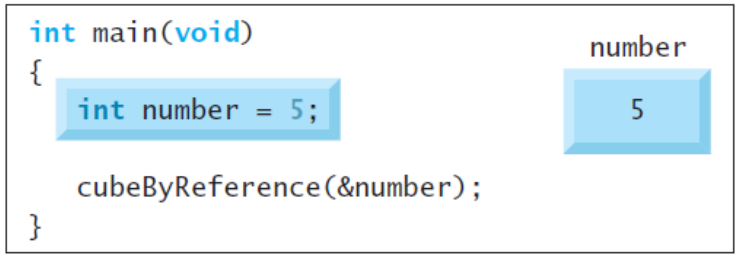
```
int main(void) {

   int number = 5;

   number = cubeByValue(number);
}
```

125 125 125

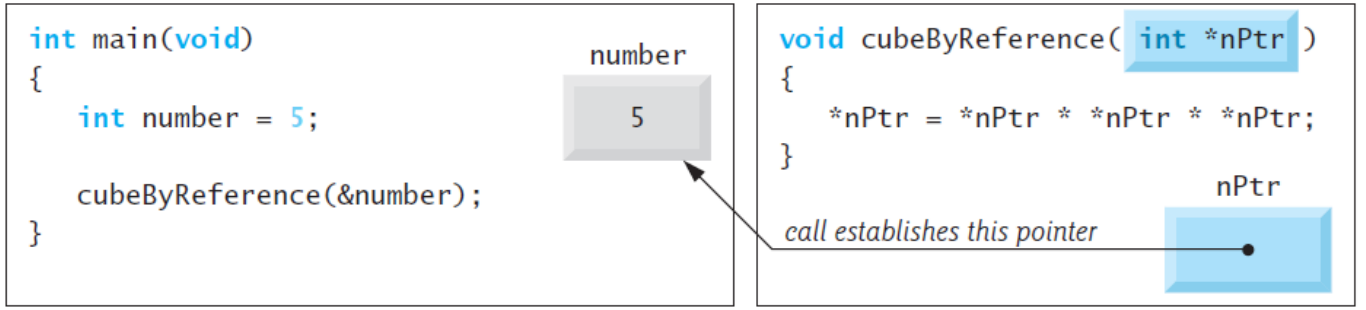
number: 125

Passing Arguments – by Reference (Visualization)

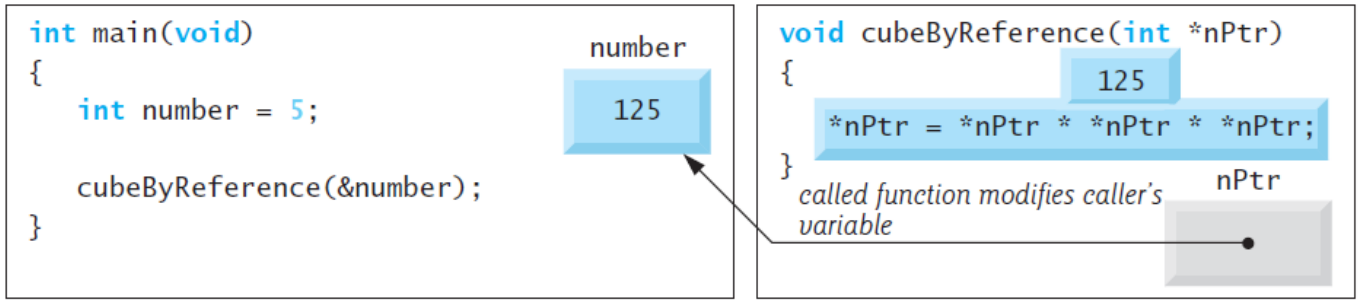
Step 1: Before `main` calls `cubeByReference`:



Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



Step 3: After `*nPtr` is cubed and before program control returns to `main`:



Using the *const* Qualifier with Pointers

- **Purpose of const:**
  - Prevents modification of variables, ensuring adherence to the principle of least privilege.
  - Reduces debugging time and avoids unintentional side effects.
  - Enhances code robustness and maintainability.
- **Legacy Issues with const:**
  - Older C versions lacked const, leading to less secure and maintainable code.
  - Opportunities exist to improve legacy code by integrating const.
- **Four Ways to Pass a Pointer to Data:**
  - Non-constant pointer to non-constant data
  - Constant pointer to non-constant data
  - Non-constant pointer to constant data
  - Constant pointer to constant data

Non-constant pointer to non-constant data

```c
// fig07_06.c
// Converting a string to uppercase using a
// non-constant pointer to non-constant data.
#include <ctype.h>
#include <stdio.h>

void convertToUppercase(char *sPtr); // prototype

int main(void) {
   char string[] = "cHaRaCters and $32.98"; // initialize char array

   printf("The string before conversion is: %s\n", string);
   convertToUppercase(string);
   printf("The string after conversion is: %s\n", string);
}

// convert string to uppercase letters
void convertToUppercase(char *sPtr) {
   while (*sPtr != ) { // current character is not
      *sPtr = toupper(*sPtr); // convert to uppercase
      ++sPtr; // make sPtr point to the next character
   }
}
```

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

Non-constant pointer to constant data

```c
// fig07_07.c
// Printing a string one character at a time using
// a non-constant pointer to constant data.

#include <stdio.h>

void printCharacters(const char *sPtr);

int main(void) {
   // initialize char array
   char string[] = "print characters of a string";

   puts("The string is:");
   printCharacters(string);
   puts("");
}

// sPtr cannot be used to modify the character to which it points,
// i.e., sPtr is a "read-only" pointer
void printCharacters(const char *sPtr) {
   // loop through entire string
   for (; *sPtr != ; ++sPtr) { // no initialization
      printf("%c", *sPtr);
   }
}
```

```
The string is:
print characters of a string
```

Constant pointer to non-constant data

```
1  // fig07_09.c
2  // Attempting to modify a constant pointer to non-constant data.
3  #include <stdio.h>
4
5  int main(void) {
6     int x = 0; // define x
7     int y = 0; // define y
8
9     // ptr is a constant pointer to an integer that can be modified
10    // through ptr, but ptr always points to the same memory location
11    int * const ptr = &x;
12
13    *ptr = 7; // allowed: *ptr is not const
14    ptr = &y; // error: ptr is const; cannot assign new address
15 }
```

Microsoft Visual C++ Error Message

```
fig07_09.c(14,4): error C2166: l-value specifies const object
```

Fig. 7.9 | Attempting to modify a constant pointer to non-constant data.

Constant pointer to constant data

```
1  // fig07_10.c
2  // Attempting to modify a constant pointer to constant data.
3  #include <stdio.h>
4
5  int main(void) {
6     int x = 5;
7     int y = 0;
8
9     // ptr is a constant pointer to a constant integer. ptr always
10    // points to the same location; the integer at that location
11    // cannot be modified
12    const int *const ptr = &x; // initialization is OK
13
14    printf("%d\n", *ptr);
15    *ptr = 7; // error: *ptr is const; cannot assign new value
16    ptr = &y; // error: ptr is const; cannot assign new address
17 }
```

Microsoft Visual C++ Error Message

```
fig07_10.c(15,5): error C2166: l-value specifies const object
fig07_10.c(16,4): error C2166: l-value specifies const object
```

Fig. 7.10 | Attempting to modify a constant pointer to constant data. (Part 2 of 2.)

Bubble Sort using Pass-By-Reference

```c
// fig07_11.c
// Putting values into an array, sorting the values into
// ascending order and printing the resulting array.
#include <stdio.h>
#define SIZE 10

void bubbleSort(int * const array, size_t size); // prototype

int main(void) {
   // initialize array a
   int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   puts("Data items in original order");

   // loop through array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   bubbleSort(a, SIZE); // sort the array

   puts("\nData items in ascending order");

   // loop through array a
   for (size_t i = 0; i < SIZE; ++i) {
      printf("%4d", a[i]);
   }

   puts("");
}
```

```c
// sort an array of integers using bubble sort algorithm
void bubbleSort(int * const array, size_t size) {
   void swap(int *element1Ptr, int *element2Ptr); // prototype

   // loop to control passes
   for (int pass = 0; pass < size - 1; ++pass) {
      // loop to control comparisons during each pass
      for (size_t j = 0; j < size - 1; ++j) {
         // swap adjacent elements if they're out of order
         if (array[j] > array[j + 1]) {
            swap(&array[j], &array[j + 1]);
         }
      }
   }
}

// swap values at memory locations to which element1Ptr and
// element2Ptr point
void swap(int *element1Ptr, int *element2Ptr) {
   int hold = *element1Ptr;
   *element1Ptr = *element2Ptr;
   *element2Ptr = hold;
}
```

```
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
```

$sizeof$ Operator

The $sizeof$ Operator in C

- The $sizeof$ operator determines an object or type's size in bytes.
- It is applied at compilation time unless the operand is a variable-length array (VLA).
- When applied to an array's name, sizeof returns the total number of bytes:
  - Example: For a float array with 20 elements (each float is 4 bytes), the size is 80 bytes.
- $sizeof$ is a compile-time operator, meaning it does not incur execution-time overhead (except for VLAs).
- Use Case: $sizeof$(array) can be used to determine memory requirements.

sizeof Operator

```c
// fig07_12.c
// Applying sizeof to an array name returns
// the number of bytes in the array.
#include <stdio.h>
#define SIZE 20

size_t getSize(const float *ptr); // prototype

int main(void){
   float array[SIZE]; // create array

   printf("Number of bytes in the array is %zu\n", sizeof(array));
   printf("Number of bytes returned by getSize is %zu\n", getSize(array));
}

// return size of ptr
size_t getSize(const float *ptr) {
   return sizeof(ptr);
}
```

```
Number of bytes in the array is 80
Number of bytes returned by getSize is 8
```

Return size of the memory address, not the size of data type it points to

Fig. 7.12 | Applying `sizeof` to an array name returns the number of bytes in the array.

sizeof Operator

```c
// fig07_13.c
// Using operator sizeof to determine standard type sizes.
#include <stdio.h>

int main(void) {
   char c = ;
   short s = 0;
   int i = 0;
   long l = 0;
   long long ll = 0;
   float f = 0.0F;
   double d = 0.0;
   long double ld = 0.0;
   int array[20] = {0}; // create array of 20 int elements
   int *ptr = array; // create pointer to array

   printf("    sizeof c = %2zu\t       sizeof(char) = %2zu\n",
      sizeof c, sizeof(char));
   printf("    sizeof s = %2zu\t      sizeof(short) = %2zu\n",
      sizeof s, sizeof(short));
   printf("    sizeof i = %2zu\t        sizeof(int) = %2zu\n",
      sizeof i, sizeof(int));
   printf("    sizeof l = %2zu\t       sizeof(long) = %2zu\n",
      sizeof l, sizeof(long));
   printf("   sizeof ll = %2zu\t  sizeof(long long) = %2zu\n",
      sizeof ll, sizeof(long long));
   printf("    sizeof f = %2zu\t      sizeof(float) = %2zu\n",
      sizeof f, sizeof(float));
   printf("    sizeof d = %2zu\t     sizeof(double) = %2zu\n",
      sizeof d, sizeof(double));
   printf("   sizeof ld = %2zu\tsizeof(long double) = %2zu\n",
      sizeof ld, sizeof(long double));
   printf("sizeof array = %2zu\n  sizeof ptr = %2zu\n",
      sizeof array, sizeof ptr);
}
```

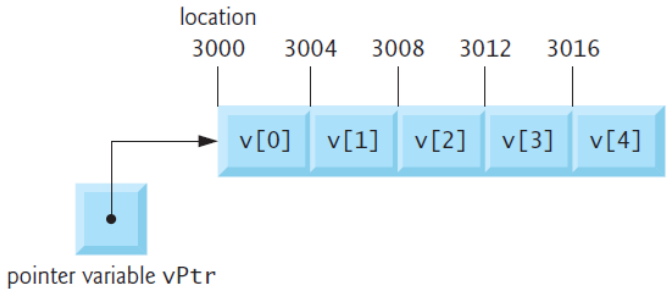
```
    sizeof c =  1            sizeof(char) =  1
    sizeof s =  2           sizeof(short) =  2
    sizeof i =  4             sizeof(int) =  4
    sizeof l =  8            sizeof(long) =  8
   sizeof ll =  8       sizeof(long long) =  8
    sizeof f =  4           sizeof(float) =  4
    sizeof d =  8          sizeof(double) =  8
   sizeof ld = 16     sizeof(long double) = 16
sizeof array = 80
  sizeof ptr =  8
```

Pointer Expressions and Pointer Arithmetic

➢ Pointer Arithmetic Operators

- Incrementing (++) or decrementing (--)
- Adding an integer to a pointer (+ or +=)
- Subtracting an integer from a pointer (- or -=)
- Subtracting one pointer from another (only meaningful when both pointers point to the same array)

➢ Aiming a Pointer at an Array

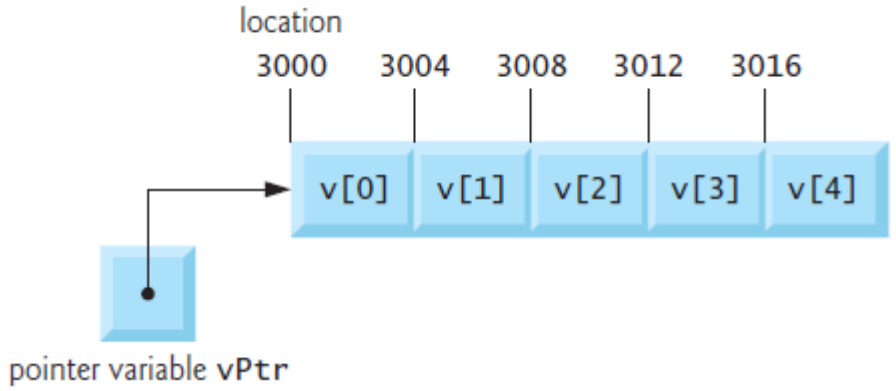


The variable `vPtr` can be initialized to point to array `v` with either of the statements

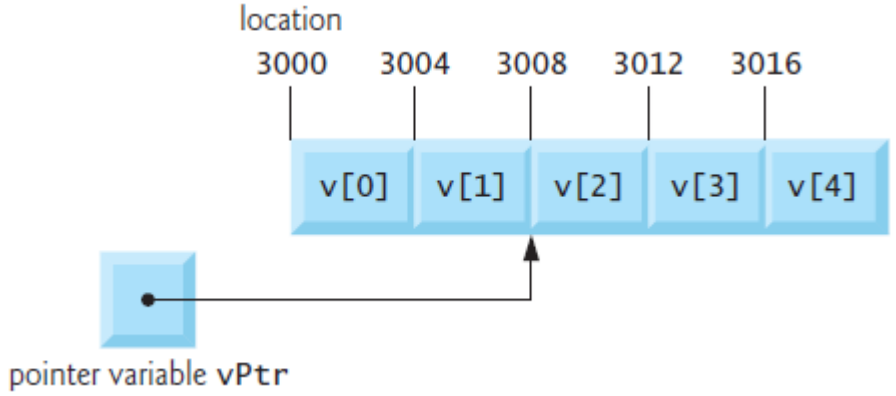
```
vPtr = v;
vPtr = &v[0];
```

Pointer Expressions and Pointer Arithmetic

```
vPtr = v;
vPtr = &v[0];
```




```
vPtr += 2;
```



Pointers and Array

- **Arrays and Pointers relationship by initialization**
  - Assuming the following definitions:

    int b[5];

    int *bPtr;

  - Set bPtr address to the address of the array's first element

    bPtr = b;

  - This equivalent to

    bPtr = &b[0]

Pointers and Array

- **Pointer/Offset Notation**
  - Array element b[3] can be referenced by the pointer expression

    $*(bPtr + 3)$
  - Address of b[3] can be referenced as

    $\&b[3]$

    $(bPtr + 3)$
- **Pointer/Subscript Notation**
  - Can be subscripted like arrays

    $bPtr[1]$

    refers to array element b[1]

Pointers and Array

```
1  // fig07_14.cpp
2  // Using subscripting and pointer notations with arrays.
3  #include <stdio.h>
4  #define ARRAY_SIZE 4
5
6  int main(void) {
7     int b[] = {10, 20, 30, 40}; // create and initialize array b
8     int *bPtr = b; // create bPtr and point it to array b
9
10    // output array b using array subscript notation
11    puts("Array b printed with:\nArray subscript notation");
12
13    // loop through array b
14    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
15       printf("b[%zu] = %d\n", i, b[i]);
16    }
17
18    // output array b using array name and pointer/offset notation
19    puts("\nPointer/offset notation where the pointer is the array name");
20
21    // loop through array b
22    for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
23       printf("*(b + %zu) = %d\n", offset, *(b + offset));
24    }
25
26    // output array b using bPtr and array subscript notation
27    puts("\nPointer subscript notation");
28
29    // loop through array b
30    for (size_t i = 0; i < ARRAY_SIZE; ++i) {
31       printf("bPtr[%zu] = %d\n", i, bPtr[i]);
32    }
33
34    // output array b using bPtr and pointer/offset notation
35    puts("\nPointer/offset notation");
36
37    // loop through array b
38    for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
39       printf("*(bPtr + %zu) = %d\n", offset, *(bPtr + offset));
40    }
41 }
```

```
Array b printed with:
Array subscript 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 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
```

Pointers and Array

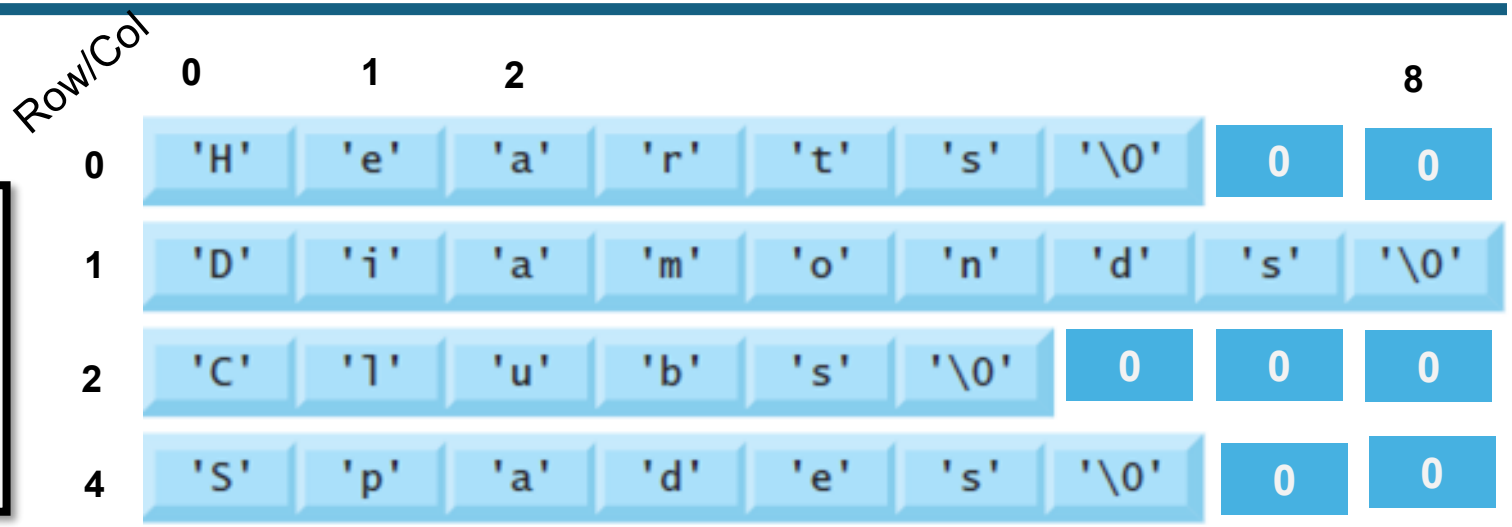
```c
// fig07_15.c
// Copying a string using array notation and pointer notation.
#include <stdio.h>
#define SIZE 10

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);
}

// 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]) != ; ++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) != ; ++s1, ++s2) {
      ; // do nothing in body
   }
}
```

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

Arrays of Pointers

Inefficient 2D array of strings

```
char suits_2d[4][9] = {
    "Hearts",    // 7 characters + \0 = 8 used
    "Diamonds",  // 8 characters + \0 = 9 used
    "Clubs",     // 6 characters + \0 = 7 used
    "Spades"     // 7 characters + \0 = 8 used
};
```



We need more highly efficient for creating string arrays because:

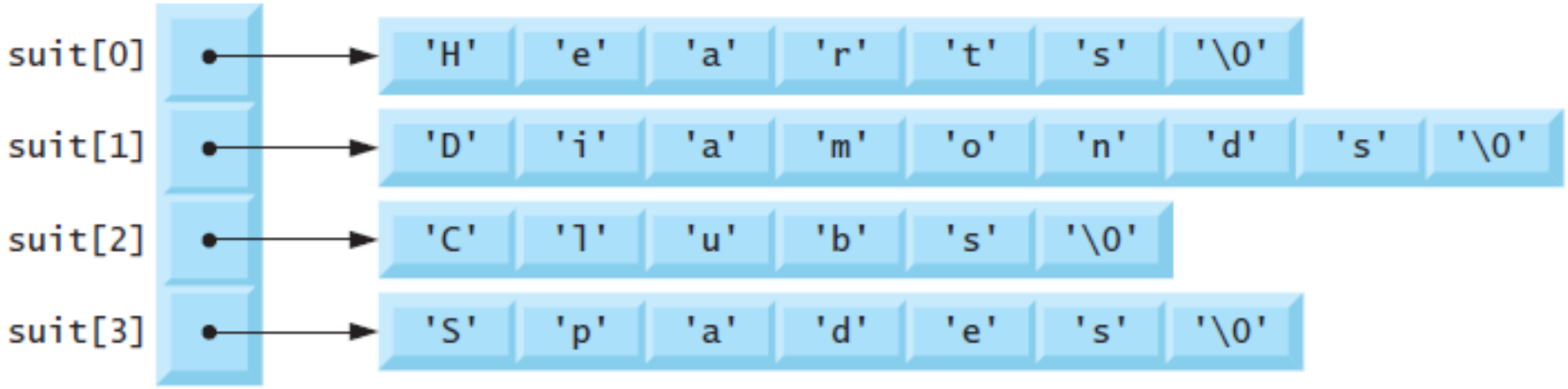
- **Flexibility (Non-Uniform Strings):** Each string can be a different length (e.g., `"Clubs"` is 5 characters, `"Diamonds"` is 8 characters). This is much more memory-efficient than a 2D character array (`char suit[4][9]`), which would force every string to be the maximum length (10 characters, wasting space).
- **Ease of Assignment:** Assigning a new string to an element is as simple as changing the address the pointer holds (e.g., `suit[0] = "Joker";`).

Arrays of Pointers

An **array of pointers** is:

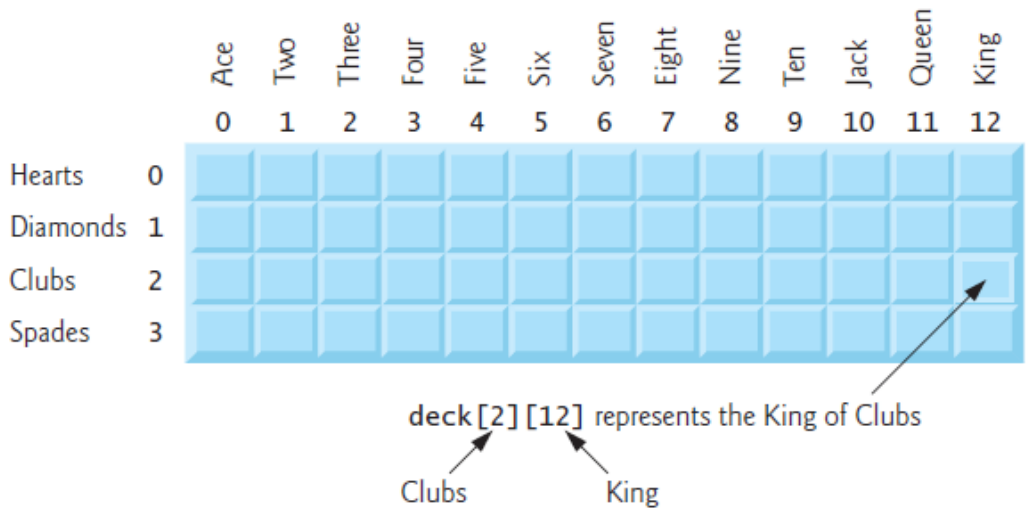
- a 1D array
- each element in the array is a **pointer** (a memory address), **not a** simple value like an integer or a single character

- Array may contain pointers.
  - Common use: array of strings, (i.e., string array)
  - $const\ char\ *\ suit[4]$ = {"Hearts", "Diamonds", "Clubs", "Spades"};



Arrays of Pointers

- Array may contain pointers.
  - Use a 4-by-13 two-dimensional array deck to represent the deck of playing cards




```c
#include <stdio.h>
int main() {
    // Card suit names
    const char *suits[] = {"Hearts", "Diamonds", "Clubs", "Spades"};
    // Card rank names
    const char *ranks[] = {
        "Ace", "Two", "Three", "Four", "Five", "Six",
        "Seven", "Eight", "Nine", "Ten", "Jack", "Queen", "King"
    };
    // 2D pointer array for the deck (4 suits, 13 ranks)
    const char *deck[4][13];
    // Fill the deck array with rank strings
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 13; j++) {
            deck[i][j] = ranks[j];
        }
    }
    // Print a specific card (King of Clubs: deck[2][12])
    printf("deck[2][12] shows the %s of %s\n", deck[2][12], suits[2]);
return 0;
}
```

Review Questions

Q1 Answer each of the following:

a) A pointer variable contains as its value another variable's _____.

b) Three values can be used to initialize a pointer—_____, _____ and _____.

c) The only integer that can be assigned to a pointer is _____.

Q2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) A pointer that's declared to be `void` can be dereferenced.

b) Pointers of different types may not be assigned to one another without a cast operation.

Review Questions

Q3 Answer each of the following. Assume that single-precision floating-point numbers are stored in four bytes, and that the array's starting address is location 1002500 in memory. Each part of the exercise should use the results of previous parts where appropriate.

a) Define a `float` array called `numbers` with 10 elements, and initialize the elements to the values `0.0`, `1.1`, `2.2`, ..., `9.9`. Assume the symbolic constant `SIZE` has been defined as `10`.

b) Define a pointer, `nPtr`, that points to a `float`.

c) Use a `for` statement and array subscript notation to print array `numbers`' elements. Use one digit of precision to the right of the decimal point.

d) Give two separate statements that assign the starting address of array `numbers` to the pointer variable `nPtr`.

Review Questions

e) Print `numbers`' elements using pointer/offset notation with the pointer `nPtr`.
f) Print `numbers`' elements using pointer/offset notation with the array name as the pointer.
g) Print `numbers`' elements by subscripting pointer `nPtr`.
h) Refer to element 4 of `numbers` using array subscript notation, pointer/offset notation with the array name as the pointer, pointer subscript notation with `nPtr` and pointer/offset notation with `nPtr`.
i) Assuming that `nPtr` points to the beginning of array `numbers`, what address is referenced by `nPtr + 8`? What value is stored at that location?
j) Assuming that `nPtr` points to `numbers[5]`, what address is referenced by `nPtr -= 4`? What's the value stored at that location?

Review Questions

Q4 For each of the following, write a statement that performs the specified task. Assume that `float` variables `number1` and `number2` are defined and that `number1` is initialized to `7.3`.

a) Define the variable `fPtr` to be a pointer to an object of type `float`.
b) Assign the address of variable `number1` to pointer variable `fPtr`.
c) Print the value of the object pointed to by `fPtr`.
d) Assign the value of the object pointed to by `fPtr` to variable `number2`.
e) Print the value of `number2`.
f) Print the address of `number1`. Use the `%p` conversion specification.
g) Print the address stored in `fPtr`. Use the `%p` conversion specifier. Is the value printed the same as the address of `number1`?