POINTERS II

Algorithms and Programming II

1

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 these expressions are valid in conjunction with pointer variables.
- This section describes the operators that can have pointers as operands, and how these operators are used.
- A limited set of arithmetic operations may be performed on pointers.

2

Pointer Expressions and Pointer Arithmetic

- A pointer may be incremented (++) or decremented (- -)
- An integer may be added to a pointer (+ or +=)
- An integer may be subtracted from a pointer (- or - =)
- One pointer may be subtracted from another
  - *this last operation is meaningful only when both pointers point to elements of the same array.*

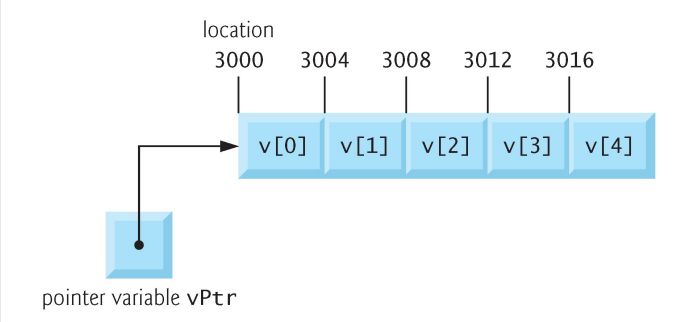
3

Pointer Expressions and Pointer Arithmetic

- 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.*
- Next figure 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

4

Pointer Expressions and Pointer Arithmetic



5

Pointer Expressions and Pointer Arithmetic

- 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.

6

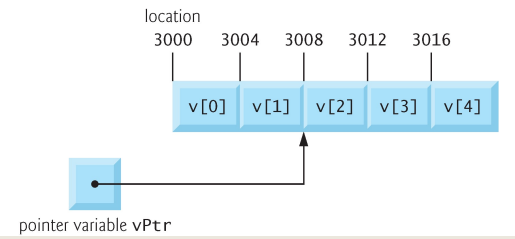
Pointer Expressions and Pointer Arithmetic

- For example, the statement

  vPtr += 2;

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

- In the array v, vPtr would now point to v[2].



7

Pointer Expressions and Pointer Arithmetic

- If an integer is 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.
- When performing pointer arithmetic on a character array, the results will be consistent with regular arithmetic, because each character is 1 byte long.

8

Pointer Expressions and Pointer Arithmetic

- 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.
- Either of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next location in the array.

9

Pointer Expressions and Pointer Arithmetic

- Either of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the next location in the array.

- Either of the statements

```
--vPtr;
vPtr--;
```

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

10

Pointer Expressions and Pointer Arithmetic

- 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).

11

Pointer Expressions and Pointer Arithmetic

- Pointer arithmetic is undefined unless performed on an array.
- We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of an array.

12

Pointer Expressions and Pointer Arithmetic

- 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.
- In both cases, a cast operation is not required.
- A pointer to void cannot be dereferenced.

13

Pointer Expressions and Pointer Arithmetic

- 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.

14

Pointer Expressions and Pointer Arithmetic

- 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.

15

Pointer Expressions and Pointer Arithmetic

- 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-numbered element of the array than the other pointer does.
- A common use of pointer comparison is determining whether a pointer is NULL.

16

Relationship between Pointers and Arrays

- Arrays and pointers are intimately related in C and often may be used interchangeably.
- An array name can be thought of as a constant pointer.
- Pointers can be used to do any operation involving array indexing.
- 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;
```

17

Relationship between Pointers and Arrays

- 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.**

18

Relationship between Pointers and Arrays

- 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.**

19

Relationship between Pointers and Arrays

- 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
```

- The array itself can be treated as a pointer and used in pointer arithmetic.

20

Relationship between Pointers and Arrays

- For example, the expression

```
*(b + 3)
```

also refers to the array element b[3].

- In general, all indexed array expressions can be written with a pointer and an offset.
- 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.
- Pointers can be indexed like arrays.

21

Relationship between Pointers and 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.

22

Relationship between Pointers and Arrays

- Next program 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, and pointer/offset with a pointer—to print the four elements of the integer array b.

23

Relationship between Pointers and Arrays

```c
/*  Using subscripting and pointer notations with arrays */

#include <stdio.h>

int main()
{
   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 */
```

24

Relationship between Pointers and Arrays

```
19
20    /* output array b using array name and pointer/offset notation */
21    printf( "\nPointer/offset notation where\n"
22            "the pointer is the array name\n" );
23
24    /* loop through array b */
25    for ( offset = 0; offset < 4; offset++ ) {
26       printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
27    } /* end for */
```

25

Relationship between Pointers and Arrays

```
28
29    /* output array b using bPtr and array subscript notation */
30    printf( "\nPointer subscript notation\n" );
31
32    /* loop through array b */
33    for ( i = 0; i < 4; i++ ) {
34       printf( "bPtr[ %d ] = %d\n", i, bPtr[ i ] );
35    } /* end for */
```

26

Relationship between Pointers and Arrays

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

27

Relationship between Pointers and Arrays

```
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
```

28

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"};
```

29

Arrays of Pointers

- 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.

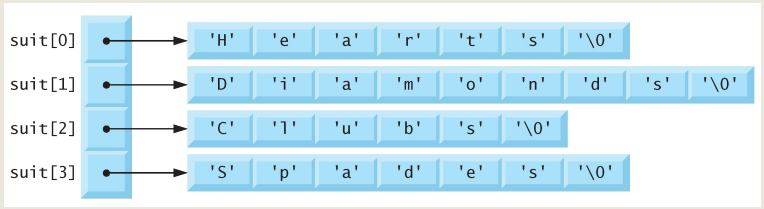
30

Arrays of Pointers

- The four strings are 7, 9, 6 and 7 characters long, respectively.
- Although it appears as though these strings are being placed in the suit array, only pointers are actually stored in the array 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.
- This flexibility is one example of C's powerful data-structuring capabilities.

31

Arrays of Pointers



32

Arrays of Pointers

- 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.

33

Pointers to Functions

- A pointer to a function contains the address of the function in memory.
- We saw that an array name is really the address in memory of the first element of the array.
- Similarly, a function name is really the starting address in memory of the code that performs the function's task.
- Pointers to functions can be passed to functions, returned from functions, stored in arrays and assigned to other function pointers.

34

Pointers to Functions

- Bubble Sort example.

35

Pointers to Functions

- The following parameter appears in the function header for bubble

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

- This tells bubble to expect a parameter (compare) that's a pointer to a function that receives two integer parameters and returns an integer result

36

Pointers to Functions

- Parentheses are needed around *compare to group the * with compare to indicate that compare is a pointer.
- If we had not included the parentheses, the declaration would have been

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

which declares a function that receives two integers as parameters and returns a pointer to an integer.

37

Pointers to Functions

- The third parameter in the prototype could have been written as

```
int (*)(int, int);
```

without the function-pointer name and parameter names.

- The function passed to bubble is called in an if statement as follows:

```
if ((*compare)(work[count], work[count + 1]))
```

- Just as a pointer to a variable is dereferenced to access the value of the variable, a pointer to a function is dereferenced to use the function

38

Pointers to Functions

- The call to the function could have been made without dereferencing the pointer as in

```
if (compare(work[count], work[count + 1]))
```

which uses the pointer directly as the function name.

- We prefer the first method of calling a function through a pointer because it explicitly illustrates that compare is a pointer to a function that's dereferenced to call the function.
- The second method of calling a function through a pointer makes it appear as if compare is an actual function.
- This may be confusing to a programmer reading the code who would like to see the definition of function compare and finds that it's never defined in the file.

39

References

- Problem Solving and Program Design in C —7th ed, Jeri R. Hanly, Elliot B. Koffman
- C How to Program, 8ed, by Paul Deitel and Harvey Deitel, Pearson, 2016.

40