POINTERS

Algorithms and Programming II

1

Pointers

- In this chapter, we discuss one of the most powerful features of the C programming language, the pointer.
- Pointers enable programs to simulate pass-by-reference, to pass functions between functions, and to create and manipulate dynamic data structures, i.e., data structures that can grow and shrink at execution time, such as linked lists, queues, stacks and trees.

2

Definitions and Initialization

- Pointers are variables whose values are *memory addresses*.
- Normally, a variable directly contains a specific value.
- A pointer, on the other hand, contains an *address* of a variable that contains a specific value.
- In this sense, a variable name *directly* references a value, and a pointer *indirectly* references a value
- Referencing a value through a pointer is called indirection.

3

Definitions and Initialization

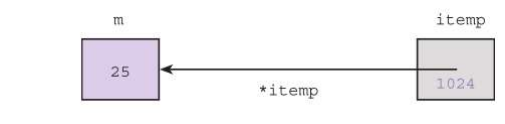
- pointer (pointer variable)
  - *a memory cell that stores the address of a data item*
  - *syntax:* *type *variable*

```
int m = 25;
int *itemp;    /* a pointer to an integer */
```

4

Indirection

- indirect reference
  - *accessing the contents of a memory cell through a pointer variable that stores it address*



5

Declaring Pointers

- Pointers, like all variables, must be defined before they can be used.
- The definition

  int *countPtr, count;

specifies that variable countPtr is of type int * (i.e., a pointer to an integer) and is read (right to left), “countPtr is a pointer to int” or “countPtr points to an object of type int.”

- Also, the variable count is defined to be an int, not a pointer to an int.

6

Declaring Pointers

- The * applies only to countPtr in the definition.
- When * is used in this manner in a definition, it indicates that the variable being defined is a pointer.
- Pointers can be defined to point to objects of any type.
- To prevent the ambiguity of declaring pointer and non-pointer variables in the same declaration as shown above, you should always declare only one variable per declaration.

7

Initializing Pointers

- Pointers should be initialized when they're defined or 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.
- NULL is a symbolic constant defined in the <stddef.h> header (and several other headers, such as <stdio.h>).

8

Initializing Pointers

- Initializing a pointer to 0 is equivalent to initializing a pointer to NULL, but NULL is preferred.
- 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.

9

Pointer Operators

- The &, or address operator, is a unary operator that returns the address of its operand.
- For example, assuming the definitions

```
int y = 5;
int *yPtr;
```

the statement

```
yPtr = &y;
```

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

- Variable yPtr is then said to "point to" y.

10

The Indirection Operator

- The unary * operator, commonly referred to as the indirection operator or dereferencing operator, returns the *value* of the object to which its operand (i.e., a pointer) points.
- For example, the statement
  - `printf("%d", *yPtr);`

  prints the value of variable y, namely 5.
- Using * in this manner is called dereferencing a pointer.

11

Pointer Operators

- Next Figure demonstrates the pointer operators & and *.
- The printf conversion specifier %p outputs the memory location as a *hexadecimal* integer on most platforms.
- Notice that the *address* of a and the *value* of aPtr are identical in the output, thus confirming that the address of a is indeed assigned to the pointer variable aPtr
- The & and * operators are complements of one another—when they're both applied consecutively to aPtr in either order, the same result is printed.

12

Pointer Operators

```c
// Fig. 7.4: fig07_04.c
// Using the & and * pointer operators.
#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);
}
```

13

Pointer Operators

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

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

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

14

Precedence of the Operators

Operators	Associativity	Type
`()` `[]` `++` *(postfix)* `--` *(postfix)*	left to right	postfix
`+` `-` `++` `--` `!` `*` `&` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

15

Precedence of the Operators

Operators	Associativity	Type
`()` `[]` `++` *(postfix)* `--` *(postfix)*	left to right	postfix
`+` `-` `++` `--` `!` `*` `&` (*type*)	right to left	unary
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>` `s>=`	left to right	relational
`==` `!=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`=` `+=` `-=` `*=` `/=` `%=`	right to left	assignment
`,`	left to right	comma

16

Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—pass-by-value and pass-by-reference.
- All arguments in C are passed by value.
- Many functions require the capability to modify variables in the caller or to pass a pointer to a large data object to avoid the overhead of passing the object by value (which incurs the time and memory overheads of making a copy of the object).
- In C, you use pointers and the indirection operator to simulate pass-by-reference.

17

Passing Arguments to Functions by Reference

- When calling a function with arguments that should be modified, the addresses of the arguments are passed.
- This is normally accomplished by applying the address operator (&) to the variable (in the caller) whose value will be modified.
- As we saw in previous chapters, arrays are not passed using operator & because C automatically passes the starting location in memory of the array (the name of an array is equivalent to &arrayName[0]).
- When the address of a variable is passed to a function, the indirection operator (*) may be used in the function to modify the value at that location in the caller's memory.

18

Pass-By-Value

```c
// Fig. 7.6: fig07_06.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", number);

   // pass number by value to cubeByValue
   number = cubeByValue(number);

   printf("\nThe new value of number is %d\n", number);
}

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

19

Pass-By-Value

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

20

Pass-By-Reference

```c
// Fig. 7.7: fig07_07.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", number);

   // pass address of number to cubeByReference
   cubeByReference(&number);

   printf("\nThe new value of number is %d\n", number);
}

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

21

Pass-By-Reference

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

22

Passing Arguments to Functions by Reference

- A function receiving an address as an argument must define a pointer parameter to receive the address.
- For example, the header for function cubeByReference is:

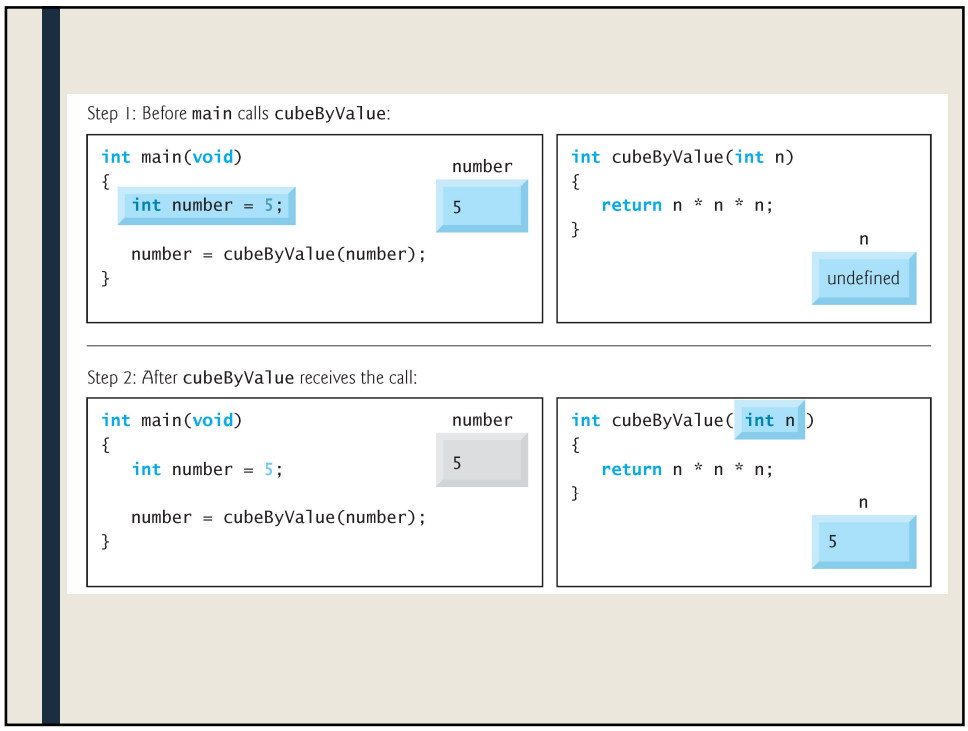
  void cubeByReference(int *nPtr)
- The header specifies that cubeByReference receives the address of an integer variable as an argument, stores the address locally in nPtr and does not return a value.
- The function prototype for cubeByReference contains int * in parentheses.

23

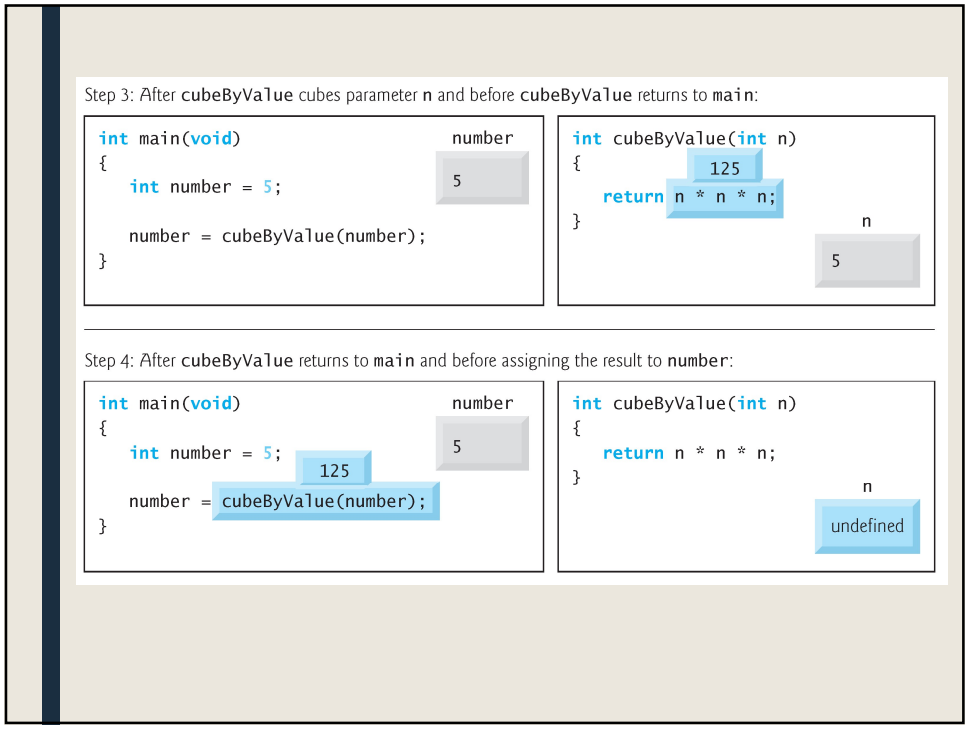
Passing Arguments to Functions by Reference

- For a function that expects a one-dimensional array as an argument, the function's prototype and header can use the pointer notation shown in the parameter list of function cubeByReference.
- The compiler does not differentiate between a function that receives a pointer and one that receives a one-dimensional array.
- This, of course, means that the function must "know" when it's receiving an array or simply a single variable for which it's to perform pass-by-reference.
- When the compiler encounters a function parameter for a one-dimensional array of the form int b[], the compiler converts the parameter to the pointer notation int *b.
- The two forms are interchangeable.

24

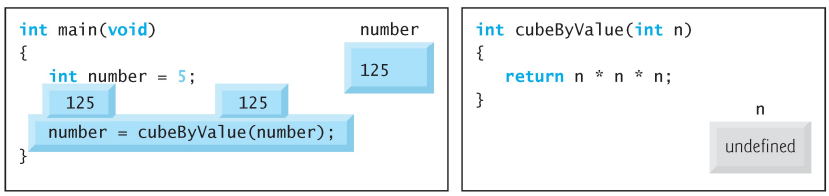

Step 1: Before main calls cubeByValue:
int main(void)
{
int number = 5;
number = cubeByValue(number);
}
number
5
int cubeByValue(int n)
{
return n * n * n;
}
n
undefined
Step 2: After cubeByValue receives the call:
int main(void)
{
int number = 5;
number = cubeByValue(number);
}
number
5
int cubeByValue(int n)
{
return n * n * n;
}
n
5

25


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)
{
125
return n * n * n;
}
n
5
Step 4: After cubeByValue returns to main and before assigning the result to number:
int main(void)
{
int number = 5;
125
number = cubeByValue(number);
}
number
5
int cubeByValue(int n)
{
return n * n * n;
}
n
undefined

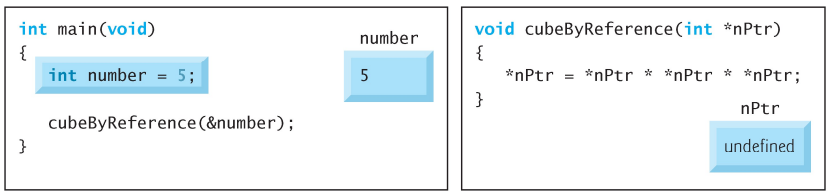
26

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

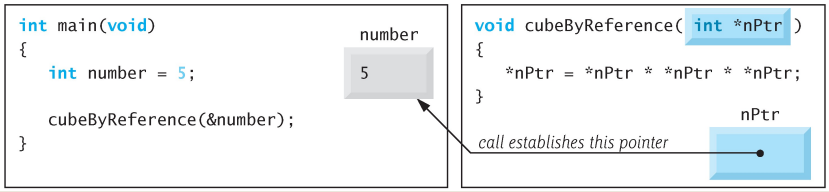


27

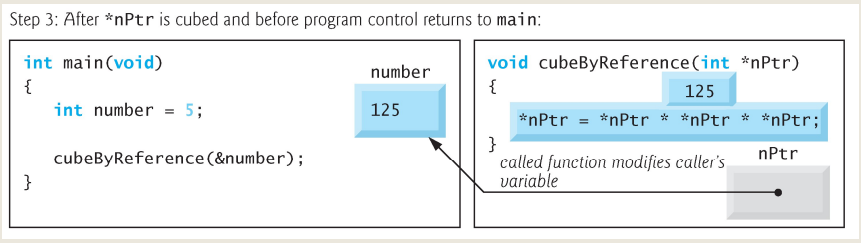
Step 1: Before `main` calls `cubeByReference`:



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



28



29

Bubble Sort Using Pass-by-Reference

- Let's write a bubble sort program using two functions—`bubbleSort` and `swap`.
- Function `bubbleSort` sorts the array.
- It calls function `swap` to exchange the array elements `array[j]` and `array[j + 1]`
- Remember that C enforces *information hiding* between functions, so `swap` does not have access to individual array elements in `bubbleSort`.
- Because `bubbleSort` *wants* `swap` to have access to the array elements to be swapped, `bubbleSort` passes each of these elements *by reference* to `swap`—the *address* of each array element is passed explicitly.

30

Bubble Sort Using Pass-by-Reference

- Although entire arrays are automatically passed by reference, individual array elements are scalars and are ordinarily passed by value.
- Therefore, `bubbleSort` uses the address operator (`&`) on each of the array elements in the `swap` call to effect pass-by-reference as follows
  - `swap(&array[j], &array[j + 1]);`
- Function `swap` receives `&array[j]` in pointer variable `element1Ptr`.

31

Bubble Sort Using Pass-by-Reference

- Even though `swap`—because of information hiding—is *not* allowed to know the name `array[j]`, `swap` may use `*element1Ptr` as a *synonym* for `array[j]`—when `swap` references `*element1Ptr`, it's *actually* referencing `array[j]` in `bubbleSort`.
- Similarly, when `swap` references `*element2Ptr`, it's actually referencing `array[j + 1]` in `bubbleSort`

32

Bubble Sort Using Pass-by-Reference

- Even though `swap` is not allowed to say

```
int hold = array[j];
array[j] = array[j + 1];
array[j + 1] = hold;
```

precisely the *same* effect is achieved by

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

33

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.

34