BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-1

Pointers

Instructor : PhD, Associate Professor Leyla Muradkhanli

Introduction

- In this topic, we discuss one of the most powerful features of the C programming language, the pointer.
- Pointers enable programs to simulate call-by-reference 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.

Pointer Variable 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 (Fig. 7.1).
- Referencing a value through a pointer is called indirection.

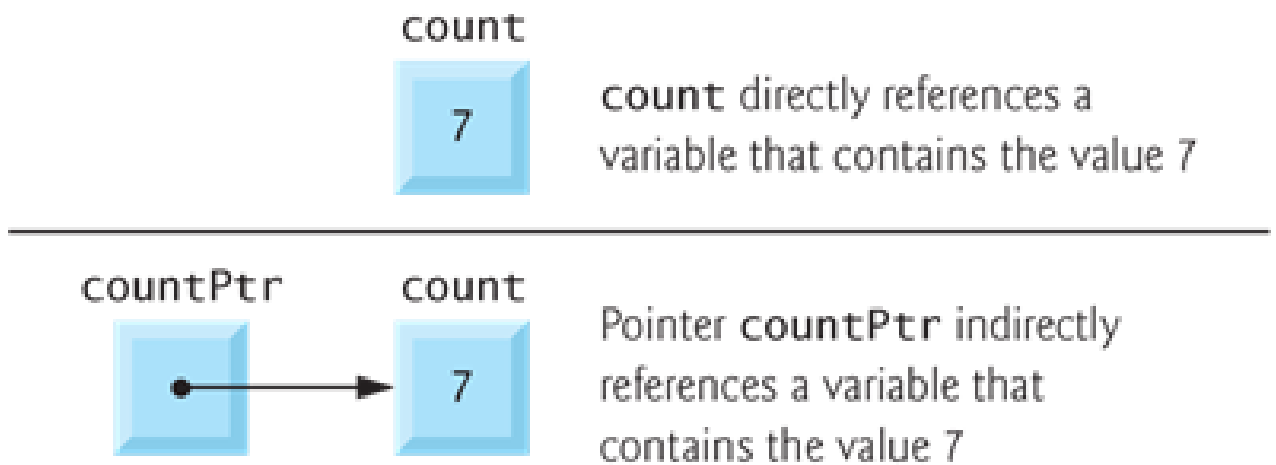


Fig. 7.1 | Directly and indirectly referencing a variable.

Pointer Variable Definitions and Initialization (cont.)

- 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, "`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`.
- The * only applies 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.

Pointer Variable Definitions and Initialization (cont.)

- Pointers should be initialized either when they're defined or in an assignment statement.
- 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>`).
- 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.

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`.
- Figure 7.2 shows a schematic representation of memory after the preceding assignment is executed.

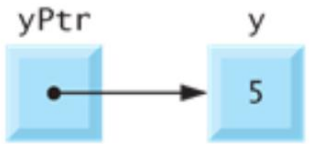


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.
    ```

Pointer Operators (Cont.)

- Figure 7.3 shows the representation of the pointer in memory, assuming that integer variable `y` is stored at location 600000, and pointer variable `yPtr` is stored at location 500000.
- The operand of the address operator must be a variable; the address operator cannot be applied to constants, to expressions or to variables declared with the storage-class `register`.

Fig. 7.3 | Representation of `y` and `yPtr` in memory.

Pointer Operators (Cont.)

- 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.

Pointer Operators (Cont.)

- Figure 7.4 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` (line 11).
- The `&` and `*` operators are complements of one another—when they're both applied consecutively to `aPtr` in either order (line 21), the same result is printed.
- Figure 7.5 lists the precedence and associativity of the operators introduced to this point.

```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 */
```

Fig. 7.4 | Using the & and * pointer operators. (Part 1 of 2.)

```
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
```

Fig. 7.4 | Using the & and * pointer operators. (Part 2 of 2.)

Passing Arguments to Functions by Reference

- There are two ways to pass arguments to a function—call-by-value and call-by-reference.
- All arguments in C are passed by value.
- Many functions require the capability to modify one or more 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 overhead of making a copy of the object).
- For these purposes, C provides the capabilities for simulating call-by-reference.
- In C, you use pointers and the indirection operator to simulate call-by-reference.

Passing Arguments to Functions by Reference (Cont.)

- 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 topic, 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.

Passing Arguments to Functions by Reference (Cont.)

- The programs in Fig. 7.6 and Fig. 7.7 present two versions of a function that cubes an integer—`cubeByValue` and `cubeByReference`.
- Figure 7.6 passes the variable `number` to function `cubeByValue` using call-by-value (line 14).
- The `cubeByValue` function cubes its argument and passes the new value back to `main` using a `return` statement.
- The new value is assigned to `number` in `main` (line 14).

```c
/* Fig. 7.6: fig07_06.c
   Cube a variable using call-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 );
   return 0; /* indicates successful termination */
} /* end main */

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

Fig. 7.6 | Cube a variable using call-by-value. (Part 1 of 2.)

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

Fig. 7.6 | Cube a variable using call-by-value. (Part 2 of 2.)

Passing Arguments to Functions by Reference (Cont.)

- Figure 7.7 passes the variable `number` using call-by-reference (line 15) - the address of `number` is passed—to function `cubeByReference`.
- Function `cubeByReference` takes as a parameter a pointer to an `int` called `nPtr` (line 22).
- The function dereferences the pointer and cubes the value to which `nPtr` points (line 24), then assigns the result to `*nPtr` (which is really `number` in `main`), thus changing the value of `number` in `main`.
- Figure 7.8 and Fig. 7.9 analyze graphically the programs in Fig. 7.6 and Fig. 7.7, respectively.

```c
/* Fig. 7.7: fig07_07.c
   Cube a variable using call-by-reference with a pointer argument */

#include <stdio.h>

void cubeByReference( int *nPtr ); /* 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 );
   return 0; /* indicates successful termination */
} /* end main */

```

Fig. 7.7 | Cube a variable using call-by-reference with a pointer argument. (Part 1 of 2.)

```
21  /* calculate cube of *nPtr; modifies variable number in main */
22  void cubeByReference( int *nPtr )
23  {
24     *nPtr = *nPtr * *nPtr * *nPtr; /* cube *nPtr */
25  } /* end function cubeByReference */
```

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

Fig. 7.7 | Cube a variable using call-by-reference with a pointer argument. (Part 2 of 2.)

Passing Arguments to Functions by Reference (Cont.)

- A function receiving an address as an argument must define a pointer parameter to receive the address.
- For example, in Fig. 7.7 the header for function `cubeByReference` (line 22) 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.
- Names included for documentation purposes are ignored by the C compiler.

Passing Arguments to Functions by Reference (Cont.)

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

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 );
}
```

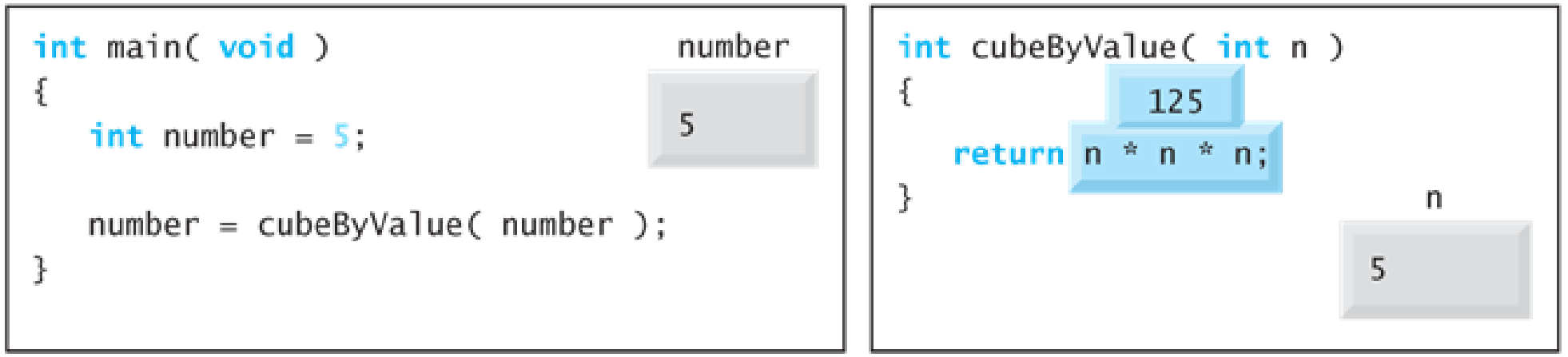
number: 5

```
int cubeByValue( int n )
{
   return n * n * n;
}
```

n: 5

Fig. 7.8 | Analysis of a typical call-by-value. (Part 1 of 3.)

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



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

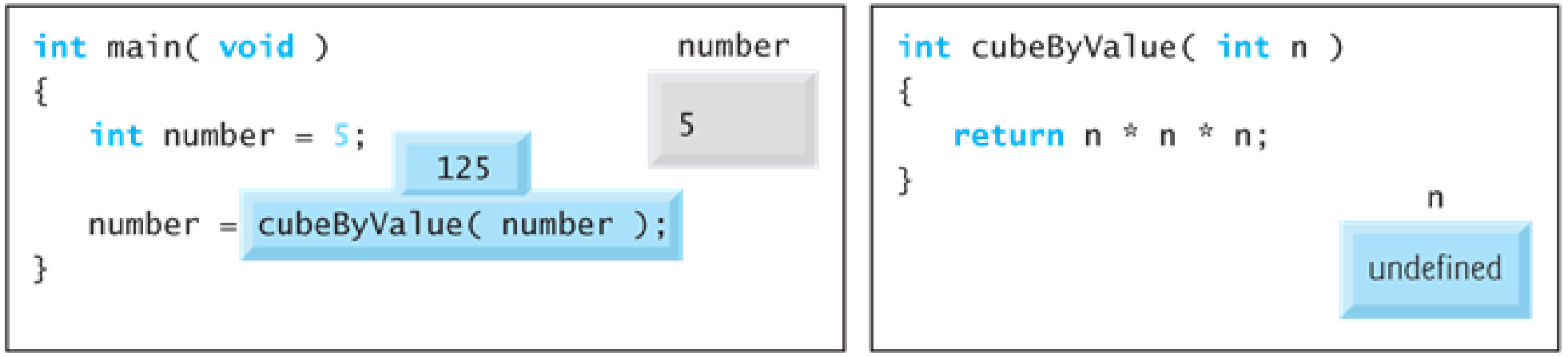


Fig. 7.8 | Analysis of a typical call-by-value. (Part 2 of 3.)

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

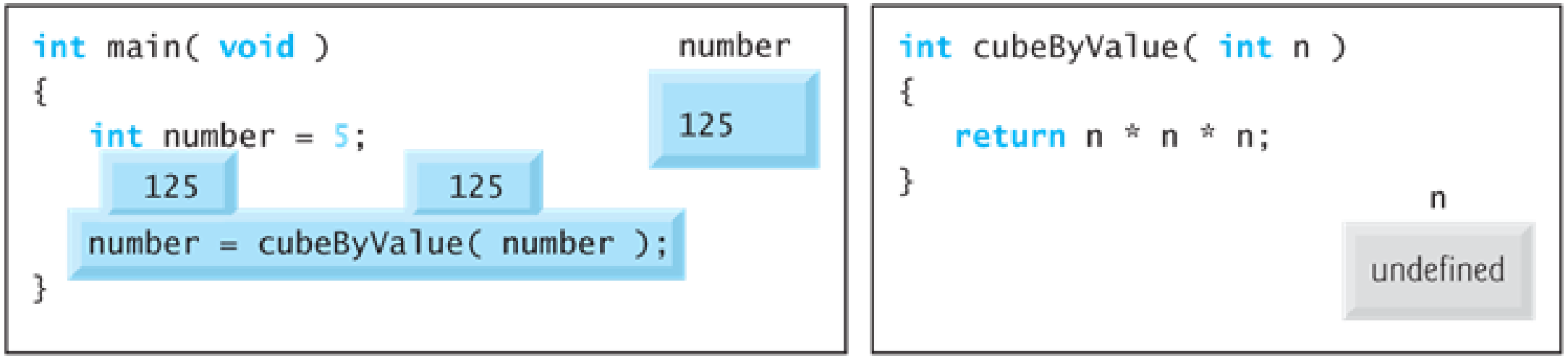


Fig. 7.8 | Analysis of a typical call-by-value. (Part 3 of 3.)

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

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr: undefined

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

```
int main( void )
{
   int number = 5;

   cubeByReference( &number );
}
```

number: 5

```
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;
}
```

nPtr

call establishes this pointer

Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

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

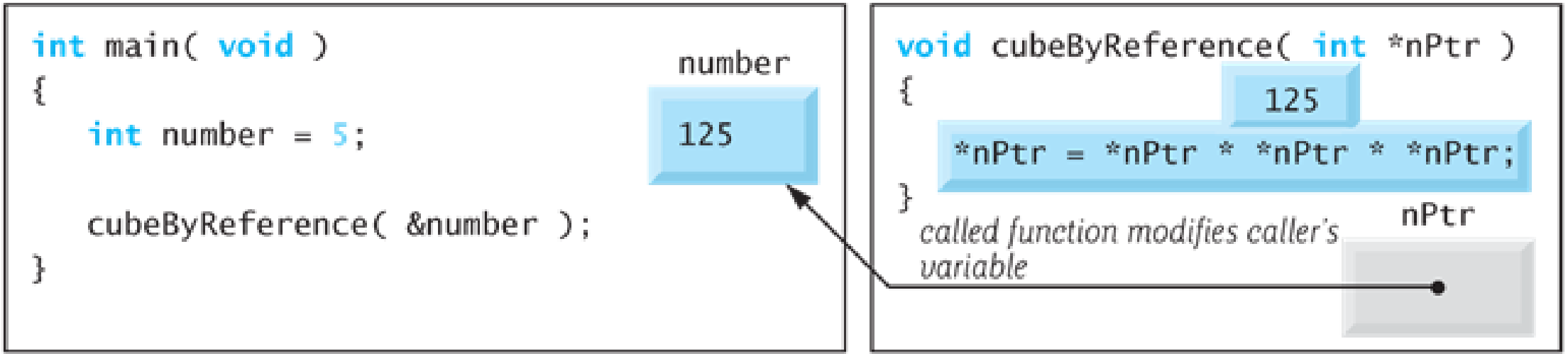


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

sizeof Operator

- C provides the special unary operator **`sizeof`** to determine the size in bytes of an array (or any other data type) during program compilation.
- When applied to the name of an array, the `sizeof` operator returns the total number of bytes in the array as an integer.

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 ] )`

sizeof Operator (Cont.)

- Figure 7.17 calculates the number of bytes used to store each of the standard data types.
- The results could be different between computers.

```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 */

   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"
```

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

```
23              "\n    sizeof ld = %d\tsizeof(long double) = %d"
24              "\n sizeof array = %d"
25              "\n   sizeof ptr = %d\n",
26             sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
27             sizeof( int ), sizeof l, sizeof( long ), sizeof f,
28             sizeof( float ), sizeof d, sizeof( double ), sizeof ld,
29             sizeof( long double ), sizeof array, sizeof ptr );
30      return 0; /* indicates successful termination */
31   } /* 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
```

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

sizeof Operator (Cont.)

- Operator `sizeof` can be applied to any variable name, type or value (including the value of an expression).
- When applied to a variable name (that is not an array name) or a constant, the number of bytes used to store the specific type of variable or constant is returned.
- The parentheses used with `sizeof` are required if a type name with two words is supplied as its operand (such as `long double` or `unsigned short`).
- Omitting the parentheses in this case results in a syntax error.
- The parentheses are not required if a variable name or a one-word type name is supplied as its operand, but they can still be included without causing an error.

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.
- 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 -=) and one pointer may be subtracted from another.

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

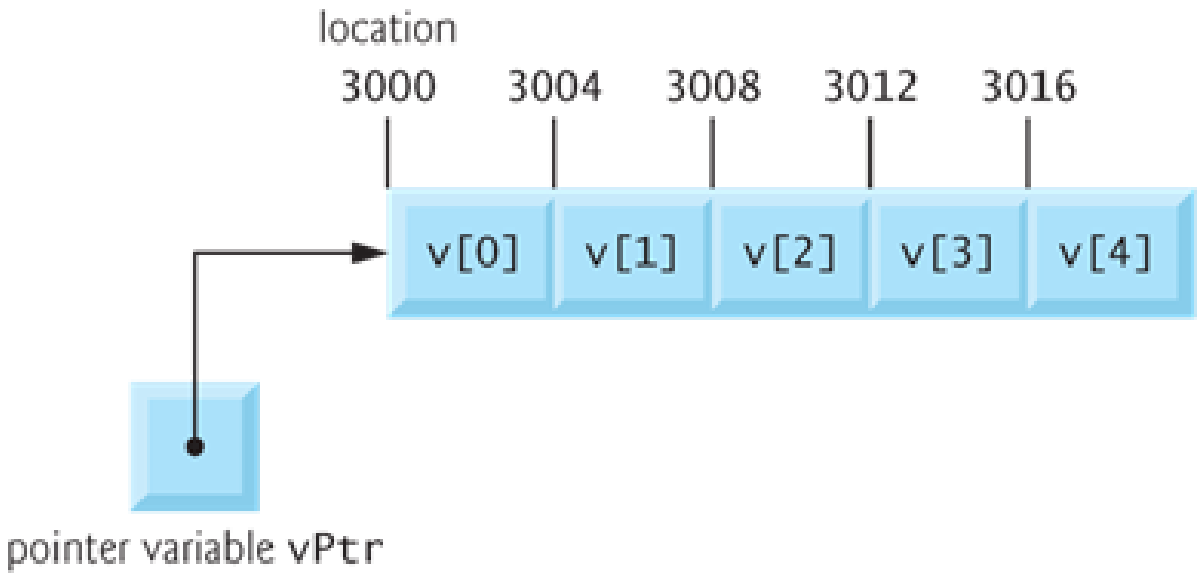


Fig. 7.18 | Array `v` and a pointer variable `vPtr` that points to `v`.

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.

Pointer Expressions and Pointer Arithmetic (Cont.)

- In the array `v`, `vPtr` would now point to `v[2]` (Fig. 7.19).
- 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.

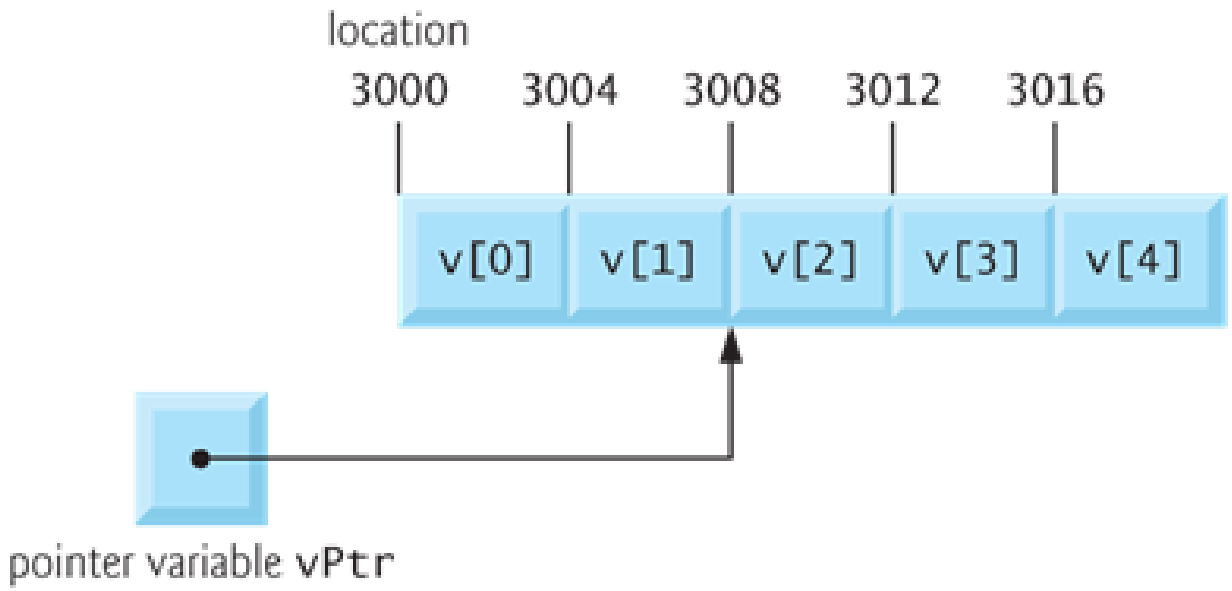


Fig. 7.19 | The pointer `vPtr` after pointer arithmetic.

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

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

Pointer Expressions and Pointer Arithmetic (Cont.)

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

  decrements the pointer to point to the previous element of the array.
- Pointer variables may be subtracted from one another.
    ```

Pointer Expressions and Pointer Arithmetic (Cont.)

- 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 meaningless 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.

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

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.

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 subscripting.
- Assume that integer array `b[5]` and integer pointer variable `bPtr` have been defined.
- Since the array name (without a subscript) 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;`

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 above expression is the offset to the pointer.
- When the pointer points to the beginning of an array, the offset indicates which element of the array should be referenced, and the offset value is identical to the array subscript.
- The preceding notation is referred to as pointer/offset notation.

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

Relationship between Pointers and Arrays (Cont.)

- For example, the expression
  - `*( b + 3 )`

  also refers to the array element `b[3]`.
- In general, all subscripted 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 subscripted exactly as arrays can.

Relationship between Pointers and Arrays (Cont.)

- For example, if `bPtr` has the value `b`, the expression
  - `bPtr[ 1 ]`

  refers to the array element `b[1]`.
- This is referred to as pointer/subscript 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.

Relationship between Pointers and Arrays (Cont.)

- Figure 7.20 uses the four methods we have discussed for referring to array elements—array subscripting, pointer/offset with the array name as a pointer, pointer subscripting, and pointer/offset with a pointer—to print the four elements of the integer array b.

```
1  /* Fig. 7.20: fig07_20.cpp
2     Using subscripting and pointer notations with arrays */
3
4  #include <stdio.h>
5
6  int main( void )
7  {
8     int b[] = { 10, 20, 30, 40 }; /* initialize array b */
9     int *bPtr = b; /* set bPtr to point to array b */
10    int i; /* counter */
11    int offset; /* counter */
12
13    /* output array b using array subscript notation */
14    printf( "Array b printed with:\nArray subscript notation\n" );
15
16    /* loop through array b */
17    for ( i = 0; i < 4; i++ ) {
18       printf( "b[ %d ] = %d\n", i, b[ i ] );
19    } /* end for */
20
21    /* output array b using array name and pointer/offset notation */
22    printf( "\nPointer/offset notation where\n"
23            "the pointer is the array name\n" );
```

Fig. 7.20 | Using four methods of referencing array elements. (Part 1 of 3.)

```
24
25       /* loop through array b */
26       for ( offset = 0; offset < 4; offset++ ) {
27          printf( "*( b + %d ) = %d\n", offset, *( b + offset ) );
28       } /* end for */
29
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    } /* end main */
```

Fig. 7.20 | Using four methods of referencing array elements. (Part 2 of 3.)

```
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
```

Fig. 7.20 | Using four methods of referencing array elements. (Part 3 of 3.)