

C Pointers



© 2007 Pearson Education, Inc. All rights reserved.

7.1 Introduction

- **Pointers**
  - **Powerful, but difficult to master**
  - **Simulate call-by-reference**
  - **Close relationship with arrays and strings**



© 2007 Pearson Education, Inc. All rights reserved.

7.2 Pointer Variable Definitions and Initialization

- **Pointer variables**
  - **Contain memory addresses as their values**
  - **Normal variables contain a specific value (direct reference)**
  - **Pointers contain address of a variable that has a specific value (indirect reference)**
  - **Indirection – referencing a pointer value**



© 2007 Pearson Education, Inc. All rights reserved.

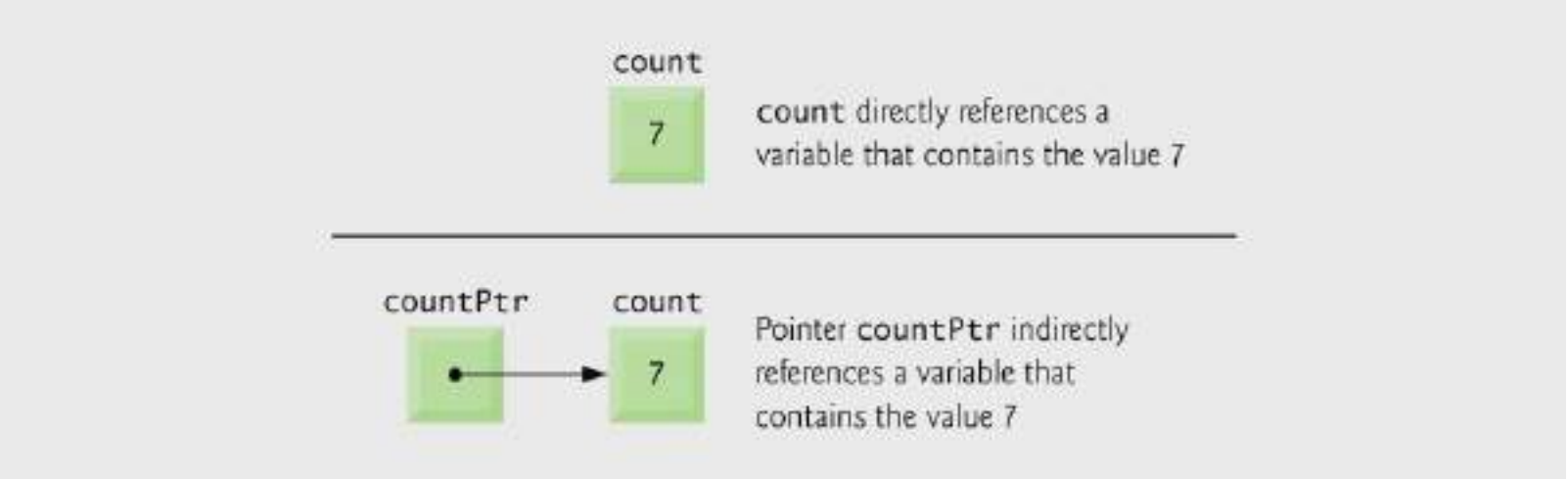


Fig. 7.1 | Directly and indirectly referencing a variable.

© 2007 Pearson Education, Inc. All rights reserved.

We've seen examples of both of these in our LC-3 programs; now we'll see them in C.

Pointer

- *Address of a variable in memory*
- *Allows us to indirectly access variables in.*
- *other words, we can talk about its address rather than its value*

Array

- *A list of values arranged sequentially in memory*
- *Example: a list of telephone numbers*
- *Expression* `a[4]` *refers to the 5th element of the array* `a`



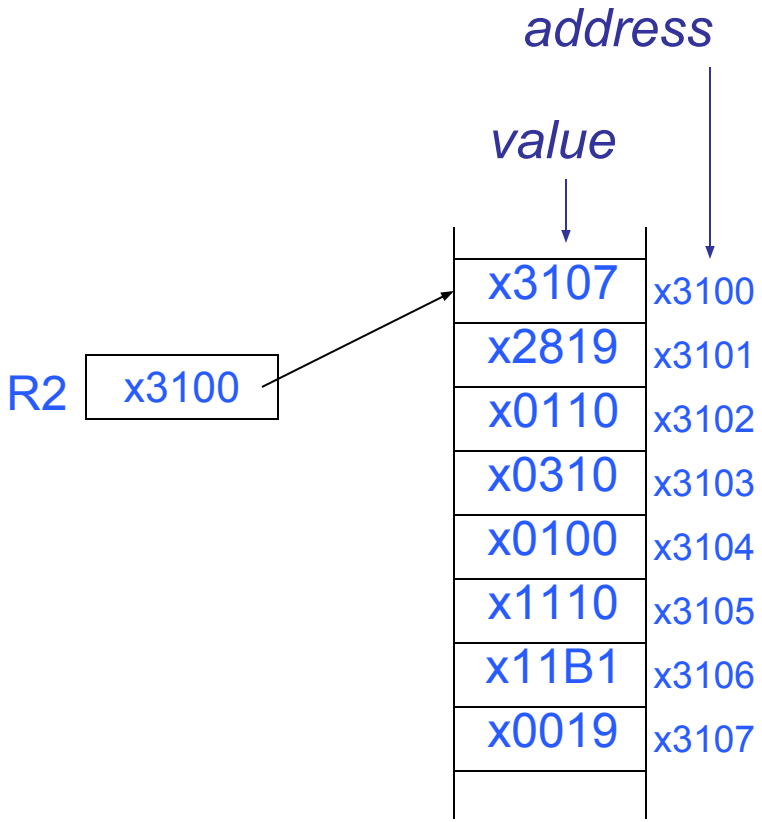
© 2007 Pearson Education, Inc. All rights reserved.

*Sometimes we want to deal with the **address** of a memory location, rather than the **value** it contains.*

Recall example from Chapter 6: adding a column of numbers.

- *R2 contains address of first location.*
- *Read value, add to sum, and increment R2 until all numbers have been processed.*

R2 is a pointer -- it contains the address of data we're interested in.



Address vs. Value

© 2007 Pearson Education, Inc. All rights reserved.

C lets us talk about and manipulate pointers as variables and in expressions.

Declaration

```
int *p;    /* p is a pointer to an int */
```

A pointer in C is always a pointer to a particular data type: `int*`, `double*`, `char*`, *etc.*

Operators

`*p` -- *returns the value pointed to by p*

`&z` -- *returns the address of variable z*



© 2007 Pearson Education, Inc. All rights reserved.

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
```

store the value 4 into the memory location associated with i

store the address of i into the memory location associated with ptr

read the contents of memory at the address stored in ptr

store the result into memory at the address stored in ptr

Example



© 2007 Pearson Education, Inc. All rights reserved.

7.2 Pointer Variable Definitions and Initialization

- **Pointer definitions**
  - * **used with pointer variables**

    ```
    int *myPtr;
    ```

  - **Defines a pointer to an int (pointer of type int *)**
  - **Multiple pointers require using a * before each variable definition**

    ```
    int *myPtr1, *myPtr2;
    ```

  - **Can define pointers to any data type**
  - **Initialize pointers to 0, NULL, or an address**
    - **0 or NULL – points to nothing (NULL preferred)**



© 2007 Pearson Education, Inc. All rights reserved.

7.3 Pointer Operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
yPtr "points to" y
```



© 2007 Pearson Education, Inc. All rights reserved.

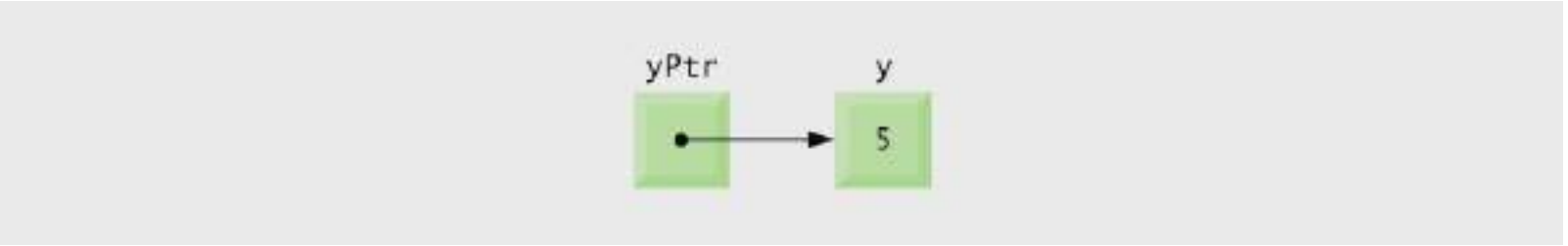


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.



© 2007 Pearson Education, Inc. All rights reserved.



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

© 2007 Pearson Education, Inc. All rights reserved.

7.3 Pointer Operators

- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - *yptr returns y (because yptr points to y)
  - * can be used for assignment
    - Returns alias to an object

```
*yptr = 7;  /* changes y to 7 */
```

  - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
  - They cancel each other out



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_04.c

(1 of 2)

```
1  /* Fig. 7.4: fig07_04.c
2     Using the & and * operators */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int a;        /* a is an integer */
8     int *aPtr;    /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;    /* aPtr set to address of a */
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr );
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr );
18
19    printf( "\n\nShowing that * and & are complements of "
20            "each other\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr );
22
23    return 0; /* indicates successful termination */
24
25 } /* end main */
```

If **aPtr** points to **a**, then **&a** and **aPtr** have the same value.

a and ***aPtr** have the same value

&*aPtr and ***&aPtr** have the same value



© 2007 Pearson Education, Inc. All rights reserved.

```
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
```

Outline

fig07_04.c

(2 of 2)



© 2007 Pearson Education, Inc. All rights reserved.

Operators	Associativity	Type
( ) [ ]	left to right	highest
+ - ++ -- ! * & (*type*)	right to left	unary
* /	left to right	multiplicative
+ -	left to right	additive
< <= > >=	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

Fig. 7.5 | Operator precedence.



© 2007 Pearson Education, Inc. All rights reserved.

Sometimes we want a pointer that points to nothing. In other words, we declare a pointer, but we're not ready to actually point to something yet.

```
int *p;
 p = NULL;  /* p is a null pointer
*/
```

NULL is a predefined macro that contains a value that a non-null pointer should never hold. Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms.

Null Pointer



© 2007 Pearson Education, Inc. All rights reserved.

7.4 Calling Functions by Reference

- **Call by reference with pointer arguments**
  - **Pass address of argument using & operator**
  - **Allows you to change actual location in memory**
  - **Arrays are not passed with & because the array name is already a pointer**
- *** operator**
  - **Used as alias/nickname for variable inside of function**

```
void double( int *number )
{
   *number = 2 * ( *number );
}
```

  - ***number used as nickname for the variable passed**



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_06.c

```
1  /* Fig. 7.6: fig07_06.c
2     Cube a variable using call-by-value */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* prototype */
6
7  int main( void )
8  {
9     int number = 5; /* initialize number */
10
11    printf( "The original value of number is %d", number );
12
13    /* pass number by value to cubeByValue */
14    number = cubeByValue( number );
15
16    printf( "\nThe new value of number is %d\n", number );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
22 /* calculate and return cube of integer argument */
23 int cubeByValue( int n )
24 {
25    return n * n * n;    /* cube local variable n and return result */
26
27 } /* end function cubeByValue */
```

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



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_07.c

```
1  /* Fig. 7.7: fig07_07.c
2     Cube a variable using call-by-reference with a pointer argument */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* prototype */
7
8  int main( void )
9  {
10    int number = 5; /* initialize number */
11
12    printf( "The original value of number is %d", number );
13
14    /* pass address of number to cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nThe new value of number is %d\n", number );
18
19    return 0; /* indicates successful termination */
20
21 } /* end main */
22
23 /* calculate cube of *nPtr; modifies variable number in main */
24 void cubeByReference( int *nPtr )
25 {
26    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube *nPtr */
27 } /* end function cubeByReference */
```

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

Function prototype takes a pointer argument

Function **cubeByReference** is passed an address, which can be the value of a pointer variable

In this program, ***nPtr** is **number**, so this statement modifies the value of **number** itself.



© 2007 Pearson Education, Inc. All rights reserved.

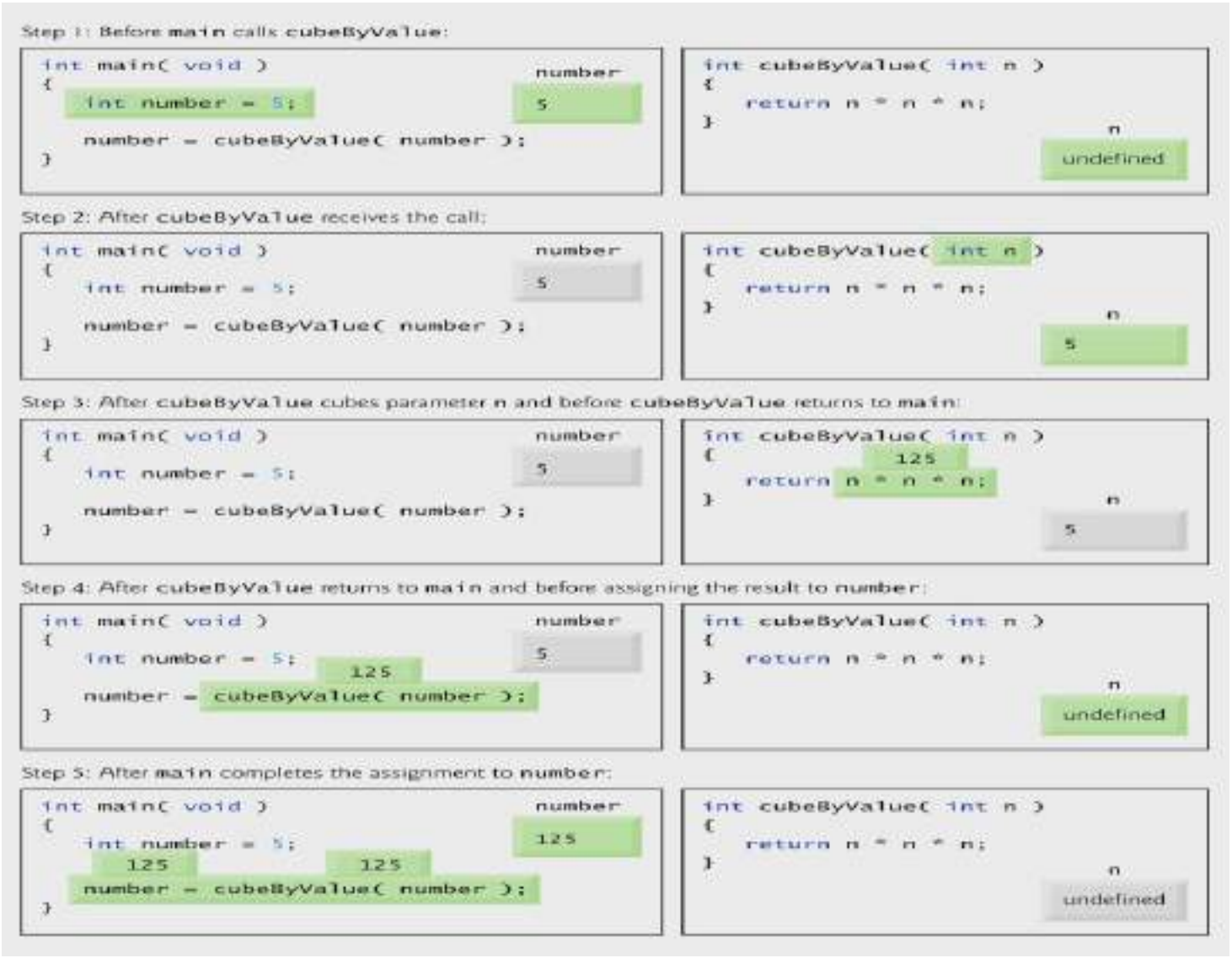


Fig. 7.8 | Analysis of a typical call-by-value.

© 2007 Pearson Education, Inc. All rights reserved.

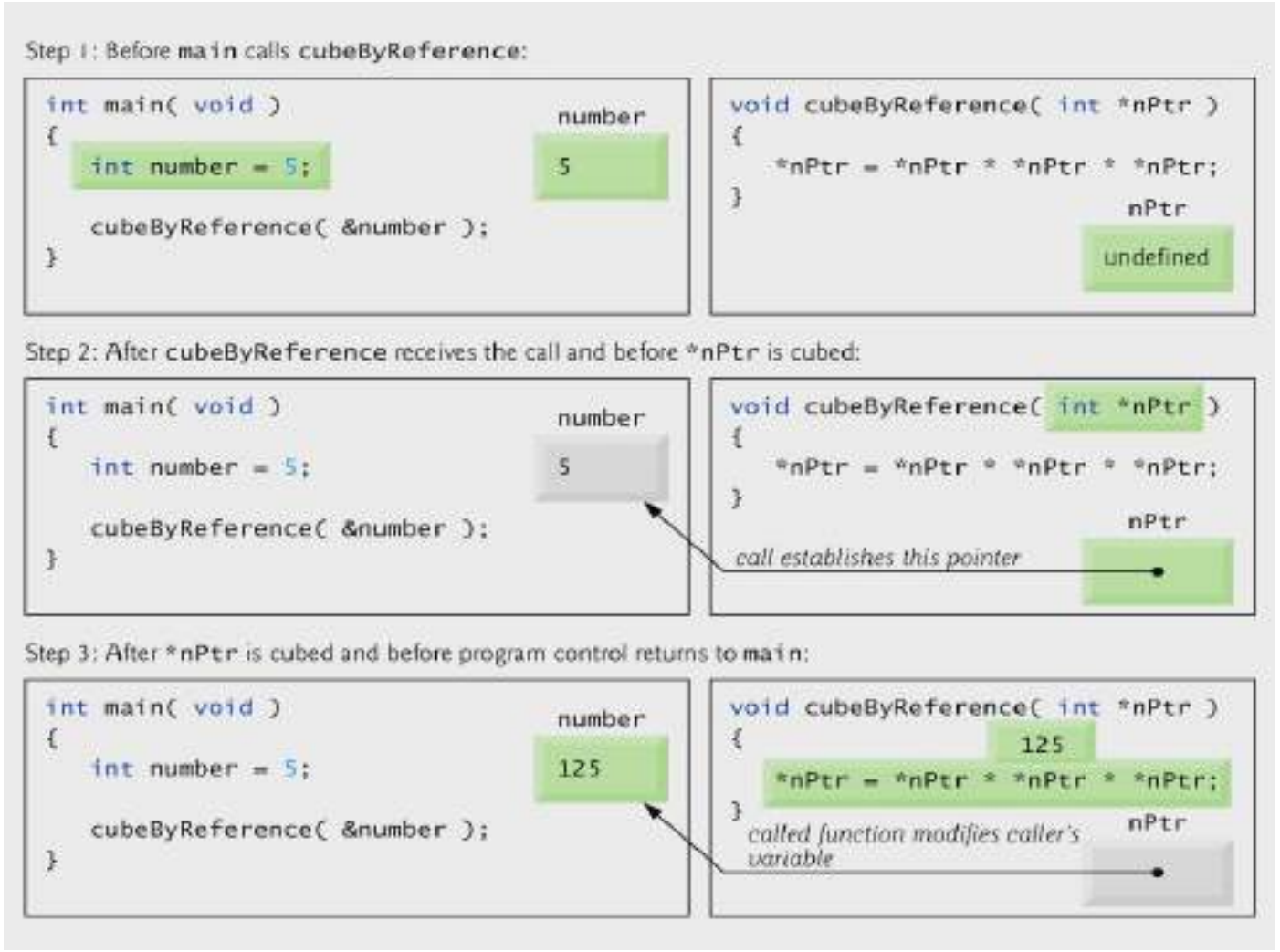


Fig. 7.9 | Analysis of a typical call-by-reference with a pointer argument.

© 2007 Pearson Education, Inc. All rights reserved.

7.5 Using the const Qualifier with Pointers

- const qualifier
  - Variable cannot be changed
  - Use const if function does not need to change a variable
  - Attempting to change a const variable produces an error
- const pointers
  - Point to a constant memory location
  - Must be initialized when defined
  - int *const myPtr = &x;
    - Type int *const – constant pointer to an int
  - const int *myPtr = &x;
    - Modifiable pointer to a const int
  - const int *const Ptr = &x;
    - const pointer to a const int
    - x itself can be changed, but not *Ptr



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_10.c

(1 of 2)

```
1  /* Fig. 7.10: fig07_10.c
2     Converting lowercase letters to uppercase letters
3     using a non-constant pointer to non-constant data */
4
5  #include <stdio.h>
6  #include <ctype.h>
7
8  void convertToUppercase( char *sPtr ); /* prototype */
9
10 int main( void )
11 {
12    char string[] = "characters and $32.98"; /* initialize char array */
13
14    printf( "The string before conversion is: %s", string );
15    convertToUppercase( string );
16    printf( "\nThe string after conversion is: %s\n", string );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

Both **sPtr** and ***sPtr** are modifiable



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_10.c

(2 of 2)

```
22 /* convert string to uppercase letters */
23 void convertToUppercase( char *sPtr )
24 {
25    while ( *sPtr != '\0' ) { /* current character is not '\0' */
26
27       if ( islower( *sPtr ) ) {    /* if character is lowercase, */
28          *sPtr = toupper( *sPtr ); /* convert to uppercase */
29       } /* end if */
30
31       ++sPtr;  /* move sPtr to the next character */
32    } /* end while */
33
34 } /* end function convertToUppercase */
```

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

Both **`sPtr`** and **`*sPtr`** are modified by the **`convertToUppercase`** function



© 2007 Pearson Education, Inc. All rights reserved.

Outline

```
1  /* Fig. 7.11: fig07_11.c
2     Printing a string one character at a time using
3     a non-constant pointer to constant data */
4
5  #include <stdio.h>
6
7  void printCharacters( const char *sPtr );
8
9  int main( void )
10 {
11    /* initialize char array */
12    char string[] = "print characters of a string";
13
14    printf( "The string is:\n" );
15    printCharacters( string );
16    printf( "\n" );
17
18    return 0; /* indicates successful termination */
19
20 } /* end main */
21
```

Pointer variable **`sPtr`** is modifiable, but the data to which it points, **`*sPtr`**, is not

fig07_11.c

(1 of 2)



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_11.c

(2 of 2)

```
22 /* sPtr cannot modify the character to which it points,
23    i.e., sPtr is a "read-only" pointer */
24 void printCharacters( const char *sPtr )
25 {
26    /* loop through entire string */
27    for ( ; *sPtr != '\0'; sPtr++ ) { /* no initialization */
28       printf( "%c", *sPtr );
29    } /* end for */
30
31 } /* end function printCharacters */
```

`sPtr` is modified by function **`printCharacters`**

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



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_12.c

```
1  /* Fig. 7.12: fig07_12.c
2     Attempting to modify data through a
3     non-constant pointer to constant data. */
4  #include <stdio.h>
5  void f( const int *xPtr ); /* prototype */
6
7
8  int main( void )
9  {
10    int y;         /* define y */
11
12    f( &y );       /* f attempts illegal modification */
13
14    return 0;      /* indicates successful termination */
15
16 } /* end main */
17
18 /* xPtr cannot be used to modify the
19    value of the variable to which it points */
20 void f( const int *xPtr )
21 {
22    *xPtr = 100;   /* error: cannot modify a const object */
23 } /* end function f */
```

Pointer variable **xPtr** is modifiable, but the data to which it points, ***xPtr**, is not

***xPtr** has the **const** qualifier, so attempting to modify its value causes an error

```
Compiling...
FIG07_12.c
c:\books\2006\chtp5\examples\ch07\fig07_12.c(22) : error C2166: l-value
    specifies const object
Error executing cl.exe.

FIG07_12.exe - 1 error(s), 0 warning(s)
```



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_13.c

```
1  /* Fig. 7.13: fig07_13.c
2     Attempting to modify a constant pointer to non-constant data */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x; /* define x */
8     int y; /* define y */
9
10    /* ptr is a constant pointer to an integer that can be modified
11       through ptr, but ptr always points to the same memory location */
12    int * const ptr = &x;
13
14    *ptr = 7; /* allowed: *ptr is not const */
15    ptr = &y; /* error: ptr is const; cannot assign new address */
16
17    return 0; /* indicates successful termination */
18
19 } /* end main */
```

Pointer **`ptr`** is not modifiable, but the data to which it points, **`*ptr`**, can be changed

```
Compiling...
FIG07_13.c
c:\books\2006\chtp5\Examples\ch07\FIG07_13.c(15) : error C2166: l-value
    specifies const object
Error executing cl.exe.

FIG07_13.exe - 1 error(s), 0 warning(s)
```



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_14.c

```
1  /* Fig. 7.14: fig07_14.c
2     Attempting to modify a constant pointer to constant data. */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     int x = 5; /* initialize x */
8     int y;     /* define y */
9
10    /* ptr is a constant pointer to a constant integer. ptr always
11       points to the same location; the integer at that location
12       cannot be modified */
13    const int *const ptr = &x;
14
15    printf( "%d\n", *ptr );
16
17    *ptr = 7; /* error: *ptr is const; cannot assign new value */
18    ptr = &y; /* error: ptr is const; cannot assign new address */
19
20    return 0; /* indicates successful termination */
21
22 } /* end main */
```

Neither pointer **sPtr** nor the data to which it points, ***sPtr**, is modifiable

```
Compiling...
FIG07_14.c
c:\books\2006\chtp5\Examples\ch07\FIG07_14.c(17) : error C2166: l-value
   specifies const object
c:\books\2006\chtp5\Examples\ch07\FIG07_14.c(18) : error C2166: l-value
   specifies const object
Error executing cl.exe.

FIG07_12.exe - 2 error(s), 0 warning(s)
```



© 2007 Pearson Education, Inc. All rights reserved.

7.7 sizeof Operator - self-study

- sizeof
  - Returns size of operand in bytes
  - For arrays: size of 1 element * number of elements
  - if sizeof(int) equals 4 bytes, then

```
int myArray[ 10 ];
printf( "%d", sizeof( myArray ) );
```

    - will print 40
- sizeof can be used with
  - Variable names
  - Type name
  - Constant values



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_16.c

```
1  /* Fig. 7.16: fig07_16.c
2     Sizeof operator when used on an array name
3     returns the number of bytes in the array. */
4  #include <stdio.h>
5
6  size_t getSize( float *ptr ); /* prototype */
7
8  int main( void )
9  {
10    float array[ 20 ]; /* create array */
11
12    printf( "The number of bytes in the array is %d"
13            "\nThe number of bytes returned by getSize is %d\n",
14            sizeof( array ), getSize( array ) );
15
16    return 0; /* indicates successful termination */
17
18 } /* end main */
19
20 /* return size of ptr */
21 size_t getSize( float *ptr )
22 {
23    return sizeof( ptr );
24
25 } /* end function getSize */
```

`float`s take up 4 bytes in memory, so 20 **`float`**s take up 80 bytes

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_17.c

(1 of 2)

```
1  /* Fig. 7.17: fig07_17.c
2     Demonstrating the sizeof operator */
3  #include <stdio.h>
4
5  int main( void )
6  {
7     char c;
8     short s;
9     int i;
10    long l;
11    float f;
12    double d;
13    long double ld;
14    int array[ 20 ];  /* create array of 20 int elements */
15    int *ptr = array; /* create pointer to array */
16
```



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_17.c

(2 of 2)

```
17    printf( "      sizeof c = %d\tsizeof(char)  = %d"
18            "\n      sizeof s = %d\tsizeof(short) = %d"
19            "\n      sizeof i = %d\tsizeof(int) = %d"
20            "\n      sizeof l = %d\tsizeof(long) = %d"
21            "\n      sizeof f = %d\tsizeof(float) = %d"
22            "\n      sizeof d = %d\tsizeof(double) = %d"
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
31    return 0; /* indicates successful termination */
32
33 } /* 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
```



© 2007 Pearson Education, Inc. All rights reserved.

7.8 Pointer Expressions and Pointer Arithmetic

- **Arithmetic operations can be performed on pointers**
  - **Increment/decrement pointer (++ or --)**
  - **Add an integer to a pointer(+ or += , - or -=)**
  - **Pointers may be subtracted from each other**
  - **Operations meaningless unless performed on an array**



© 2007 Pearson Education, Inc. All rights reserved.

7.8 Pointer Expressions and Pointer Arithmetic

- 5 element int array on machine with 4 byte ints
  - vPtr points to first element v[0]
    - at location 3000 (vPtr = 3000)
  - vPtr += 2; sets vPtr to 3008
    - vPtr points to v[2] (incremented by 2), but the machine has 4 byte ints, so it points to address 3008



© 2007 Pearson Education, Inc. All rights reserved.

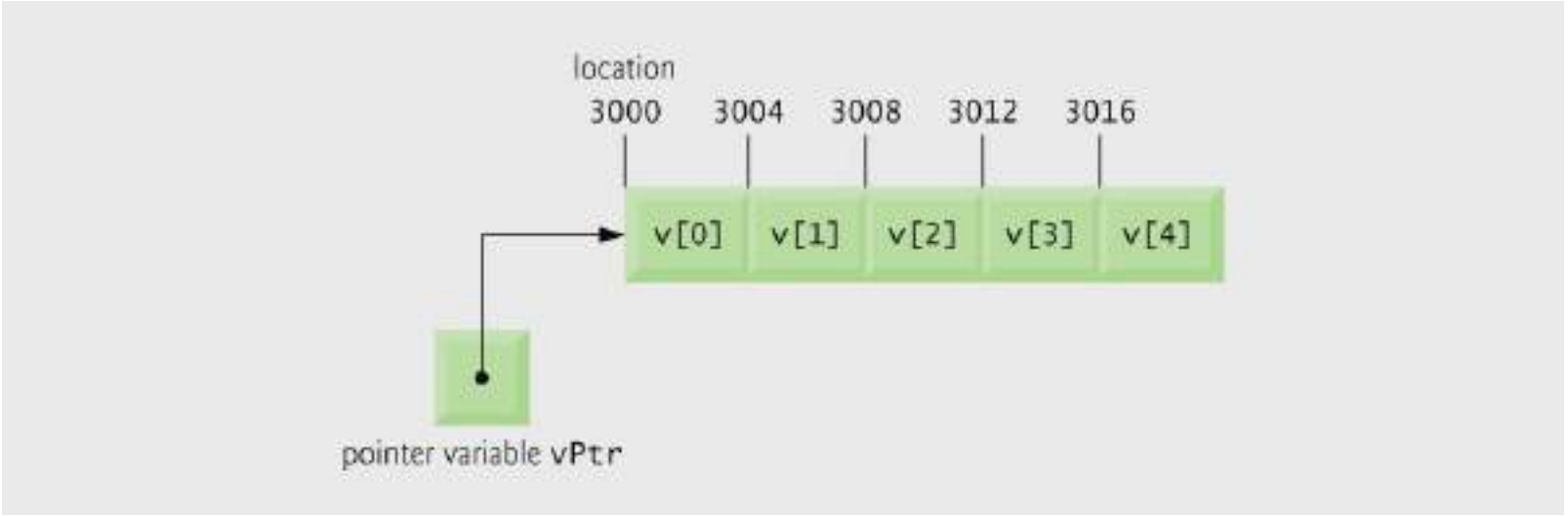


Fig. 7.18 | Array **v** and a pointer variable **vPtr** that points to **v**.

© 2007 Pearson Education, Inc. All rights reserved.

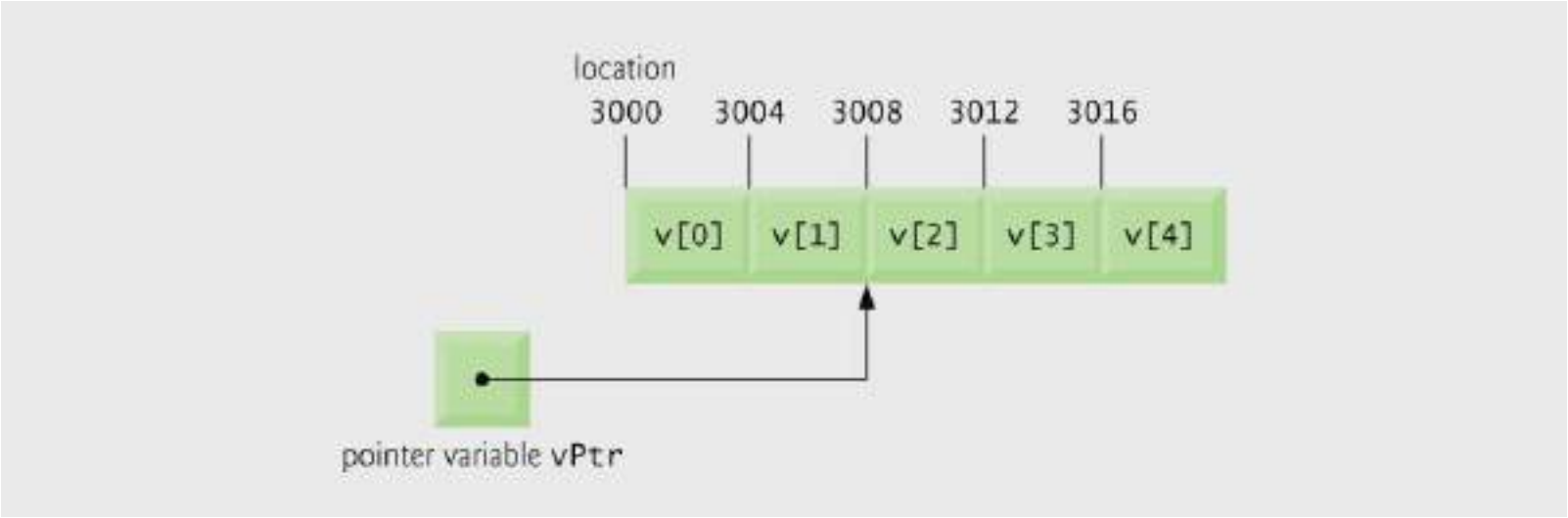


Fig. 7.19 | The pointer **vPtr** after pointer arithmetic.

© 2007 Pearson Education, Inc. All rights reserved.

7.8 Pointer Expressions and Pointer Arithmetic

- **Subtracting pointers**
  - **Returns number of elements from one to the other. If**

    vPtr2 = v[2];

    vPtr = v[0];
  - **vPtr2 - vPtr would produce 2**
- **Pointer comparison (<, == , >)**
  - **See which pointer points to the higher numbered array element**
  - **Also, see if a pointer points to 0**



© 2007 Pearson Education, Inc. All rights reserved.

7.8 Pointer Expressions and Pointer Arithmetic

- **Pointers of the same type can be assigned to each other**
  - **If not the same type, a cast operator must be used**
  - **Exception: pointer to void (type void *)**
    - **Generic pointer, represents any type**
    - **No casting needed to convert a pointer to void pointer**
    - **void pointers cannot be dereferenced**



© 2007 Pearson Education, Inc. All rights reserved.

7.9 Relationship Between Pointers and Arrays

- **Arrays and pointers closely related**
  - **Array name like a constant pointer**
  - **Pointers can do array subscripting operations**
- **Define an array b[5] and a pointer bPtr**
  - **To set them equal to one another use:**

```
bPtr = b;
```

    - **The array name (b) is actually the address of first element of the array b[5]**

```
bPtr = &b[ 0 ]
```

    - **Explicitly assigns bPtr to address of first element of b**



© 2007 Pearson Education, Inc. All rights reserved.

7.9 Relationship Between Pointers and Arrays

- **Element b[3]**
  - **Can be accessed by *(bPtr + 3)**

    Where 3 is called the offset. Called pointer/offset notation
  - **Can be accessed by bptr[3]**

    Called pointer/subscript notation

    bPtr[3] same as b[3]
  - **Can be accessed by performing pointer arithmetic on the array itself**

    ***(b + 3)**



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_20.c

(1 of 3)

```
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" );
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
```

Array subscript notation

Pointer/offset notation



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_20.c

(2 of 3)

```
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
48 } /* end main */
```

Pointer subscript notation

Pointer offset notation

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

(continued on next slide...)



© 2007 Pearson Education, Inc. All rights reserved.

(continued from previous slide...)

```
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
```

Outline

fig07_20.c

(3 of 3)



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_21.c

(1 of 2)

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

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

int main( void )
{
   char string1[ 10 ];           /* create array string1 */
   char *string2 = "Hello";      /* create a pointer to a string */
   char string3[ 10 ];           /* create array string3 */
   char string4[] = "Good Bye"; /* create a pointer to a string */

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

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

   return 0; /* indicates successful termination */

} /* end main */
```



© 2007 Pearson Education, Inc. All rights reserved.

Outline

fig07_21.c

(2 of 2)

```
25 /* copy s2 to s1 using array notation */
26 void copy1( char * const s1, const char * const s2 )
27 {
28    int i; /* counter */
29
30    /* loop through strings */
31    for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ ) {
32       ; /* do nothing in body */
33    } /* end for */
34
35 } /* end function copy1 */
36
37 /* copy s2 to s1 using pointer notation */
38 void copy2( char *s1, const char *s2 )
39 {
40    /* loop through strings */
41    for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ ) {
42       ; /* do nothing in body */
43    } /* end for */
44
45 } /* end function copy2 */
```

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

Condition of **`for`** loop actually performs an action



© 2007 Pearson Education, Inc. All rights reserved.

7.10 Arrays of Pointers

- Arrays can contain pointers
- For example: an array of strings

```
char *suit[ 4 ] = { "Hearts", "Diamonds",
   "Clubs", "Spades" };
```

  - Strings are pointers to the first character
  - char * – each element of suit is a pointer to a char
  - The strings are not actually stored in the array suit, only pointers to the strings are stored



© 2007 Pearson Education, Inc. All rights reserved.

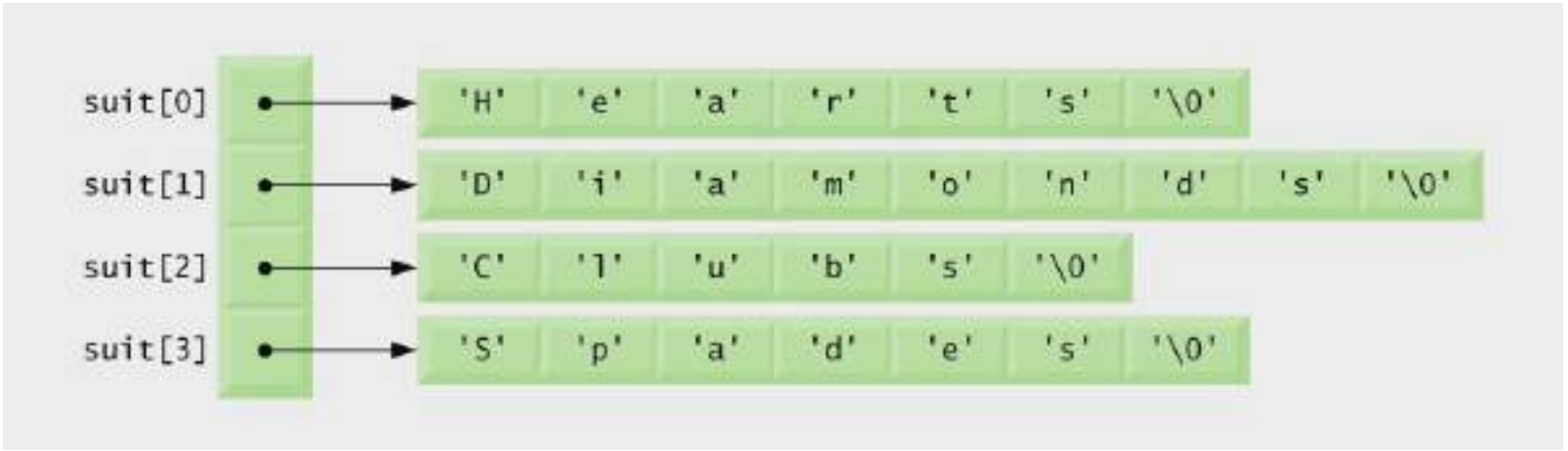


Fig. 7.22 | Graphical representation of the **suit** array.



© 2007 Pearson Education, Inc. All rights reserved.