MIS 102: Computer Programming

Unit 6 – C Pointers

Yihuang K. Kang, PhD

“Computational thinking is a fundamental skill for everyone, not just for computer scientists.”

– *Jeannette M. Wing* (周以真)

Introduction to C Pointer

- ***Pointer*** is one of the most powerful features of the C programming language. Pointer enables program to simulate ***pass-by-reference***, create/manipulate dynamic data structures, and manage memory usages.

- Normally, a variable directly contains a specific value. The pointers, however, are actually variables whose values are ***memory addresses***. In this sense, a variable name directly references a value, and a pointer indirectly references a value

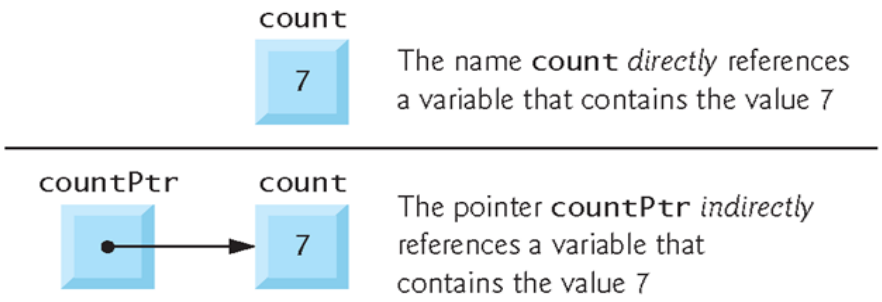


Fig. 7.1 | Directly and indirectly referencing a variable.

Pointer 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 *** 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**.

- Note that the asterisk notation(*) only applies to the following variable name. A good programming practice is that the '*', should be with the variable name rather than with the type, i.e. use

  char *s, *t, *u;

  instead of

  char* s, t, u;

  which is wrong, since **t** and **u** DO NOT get declared as pointers.

Pointer Definitions and Initialization(cont.)

- Similar to regular variable, pointer variable should be initialized before it is used so that we can prevent unexpected results due to memory relocation.

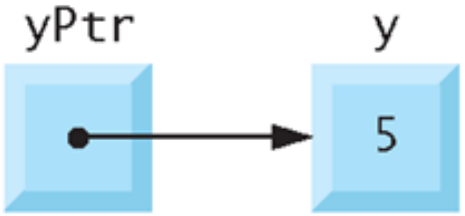
- A defensive programming style is to set unused pointers to **NULL** (or **0**, although not suggested), which is a symbolic constant defined in <*stddef.h*> that indicate *nothing*.

Pointer Definitions and Initialization(cont.)

- Recall that we have been using address operator (**&**) that returns the address of its operand, which can also be used to assign the address to a pointer. For example,

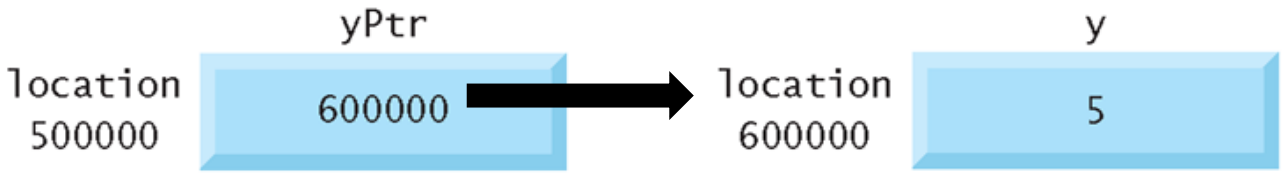
```
int y = 5; int *yPtr;
yPtr = &y;
```

assign the address of variable **y** to pointer variable **yPtr**.



Pointer Definitions and Initialization(cont.)

- Again, pointer is a variable that store the memory location (address). Here is an example.



- Another operator, also an asterisk notation(*) but called *dereferencing operator*, can actually return the value of the object to which its operand (a pointer) points. For example,

```
printf( "%d", *yPtr );
```

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

Pointer Definitions and Initialization(cont.)

```
1   // Fig. 7.4: fig07_04.c
2   // Using the & and * pointer 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; // set aPtr to the 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  } // end main
```

```
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
```

Passing Arguments to Functions

- There are two ways to pass arguments to a function—***pass-by-value*** and ***pass-by-reference***.

- All arguments in C are actually passed by value. However, we can use pointers and the dereferencing operator to simulate pass-by-reference.

- 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 large object by value (which incurs the time and memory overheads of making a copy of the object).

Passing Arguments to Functions(cont.)

- When calling a function with arguments that should be modified, the *addresses* of the arguments are passed. It can be accomplished by applying the address operator (**&**) to the variable (in the caller) whose value will be modified.

- We have discussed that 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 dereferencing operator (*) may be used in the function to modify the value at that location in the caller's memory.

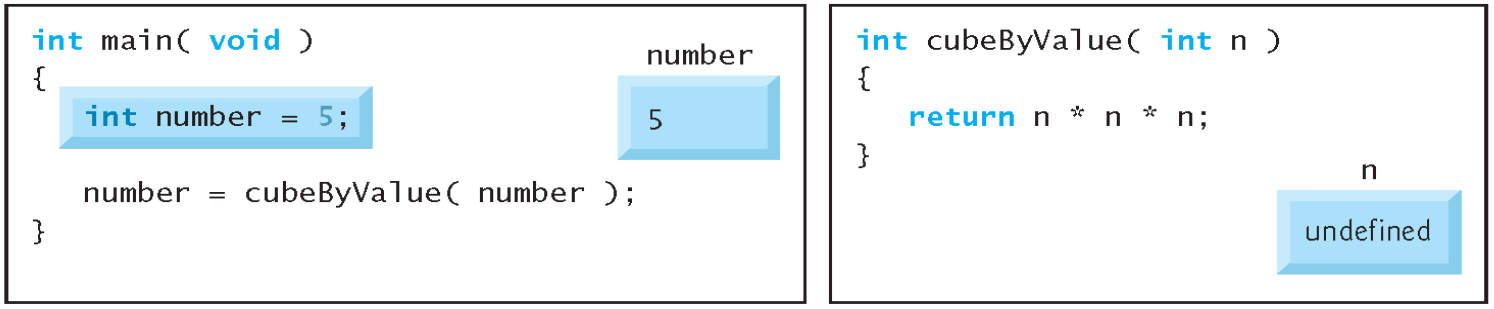
Passing Arguments to Functions(cont.)

```
1  // Fig. 7.6: fig07_06.c
2  // Cube a variable using pass-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 } // end main
18
19 // calculate and return cube of integer argument
20 int cubeByValue( int n )
21 {
22    return n * n * n; // cube local variable n and return result
23 } // end function cubeByValue
```

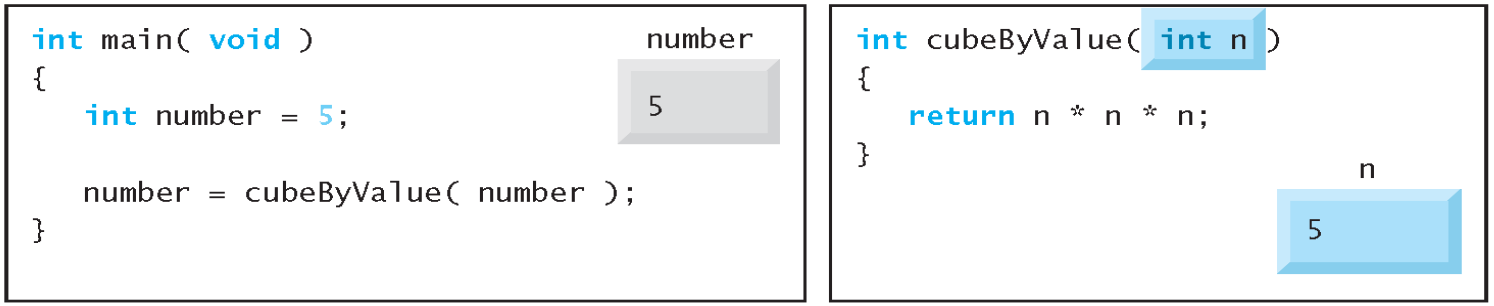
```
The original value of number is 5
The new value of number is 125
```

Passing Arguments to Functions(cont.)

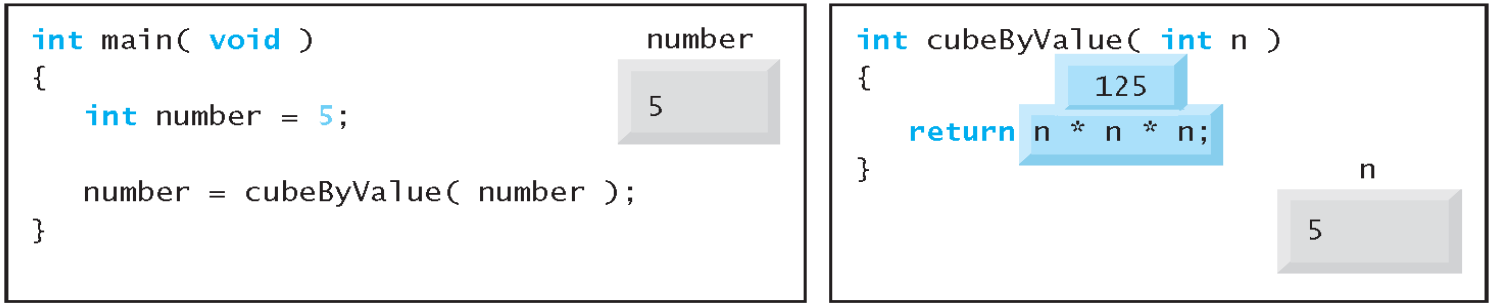
Step 1: Before `main` calls `cubeByValue`:



Step 2: After `cubeByValue` receives the call:

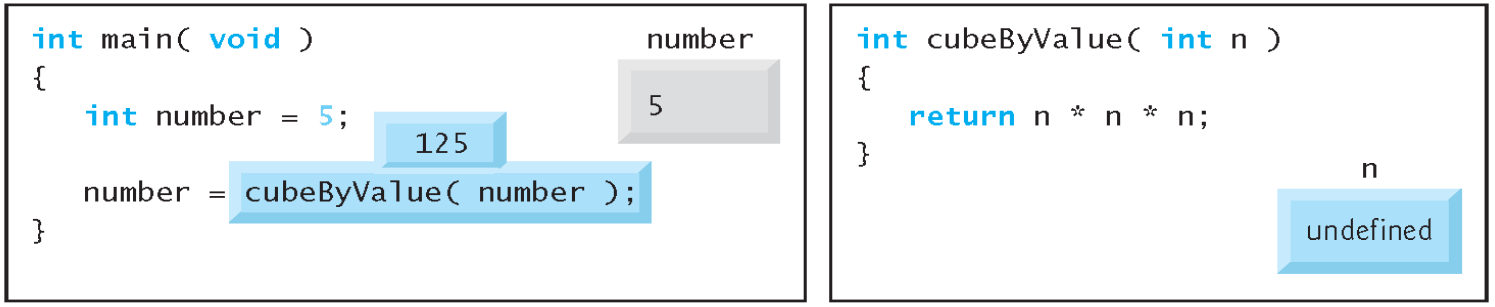


Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:

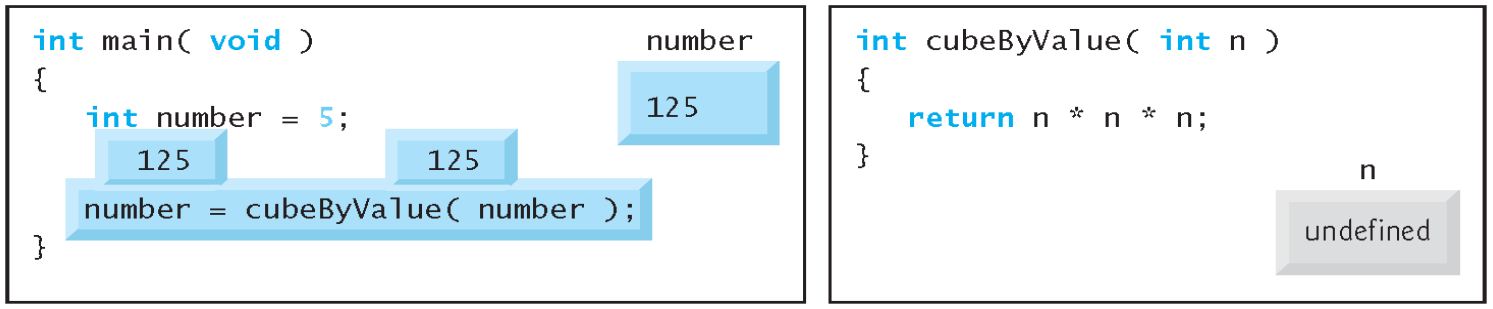


Passing Arguments to Functions(cont.)

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



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



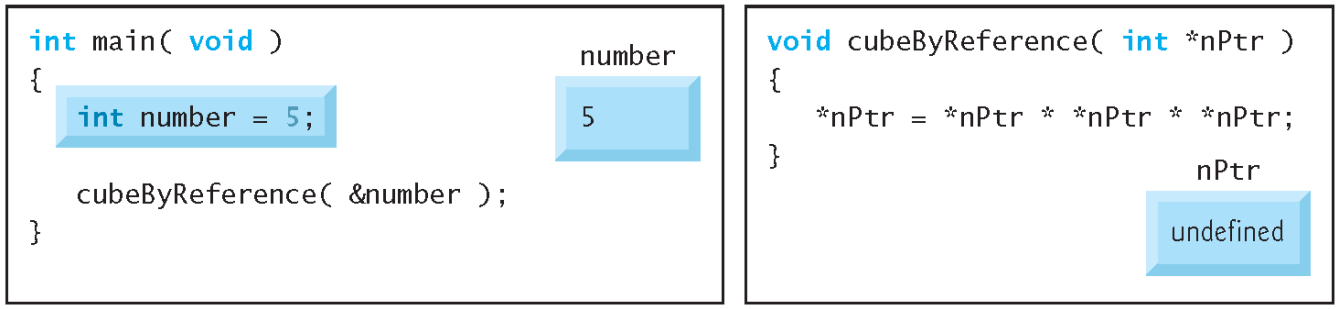
Passing Arguments to Functions(cont.)

```
1  // Fig. 7.7: fig07_07.c
2  // Cube a variable using pass-by-reference with a pointer argument.
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); // function 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 } // end main
20 // calculate cube of *nPtr; actually modifies number in main
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
24 } // end function cubeByReference
```

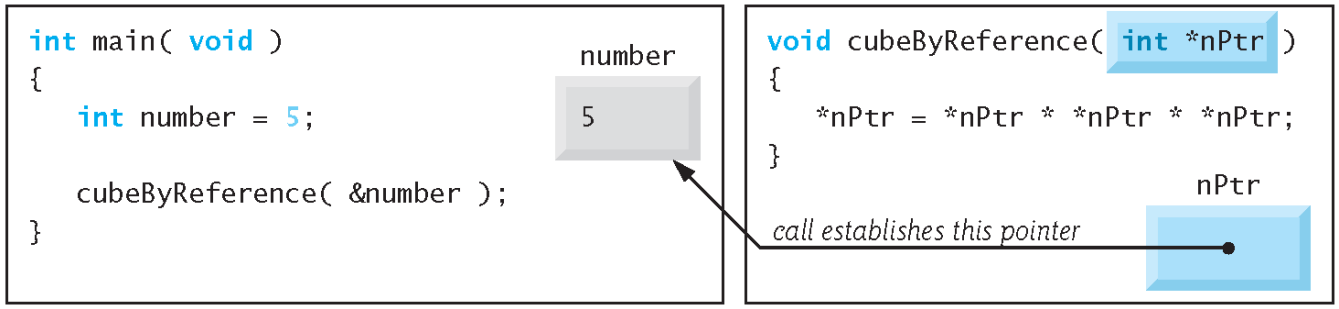
```
The original value of number is 5
The new value of number is 125
```

Passing Arguments to Functions(cont.)

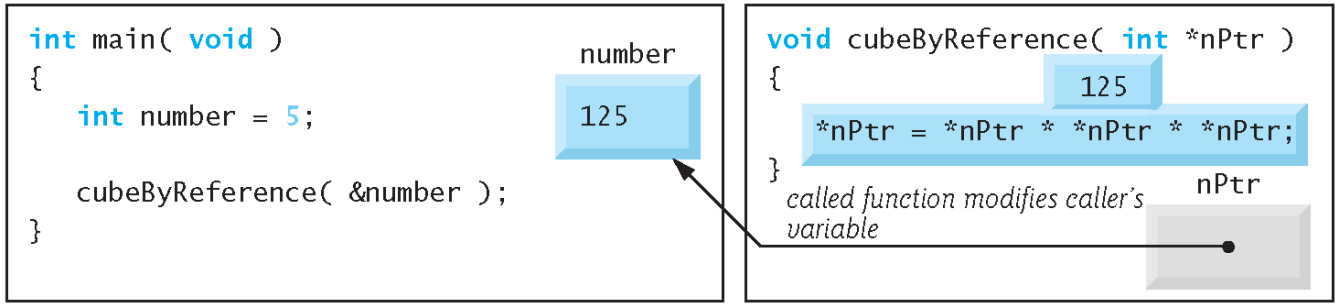
Step 1: Before `main` calls `cubeByReference`:



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`:



The ***const*** Qualifier with Pointers

- The ***const*** qualifier enables you to inform the compiler that the value of a particular variable should not be modified, which also makes your program easier to maintain.

- If an attempt is made to modify a value that's declared ***const***, the compiler catches it and issues either a warning or an error, depending on the particular compiler.

- At runtime, a memory-access violation (segmentation faults) message is generated, if our programs try to dereference values as pointers.

The *const* Qualifier with Access Privileges

		Data (Variable value)	
		Non-Constant	**Constant**
Pointer	**Non-Constant**	Data is **modifiable**. Address is **modifiable**. e.g. int *ptr;	Data is **NOT modifiable**. Address is **modifiable**. e.g. const int *ptr;
	Constant	Data is **modifiable**. Address is **NOT modifiable**. e.g. int *const ptr;	Data is **NOT modifiable**. Address is **NOT modifiable**. e.g. const int *const ptr;

Non-Constant Pointer and Non-Constant Data

- The data can be modified through the dereferenced pointer, and the pointer can be modified to point to other data items.

- A declaration for a non-constant pointer to non-constant data does NOT include ***const***.

- Such a pointer might be used to receive a string as an argument to a function that processes (and possibly modifies) each character in the string.

Non-Constant Pointer and Non-Constant Data(cont.)

```c
// Fig. 7.10: fig07_10.c
// Converting a string to uppercase using a
// non-constant pointer to non-constant data.
#include <stdio.h>
#include <ctype.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", string );
   convertToUppercase( string );
   printf( "\nThe string after conversion is: %s\n", string );
} // end main

// convert string to uppercase letters
void convertToUppercase( char *sPtr )
{
   while ( *sPtr != '\0' ) { // current character is not '\0'
      *sPtr = toupper( *sPtr ); // convert to uppercase
      ++sPtr; // make sPtr point to the next character
   } // end while
} // end function convertToUppercase
```

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

Non-Constant Pointer and Constant Data

- A non-constant pointer to constant data can be modified to point to any data item of the appropriate type, but the data to which it points CANNOT be modified. For example, it will show an error saying “assignment of read-only location” when we try to compile & run the following code:

```
int a = 10;
const int *ptr = &a;
*ptr = 100; // try to assign a value to a read-only location
```

- Such a pointer might be used to receive an array argument to a function that will process each element *without modifying the data*.

Non-Constant Pointer and **Constant** Data(cont.)

```
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     puts( "The string is:" );
15     printCharacters( string );
16     puts( "" );
17  } // end main
18
19  // sPtr cannot modify the character to which it points,
20  // i.e., sPtr is a "read-only" pointer
21  void printCharacters( const char *sPtr )
22  {
23     // loop through entire string
24     for ( ; *sPtr != '\0'; ++sPtr ) { // no initialization
25        printf( "%c", *sPtr );
26     } // end for
27  } // end function printCharacters
```

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

Non-Constant Pointer and Constant Data(cont.)

```
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  int main( void )
8  {
9     int y; // define y
10
11    f( &y ); // f attempts illegal modification
12 } // end main
13
14 // xPtr cannot be used to modify the
15 // value of the variable to which it points
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // error: cannot modify a const object
19 } // end function f
```

```
c:\examples\ch07\fig07_12.c(18) : error C2166: l-value specifies const object
```

Fig. 7.12 | Attempting to modify data through a non-constant pointer to constant data.

Non-Constant Pointer and Constant Data(cont.)

- When data must be passed to a function, we can use non-constant pointers to constant data to get the performance of pass-by-reference and the protection of pass-by-value.

- If memory is low and execution efficiency is a concern, use pointers. If memory is in abundance and efficiency is not a major concern, pass data by value to enforce the principle of least privilege.

- Note that some compilers do not fully support ***const*** well, so pass-by-value is still the best way to prevent data from being modified.

Constant Pointer and Non-**Constant** Data

- A constant pointer to non-constant data *always points to the same memory location*, and the data at that location can be modified through the pointer. This is the default for *an array name*, which is a constant pointer to the beginning of the array.

- A constant pointer to non-constant data can be used to receive an array as an argument to a function that accesses array elements using only array subscript notation.

- Also note that pointers declared ***const*** must be initialized when they're defined (if the pointer is a function parameter, it's initialized with a pointer that's passed to the function).

Constant Pointer and Non-Constant Data(cont.)

```
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 } // end main
```

```
c:\examples\ch07\fig07_13.c(15) : error C2166: l-value specifies const object
```

Constant Pointer and **Constant** Data

- The least access privilege is granted by a constant pointer to constant data—a pointer *always points to the same memory location*, and the *data at that memory location cannot be modified*. Here is an example,

```
const int *const ptr;
```

where **ptr** is an constant pointer to an constant **int** data.

- Such type of pointer is usually used when we only looks at the array using array subscript notation and does not modify the array.

Constant Pointer and **Constant** Data(cont.)

```
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; // initialization is OK
14
15    printf( "%d\n", *ptr );
16    *ptr = 7; // error: *ptr is const; cannot assign new value
17    ptr = &y; // error: ptr is const; cannot assign new address
18 } // end main
```

```
c:\examples\ch07\fig07_14.c(16) : error C2166: l-value specifies const object
c:\examples\ch07\fig07_14.c(17) : error C2166: l-value specifies const object
```

Fig. 7.14 | Attempting to modify a constant pointer to constant data.

Bubble Sort Using Pass-by-Reference

- We can now improve the Bubble Sort program (Fig. 6.15) we discussed previously by rewriting it. Let's create 2 functions, ***bubblesort***() and ***swap***() that sort an integer array and exchange the array elements.

- 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 only 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.

Bubble Sort Using Pass-by-Reference(cont.)

```
35  // sort an array of integers using bubble sort algorithm
36  void bubbleSort( int * const array, size_t size )
37  {
38     void swap( int *element1Ptr, int *element2Ptr ); // prototype
39     unsigned int pass; // pass counter
40     size_t j; // comparison counter
41
42     // loop to control passes
43     for ( pass = 0; pass < size - 1; ++pass ) {
44
45        // loop to control comparisons during each pass
46        for ( j = 0; j < size - 1; ++j ) {
47
48           // swap adjacent elements if they're out of order
49           if ( array[ j ] > array[ j + 1 ] ) {
50              swap( &array[ j ], &array[ j + 1 ] );
51           } // end if
52        } // end inner for
53     } // end outer for
54  } // end function bubbleSort
55
56  // swap values at memory locations to which element1Ptr and
57  // element2Ptr point
58  void swap( int *element1Ptr, int *element2Ptr )
59  {
60     int hold = *element1Ptr;
61     *element1Ptr = *element2Ptr;
62     *element2Ptr = hold;
63  } // end function swap
```

Bubble Sort Using Pass-by-Reference(cont.)

```
1  // Fig. 7.15: fig07_15.c
2  // Putting values into an array, sorting the values into
3  // ascending order and printing the resulting array.
4  #include <stdio.h>
5  #define SIZE 10
6
7  void bubbleSort( int * const array, size_t size ); // prototype
8
9  int main( void )
10 {
11    // initialize array a
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    size_t i; // counter
15
16    puts( "Data items in original order" );
17
18    // loop through array a
19    for ( i = 0; i < SIZE; ++i ) {
20       printf( "%4d", a[ i ] );
21    } // end for
22
23    bubbleSort( a, SIZE ); // sort the array
24
25     puts( "\nData items in ascending order" );
26
27     // loop through array a
28     for ( i = 0; i < SIZE; ++i ) {
29        printf( "%4d", a[ i ] );
30     } // end for
31
32     puts( "" );
33 } // end main
34
```

sizeof Operator and *sizeof*() Revisit

- C provides the special unary operator ***sizeof*** and function ***sizeof***() to determine the size in bytes of an array (or any other data type) . 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 and ***sizeof***() Revisit(cont.)

- Note that when you use ***sizeof*** with a pointer, it returns the size of the pointer, not the size of the item to which it points.

- Also note that to determine the number of array elements using ***sizeof*** works only when using the actual array, not when using a pointer to the array—an *array decaying* (into pointers) problem in C.

- We will soon discuss how to dynamically identify array size using aggregate data type ***structure*** later.

sizeof Operator and *sizeof*() Revisit(cont.)

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

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

sizeof Operator and *sizeof*() Revisit(cont.)

```
1   // Fig. 7.17: fig07_17.c
2   // Using operator sizeof to determine standard data type sizes.
3   #include <stdio.h>
4
5   int main( void )
6   {
7      char c;
8      short s;
9      int i;
10     long l;
11     long long ll;
12     float f;
13     double d;
14     long double ld;
15     int array[ 20 ]; // create array of 20 int elements
16     int *ptr = array; // create pointer to array
17
18     printf( "     sizeof c = %u\tsizeof(char)  = %u"
19             "\n     sizeof s = %u\tsizeof(short) = %u"
20             "\n     sizeof i = %u\tsizeof(int) = %u"
21             "\n     sizeof l = %u\tsizeof(long) = %u"
22             "\n    sizeof ll = %u\tsizeof(long long) = %u"
23             "\n     sizeof f = %u\tsizeof(float) = %u"
```

sizeof Operator and *sizeof*() Revisit(cont.)

```
24              "\n     sizeof d = %u\tsizeof(double) = %u"
25              "\n    sizeof ld = %u\tsizeof(long double) = %u"
26              "\n sizeof array = %u"
27              "\n   sizeof ptr = %u\n",
28             sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
29             sizeof( int ), sizeof l, sizeof( long ), sizeof ll,
30             sizeof( long long ), sizeof f, sizeof( float ), sizeof d,
31             sizeof( double ), sizeof ld, sizeof( long double ),
32             sizeof array, sizeof ptr );
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 ll = 8       sizeof(long long) = 8
     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
```

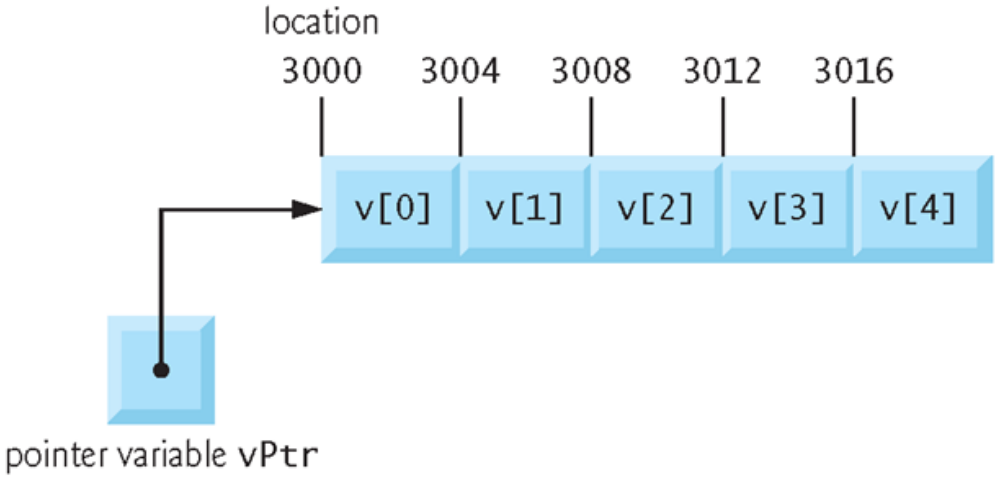
Pointer Expressions and Arithmetic

- 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.

- A limited set of arithmetic operations may be performed on pointers. For example, 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—this last operation is meaningful only when both pointers point to elements of the same array.

Pointer Expressions and Arithmetic(cont.)

- Assume that array **int v[5]** has been defined and its first element is at location 3000 in memory. And a pointer **vPtr** has been initialized to point to **v[0]**—i.e., the value of vPtr is 3000. Below illustrates this situation for a machine with *4-byte integers*.



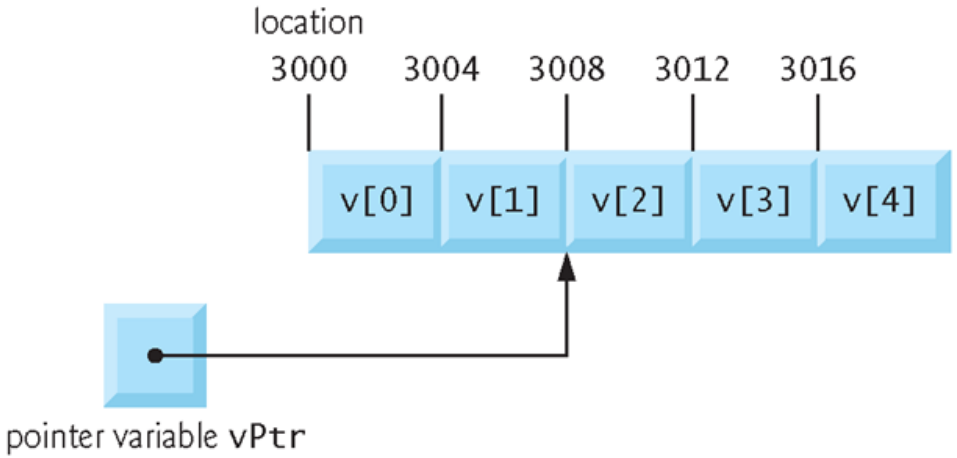
- Note that the results of pointer arithmetic depend on the size of the objects a pointer points to, pointer arithmetic is *machine dependent*.

Pointer Expressions and Arithmetic(cont.)

- In 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. 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 Arithmetic(cont.)

- If **vPtr** had been incremented to 3016, which points to v[4], the statement

vPtr -= 4;

would set vPtr back to 3000. And the statements, for example,

++vPtr;
- -vPtr;

increments/decrements the pointer to point to the *next/previous* location in the 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 Arithmetic(cont.)

- Another pointer arithmetic is the differences between two pointers. Let's say **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).

- Note that using pointer arithmetic on a pointer and comparing two pointers that do not refer to elements in the same array would generate errors.

Pointer to *void*

- A pointer should be assigned to another pointer if both have the same data type, but one exception is the *pointer to void* (i.e., **void ***).

- 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.

- Pointer to void is often used when the exact data type is unknown. All pointer types can be assigned a pointer to void, and vice versa. Also note that a pointer to void CANNOT be dereferenced without *type casting.*

Pointer to *void*(cont.)

- Consider the following example:

```
int main ()
{
   int a = 10;  char b[] = "This is a string.";

   void *aPtr, *bPtr;
   aPtr = &a; bPtr = &b;

   // Type casting
   printf("*aPtr = %d\n", * ((int *)aPtr) ); // “10”
   printf("*bPtr = %s\n", (char *)bPtr ); // “This is a string.”
   printf("b[0] = %c\n", ((char *)bPtr)[0] ); // “T”
   printf("*aPtr = %d\n", *aPtr ); //Syntax errors!
   printf("*bPtr = %s\n", *bPtr ); //Syntax errors!
}
```

Relationship between Pointers and Arrays

- As discussed previously, arrays and pointers are related in C and often used interchangeably. An array name can be thought of as *a constant pointer*. Pointers to arrays can be used to do any operation involving array subscripting.

- Let's say an array **b[5]** and a pointer variable **bPtr** have been defined. The array name **b** 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, as:

```
bPtr = b; // equivalent to bPtr = &b[ 0 ];
```

- Array element **b[3]** can then alternatively be referenced with the pointer expression

```
*( bPtr + 3 )
```

The **3** in the expression is the *offset* to the pointer.

Relationship between Pointers and Arrays(cont.)

- The array itself can be treated as a pointer and used in pointer arithmetic. For example, the expression

  ***(b + 3)**

  also refers to the array element **b[3]**.
- Pointers can be subscripted like arrays. For example,

  bPtr[1]

  refers to the array element **b[1]**.
- Also note that the statement

  bPtr += 3;

  works. But,

  b += 3;

  is *invalid* because it attempts to modify the value of the array name with pointer arithmetic. Again, array name is essentially a *constant pointer*.

Relationship between Pointers and Arrays(cont.)

```
1  // Fig. 7.21: fig07_21.c
2  // Copying a string using array notation and pointer notation.
3  #include <stdio.h>
4  #define SIZE 10
5
6  void copy1( char * const s1, const char * const s2 ); // prototype
7  void copy2( char *s1, const char *s2 ); // prototype
8
9  int main( void )
10 {
11    char string1[ SIZE ]; // create array string1
12    char *string2 = "Hello"; // create a pointer to a string
13    char string3[ SIZE ]; // create array string3
14    char string4[] = "Good Bye"; // create a pointer to a string
15
16    copy1( string1, string2 );
17    printf( "string1 = %s\n", string1 );
18
19    copy2( string3, string4 );
20    printf( "string3 = %s\n", string3 );
21 } // end main
22
```

Fig. 7.21 | Copying a string using array notation and pointer notation. (Part 1 of 2.)

Relationship between Pointers and Arrays(cont.)

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

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

Arrays of Pointers

- A array may contain pointers—a set of pointers that point to different objects/addresses but with the same data type. A common use of an array of pointers is to form an array of strings. Here is an example:

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

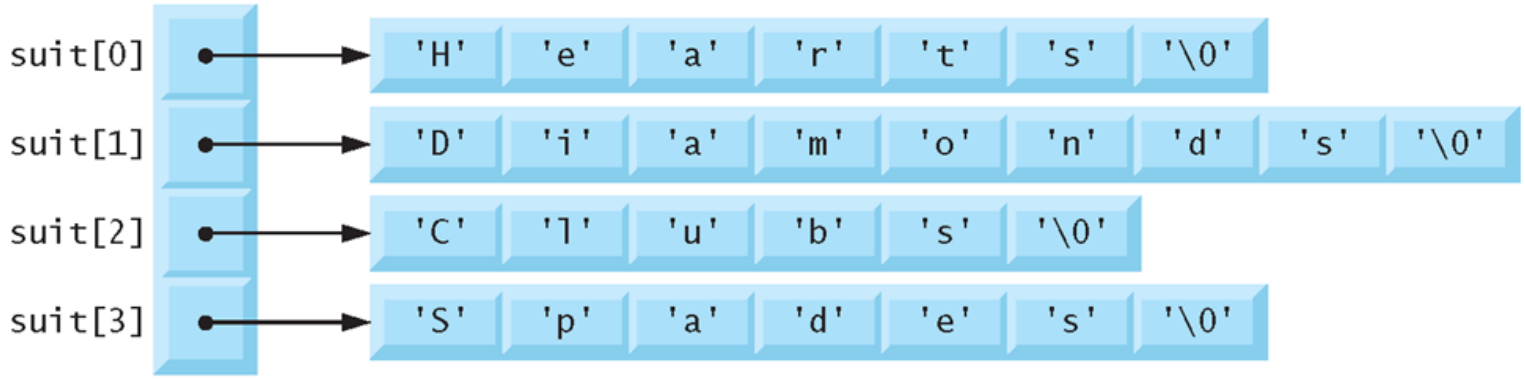


Fig. 7.22 | Graphical representation of the `suit` array.

Arrays of Pointers(cont.)

```
int main ()
{
   const char *suit[ 4 ] = { "Hearts", "Diamonds",
                             "Clubs", "Spades" };

   int a = 10, b = 20;
   int *ptr[20] = {&a, &b};

   printf("*ptr[0] = %d\n", *ptr[0]); // *ptr[0] = 10
   printf("*ptr[1] = %d\n", *ptr[1]); // *ptr[1] = 20
   printf("suit[1] = %s\n", suit[1]);  // suit[1] = Diamonds
}
```