02_07 C Pointers

Objectives

In this chapter, you'll:

- Use pointers and pointer operators.
- Pass arguments to functions by reference using pointers.
- Understand the various placements of the `const` qualifier and how they affect what you can do with a variable.
- Use the `sizeof` operator with variables and types.
- Use pointer arithmetic to process the elements in arrays.
- Understand the close relationships among pointers, arrays and strings.
- Define and use arrays of strings.
- Use pointers to functions.
- Learn about secure C programming issues with regard to pointers.

Why?

- C was developed when computers were much less powerful than they are today and being very efficient with speed and memory usage was often not just desirable but vital.

- The raw ability to work with particular memory locations was obviously a useful option to have. A few tasks these days, such as programming microcontrollers, still need this.

- However most modern programmers do not need such fine control and the complications of using pointers make programs less clear to understand and add to the ways in which they can be go wrong.

- So why are pointers still used so much in C & its successor, C++?

Why?

- C has pointers because it was designed in the 60's. At that time computers had very little memory and pointers allowed the implementation of Strings, arrays and parameter passing.

- They can also be used to optimize a program **to run faster** or **use less memory** that it would otherwise.

Why?

- Pointers are used (in the C language) in three different ways:
  1. simulate pass-by-reference,
  2. pass functions between functions
  3. 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.

Why?

- One of the complications when reading C programs is that a pointer could be being used for any, several or all of these different reasons with little or no distinction in the language so, unless the programmer has put in helpful comments, one has to follow through the program to see what each pointer is used for in order to work out why it is there instead of a plain simple variable.

So why use pointers? Why don't we use arrays to create data structures?

- The answer is simple. With an array you have to declare its maximum size (for every dimension) at the beginning. Let's say you create an array that can hold a maximum of twenty megabytes. When the array is declared, the twenty megabytes is claimed. Now this time you have only data for ten megabytes.

- (A next time it could be fifteen megabytes or five megabytes). So in this case ten megabytes of memory is wasted, because only ten megabytes from the twenty is used.

- This is where pointers come in. **With pointers, you can create dynamic data structures**. Instead of claiming the memory up-front, the memory is allocated (from the heap) while the program is running. So the exact amount of memory is claimed and there is no waste. Even better, memory not used will be returned to the heap. (Freed memory can be used for other programs).

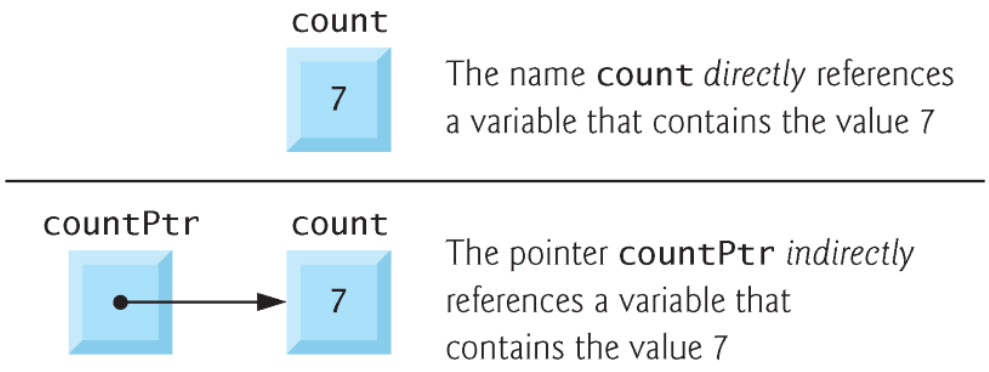


Fig. 7.1 | Directly and indirectly referencing a variable.

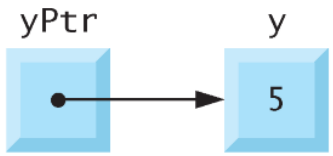


Fig. 7.2 | Graphical representation of a pointer pointing to an integer variable in memory.

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

```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);
}
```

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

```
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
```

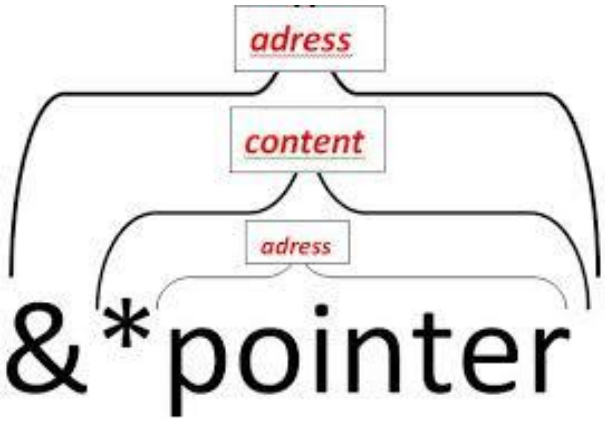
Fig. 7.4 | Using the & and * pointer operators. (Part 2 of 2.)

Pointer Variable Definitions and Initialization

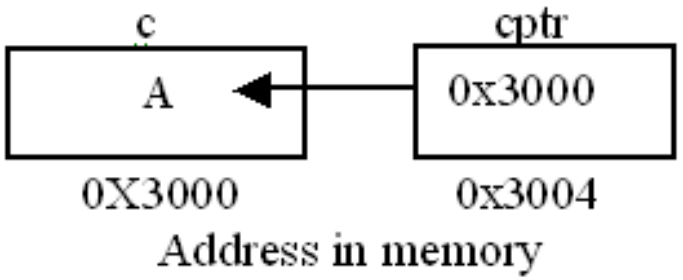
Initializing and Assigning Values to Pointers

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

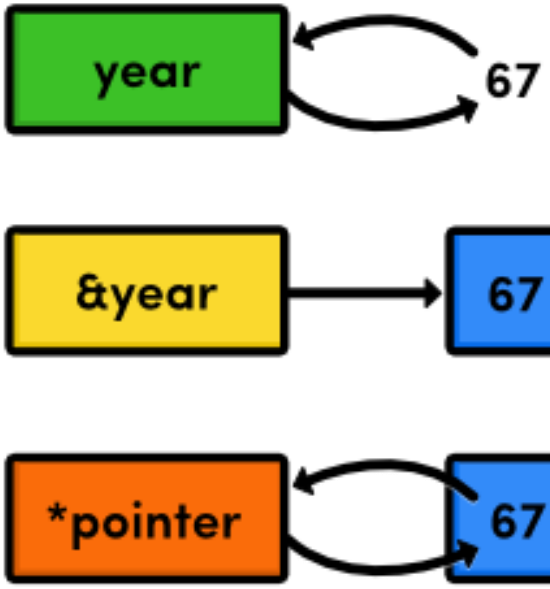
What Is a Pointer?




```
char c = 'A';
char *cptr;
cptr=&c;
```



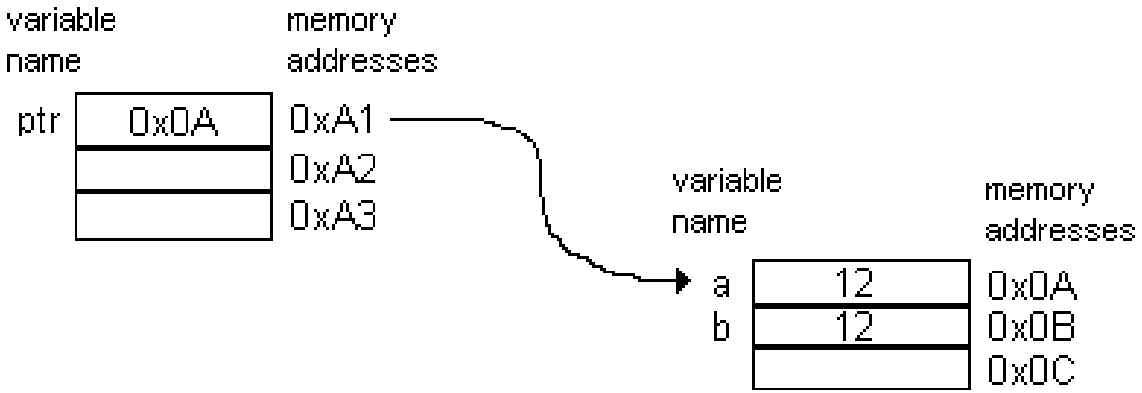
What Is a Pointer?



A variable transparently stores a value with no notion of memory addresses.

The reference operator returns the memory address of a variable.

The dereference operator accesses the value stored in a memory address.




```
int main {
    int *ptr;     // pointer to an integer variable
    int a, b;     // integer variables

    a = 12;       // assign value to a
    ptr = &a;     // assign address of a (0x0A in the example) to ptr
    b = *ptr;     // assign value at the address contained in ptr to b
}
```

The result is that a and b contain the same value (12).
And ptr still contains the address of a.

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

Fig. 7.5 | Precedence and associativity of the operators discussed so far.

The Dereference Operator (*)

- Don't confuse the dereference operator with the multiplication operator, although they share the same symbol, *.
- The dereference operator is a unary operator, which takes only one operand. The operand contains the address (that is, left value) of a variable.
- On the other hand, the multiplication operator is a binary operator that requires two operands to perform the operation of multiplication.

Does the size of pointers vary in C?

- So a pointer to a float, a char or an int are all the same size.
- Pointer is a memory address - and hence should be the same on a specific machine.
  - 32 bit machine => 4Bytes,
  - 64 bit => 8 Bytes.

```
sizeof(int*): 4
sizeof(float*): 4
sizeof(void*): 4
```

```
sizeof(int*): 8
sizeof(float*): 8
sizeof(void*): 8
```

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    printf("sizeof(int*): %i\n", sizeof(int*));
    printf("sizeof(float*): %i\n", sizeof(float*));
    printf("sizeof(void*): %i\n", sizeof(void*));
    return 0;
}
```

```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
}
```

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

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);
}
```

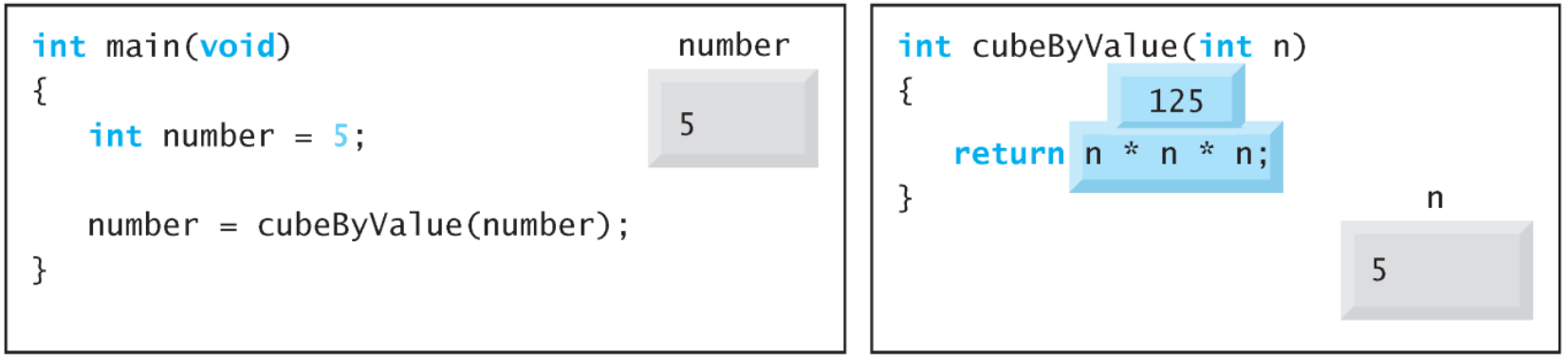
number: 5

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

n: 5

Fig. 7.8 | Analysis of a typical pass-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`:

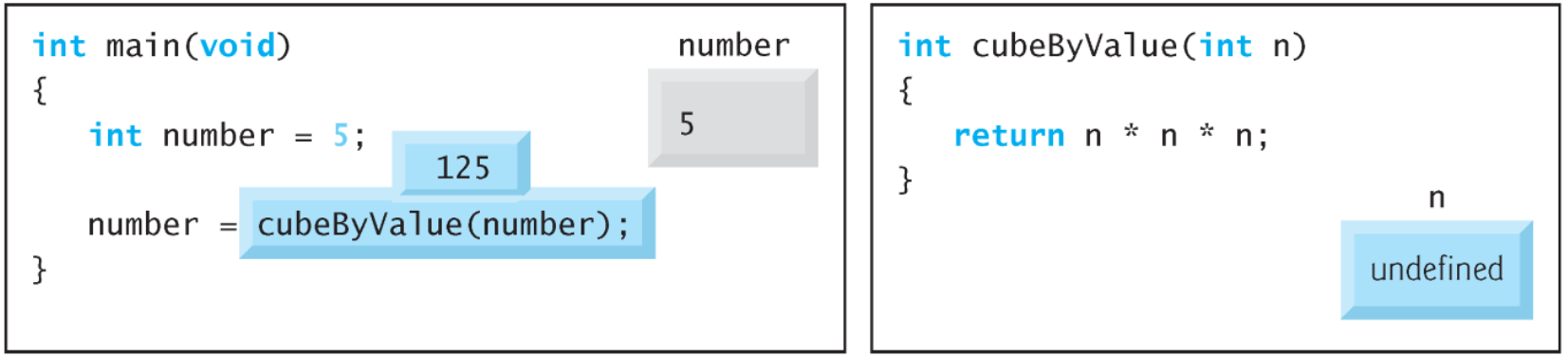


Fig. 7.8 | Analysis of a typical pass-by-value. (Part 2 of 3.)

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

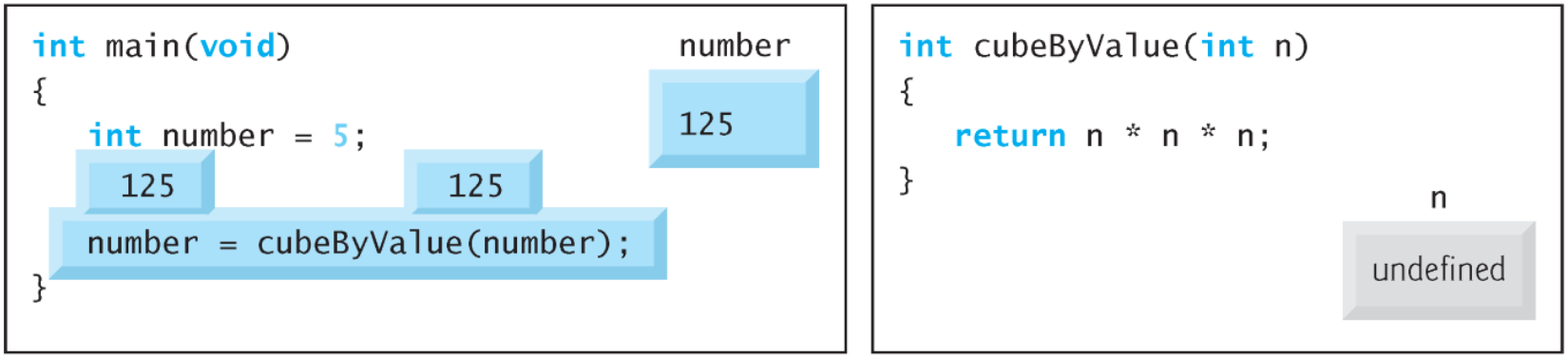


Fig. 7.8 | Analysis of a typical pass-by-value. (Part 3 of 3.)

```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
}
```

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

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

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

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 pass-by-reference with a pointer argument. (Part 1 of 2.)

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

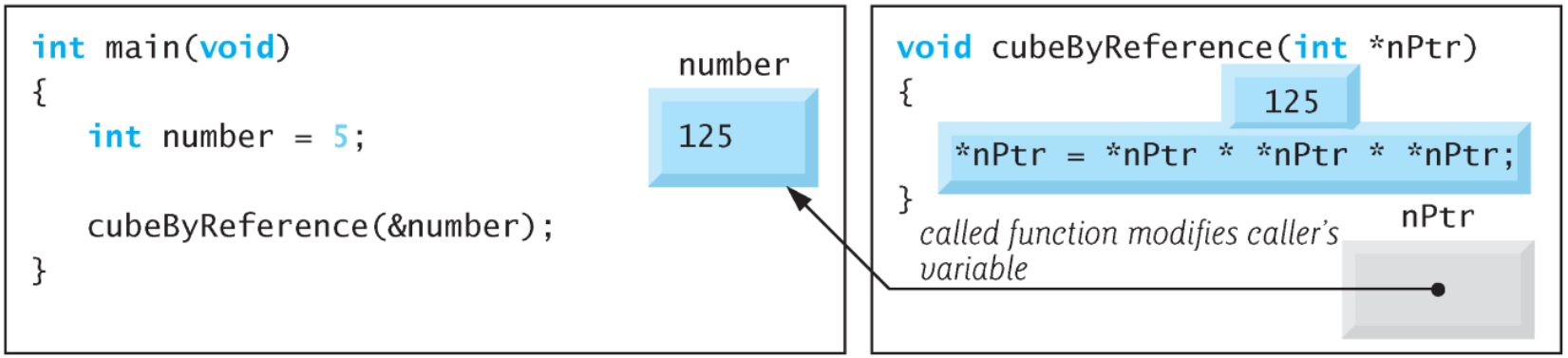


Fig. 7.9 | Analysis of a typical pass-by-reference with a pointer argument. (Part 2 of 2.)

7.5 Using the const Qualifier with Pointers (Cont.)

- There are four ways to pass a pointer to a function:
  - a non-constant pointer to non-constant data,
  - a constant pointer to nonconstant data,
  - a non-constant pointer to constant data, and
  - a constant pointer to constant data.

- Each of the four combinations provides different access privileges.

- These are discussed in the next several examples.

- Let the principle of least privilege be your guide.

```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);
}
```

Fig. 7.10 | Converting a string to uppercase using a non-constant pointer to non-constant data. (Part 1 of 2.)

```
18  // convert string to uppercase letters
19  void convertToUppercase(char *sPtr)
20  {
21     while (*sPtr != '\0') { // current character is not '\0'
22        *sPtr = toupper(*sPtr); // convert to uppercase
23        ++sPtr; // make sPtr point to the next character
24     }
25  }
```

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

Fig. 7.10 | Converting a string to uppercase using a non-constant pointer to non-constant data. (Part 2 of 2.)

```c
// Fig. 7.11: fig07_11.c
// Printing a string one character at a time using
// a non-constant pointer to constant data.

#include <stdio.h>

void printCharacters(const char *sPtr);

int main(void)
{
   // initialize char array
   char string[] = "print characters of a string";

   puts("The string is:");
   printCharacters(string);
   puts("");
}

```

Fig. 7.11 | Printing a string one character at a time using a non-constant pointer to constant data. (Part 1 of 2.)

```
19  // sPtr cannot be used to 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     }
27  }
```

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

Fig. 7.11 | Printing a string one character at a time using a non-constant pointer to constant data. (Part 2 of 2.)

```c
// Fig. 7.12: fig07_12.c
// Attempting to modify data through a
// non-constant pointer to constant data.
#include <stdio.h>
void f(const int *xPtr); // prototype

int main(void)
{
   int y; // define y

   f(&y); // f attempts illegal modification
}

// xPtr cannot be used to modify the
// value of the variable to which it points
void f(const int *xPtr)
{
   *xPtr = 100; // error: cannot modify a const object
}
```

```
error C2166: l-value specifies const object
```

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

```c
// Fig. 7.13: fig07_13.c
// Attempting to modify a constant pointer to non-constant data.
#include <stdio.h>

int main(void)
{
   int x; // define x
   int y; // define y

   // ptr is a constant pointer to an integer that can be modified
   // through ptr, but ptr always points to the same memory location
   int * const ptr = &x;

   *ptr = 7; // allowed: *ptr is not const
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

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

Fig. 7.13 | Attempting to modify a constant pointer to non-constant data.

```c
// Fig. 7.14: fig07_14.c
// Attempting to modify a constant pointer to constant data.
#include <stdio.h>

int main(void)
{
   int x = 5; // initialize x
   int y; // define y

   // ptr is a constant pointer to a constant integer. ptr always
   // points to the same location; the integer at that location
   // cannot be modified
   const int *const ptr = &x; // initialization is OK

   printf("%d\n", *ptr);
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ptr = &y; // error: ptr is const; cannot assign new address
}
```

```
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.

```c
// Fig. 7.16: fig07_16.c
// Applying sizeof to an array name returns
// the number of bytes in the array.
#include <stdio.h>
#define SIZE 20

size_t getSize(float *ptr); // prototype

int main(void)
{
   float array[SIZE]; // create array

   printf("The number of bytes in the array is %u"
          "\nThe number of bytes returned by getSize is %u\n",
          sizeof(array), getSize(array));
}

// return size of ptr
size_t getSize(float *ptr)
{
   return sizeof(ptr);
}
```

Fig. 7.16 | Applying `sizeof` to an array name returns the number of bytes in the array. (Part 1 of 2.)

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

Fig. 7.16 | Applying `sizeof` to an array name returns the number of bytes in the array. (Part 2 of 2.)

```c
// Fig. 7.17: fig07_17.c
// Using operator sizeof to determine standard data type sizes.
#include <stdio.h>

int main(void)
{
   char c;
   short s;
   int i;
   long l;
   long long ll;
   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 = %u\tsizeof(char)  = %u"
          "\n     sizeof s = %u\tsizeof(short) = %u"
          "\n     sizeof i = %u\tsizeof(int) = %u"
          "\n     sizeof l = %u\tsizeof(long) = %u"
          "\n    sizeof ll = %u\tsizeof(long long) = %u"
          "\n     sizeof f = %u\tsizeof(float) = %u"
```

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

```
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   }
```

```
     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
```

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

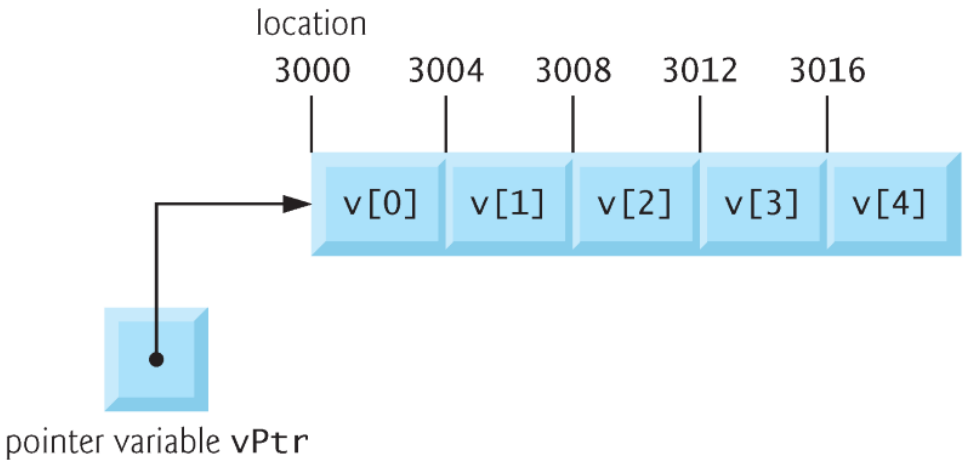


Fig. 7.18 | Array `v` and a pointer variable `vPtr` that points to `v`.

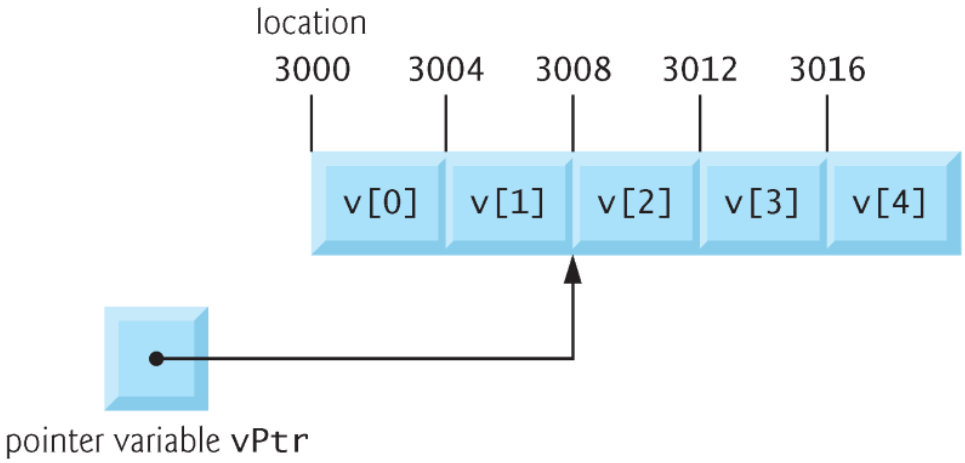


Fig. 7.19 | The pointer `vPtr` after pointer arithmetic.

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

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 1 of 3.)

```
23      // loop through array b
24      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
25         printf("*(b + %u) = %d\n", offset, *(b + offset));
26      }
27
28      // output array b using bPtr and array index notation
29      puts("\nPointer index notation");
30
31      // loop through array b
32      for (size_t i = 0; i < ARRAY_SIZE; ++i) {
33         printf("bPtr[%u] = %d\n", i, bPtr[i]);
34      }
35
36      // output array b using bPtr and pointer/offset notation
37      puts("\nPointer/offset notation");
38
39      // loop through array b
40      for (size_t offset = 0; offset < ARRAY_SIZE; ++offset) {
41         printf("*(bPtr + %u) = %d\n", offset, *(bPtr + offset));
42      }
43   }
```

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 2 of 3.)

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

Pointer/offset notation where
the pointer is the array name
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Pointer index notation
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 7.20 | Using indexing and pointer notations with arrays. (Part 3 of 3.)

```c
// Fig. 7.21: fig07_21.c
// Copying a string using array notation and pointer notation.
#include <stdio.h>
#define SIZE 10

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

int main(void)
{
   char string1[SIZE]; // create array string1
   char *string2 = "Hello"; // create a pointer to a string

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

   char string3[SIZE]; // create array string3
   char string4[] = "Good Bye"; // create an array containing a string

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

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

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

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

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

7.10 Arrays of Pointers

- Arrays may contain pointers.
- A common use of an array of pointers is to form an array of strings, referred to simply as a string array.
- Each entry in the array is a string, but in C a string is essentially a pointer to its first character.
- So each entry in an array of strings is actually a pointer to the first character of a string.
- Consider the definition of string array `suit`, which might be useful in representing a deck of cards.

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

7.10 Arrays of Pointers (Cont.)

- The `suit[4]` portion of the definition indicates an array of 4 elements.
- The `char *` portion of the declaration indicates that each element of array `suit` is of type "pointer to `char`."
- Qualifier `const` indicates that the strings pointed to by each element pointer will not be modified.
- The four values to be placed in the array are `"Hearts"`, `"Diamonds"`, `"Clubs"` and `"Spades"`.
- Each is stored in memory as a *null-terminated character string* that's one character longer than the number of characters between quotes.

7.10 Arrays of Pointers (Cont.)

- The four strings are 7, 9, 6 and 7 characters long, respectively.
- Although it appears as though these strings are being placed in the `suit` array, only pointers are actually stored in the array (Fig. 7.22).
- Each pointer points to the first character of its corresponding string.
- Thus, even though the `suit` array is *fixed* in size, it provides access to character strings of *any length*.
- This flexibility is one example of C's powerful data-structuring capabilities.

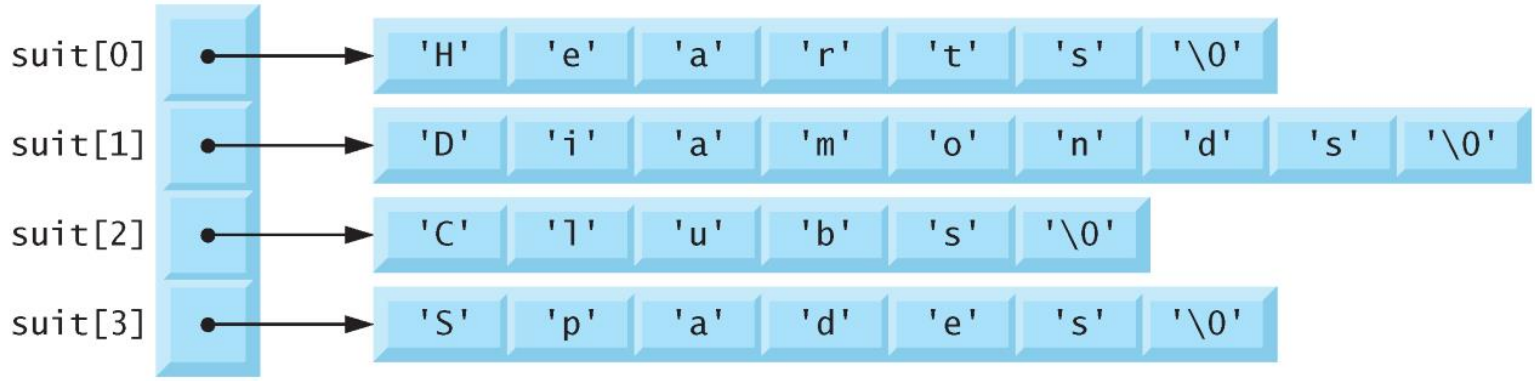


Fig. 7.22 | Graphical representation of the `suit` array.

7.10 Arrays of Pointers (Cont.)

- The suits could have been placed in a two-dimensional array, in which each row would represent a suit and each column would represent a letter from a suit name.
- Such a data structure would have to have a fixed number of columns per row, and that number would have to be as large as the largest string.
- Therefore, considerable memory could be wasted when storing a large number of strings of which most were shorter than the longest string.

```c
// Fig. 7.28: fig07_28.c
// Demonstrating an array of pointers to functions.
#include <stdio.h>

// prototypes
void function1(int a);
void function2(int b);
void function3(int c);

int main(void)
{
   // initialize array of 3 pointers to functions that each take an
   // int argument and return void
   void (*f[3])(int) = { function1, function2, function3 };

   printf("%s", "Enter a number between 0 and 2, 3 to end: ");
   size_t choice; // variable to hold user's choice
   scanf("%u", &choice);

```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 1 of 3.)

```
20      // process user's choice
21      while (choice >= 0 && choice < 3) {
22
23         // invoke function at location choice in array f and pass
24         // choice as an argument
25         (*f[choice])(choice);
26
27         printf("%s", "Enter a number between 0 and 2, 3 to end: ");
28         scanf("%u", &choice);
29      }
30
31      puts("Program execution completed.");
32   }
33
34   void function1(int a)
35   {
36      printf("You entered %d so function1 was called\n\n", a);
37   }
38
39   void function2(int b)
40   {
41      printf("You entered %d so function2 was called\n\n", b);
42   }
43
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 2 of 3.)

```
44  void function3(int c)
45  {
46     printf("You entered %d so function3 was called\n\n", c);
47  }
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Fig. 7.28 | Demonstrating an array of pointers to functions. (Part 3 of 3.)