Storage Class Specifiers

- There are four storage classes

1. Automatic (auto variable)
2. Register (register variable)
3. Static (static variable)
4. External (extern variable)

Automatic Variables (1 of 2)

- The default and the most commonly used storage class is automatic.
- Memory for automatic variables is allocated when a block or function is created. They are defined and are "local" to the block.
- When the block is exited, the system releases the memory that was allocated to the automatic variables, and their values are lost.
- These variables are also referred as local variables.

Register Variables (1 of 2)

- Register variables specify to the compiler that the variables should be stored in high-speed memory registers if possible. This is done for efficiency.
- If no register space is free, variables become auto instead.
- Keep register variables in small local blocks which are reallocated quickly.
- The address of a register variable is unavailable to the user. The user program can't use both the address operator and the indirection operator (pointer) with a register.

Register Variables (2 of 2)

- Characteristics:
  - Scope: block
  - Extent: automatic/register
  - Linkage: internal
- Declaration:

  register type variable name;
- Initialization:

  A register variable can be initialized where it is defined or left uninitialized.

Static Variables (1 of 3)

- The value of static variables persists until the end of the program.
- It is declared using the keyword **static** :

```
static int x;
static float y;
```

- By default, Static variables are initialized with zero.
- The static specifier has two different usages depending on its scope.
  1. static variable with block scope
  2. static variable with file scope.

Static Variables (2 of 3)

Static variable with block scope

- When a static variable is declared in a block scope, static defines the extent of the variable
- Static variables with Block Scope are also known as Static Local Variables

Scope: Block
Extent: Static
Linkage: Internal

- Declaration:

  static type variable_name

- Initialization:
  - A static variable name can be initialized where it is defined, or it can be left uninitialized. If it is initialized, it is initialized only once. If not initialized, its value will be initialized to zero

Static Variable (3 of 3)

Static variable with file scope

- When the static specifier is used with a variable that has a file scope (global scope) and its linkage is internal.

Scope: File

Extent: Static

Linkage: Internal

Declaration:

Static type variable name;

```c
#include <stdio.h>
int main()
{
   printf("%d",func());
   printf("\n%d",func());

    return 0;
}
int func()
{
    static int count=0;
    count++;
    return count;
}
```

```c
#include <stdio.h>
int main()
{
        static int x;
        int y;
        printf("%d \n %d", x, y);
}
```

External Variables (1 of 2)

- Extern is used for global variables that are shared across code in several files.
- External variables are used with separate compilations.
- Large projects are decomposed in to many source files.
- Decomposed source files are compiled separately and linked together to form a unit.
- Storage is permanently assigned to Global variables (defined outside functions) and all functions of the storage class extern.

External Variables (2 of 2)

- Characteristics:
  - Scope: file
  - Extent: static
  - Linkage: external

- Declaration:

```
extern type variable_name;
```

```
// variables.c// declaring and defining a variable
float extern_var = 19.0;
```

```
// main.c
#include <stdio.h>
#include "variables.c“
// making the variable, an extern variable
extern float extern_var;
int main()
{
// print the variable from another file
printf("%f", extern_var);
return 0;
}
```

```c
//globals.h
// declaring an extern variable
extern int common_variable; //declaring
```

```c
//changer.c

#include <stdio.h>
#include "globals.h"
// defining the extern global variable
int common_variable = 20;

void increment()
{
    printf("Before increment: %d\n", common_variable);
    // incrementing the extern variable
    common_variable++;
    printf("After increment: %d\n", common_variable);
}
```

```c
// main.c
#include <stdio.h>
#include "changer.c"
int main()
{
    // accessing the global variable
    printf("Global Variable: %d\n", common_variable);
    // accessing the global function
    increment();
    // incrementing the extern variable
    common_variable++;
    increment();
    return 0;
}
```

```c
//functions.c

// declaring an extern function
extern float get_constant()
{
    return 3.14;
}
```

```c
#include <stdio.h>
// include the file
#include "functions.c"
int main()
{
// calling the function
    printf("%f", get_constant());
    return 0;
}
```

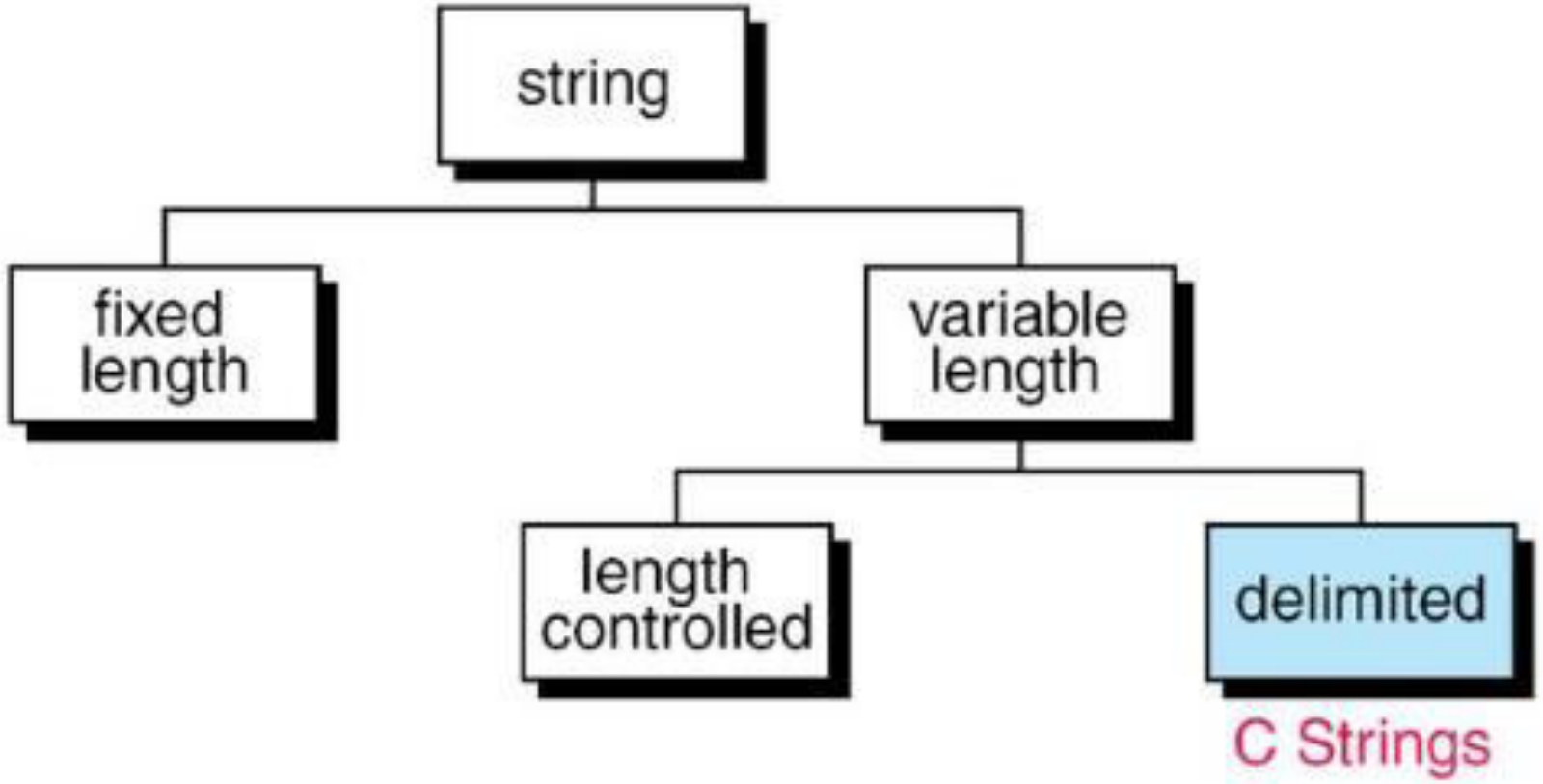
Summary of Storage Classes

Class	Scope	Extent	Linkage	Keyword
auto	block	automatic	internal	auto or none
register	block	automatic	internal	register
static(extent)	block	static	internal	static
static(linkage)	file	static	internal	static
extern	file	static	internal	extern or none

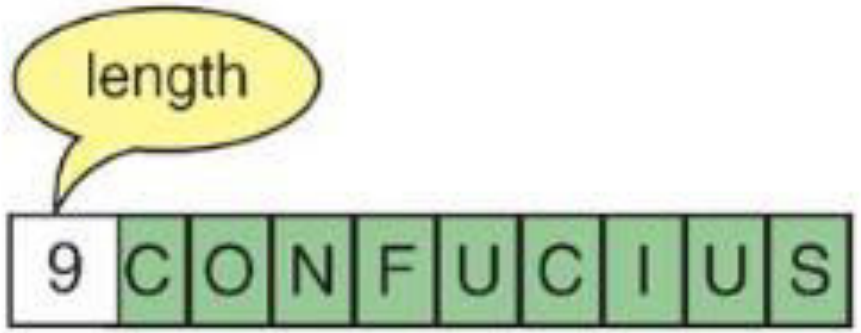
STRINGS

String Concepts

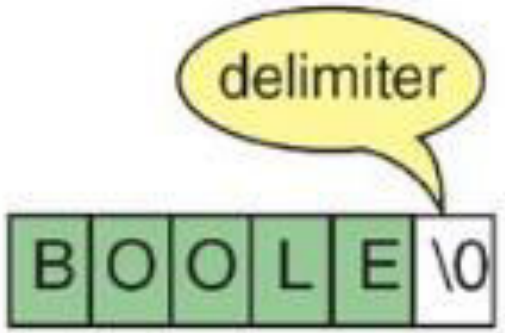
In general, a string is a series of characters treated as a unit. Computer science has long recognized the importance of strings, but it has not adapted a standard for their implementation. We find, therefore, that a string created in Pascal differs from a string created in C.



String Taxonomy



Length-controlled string



Delimited string

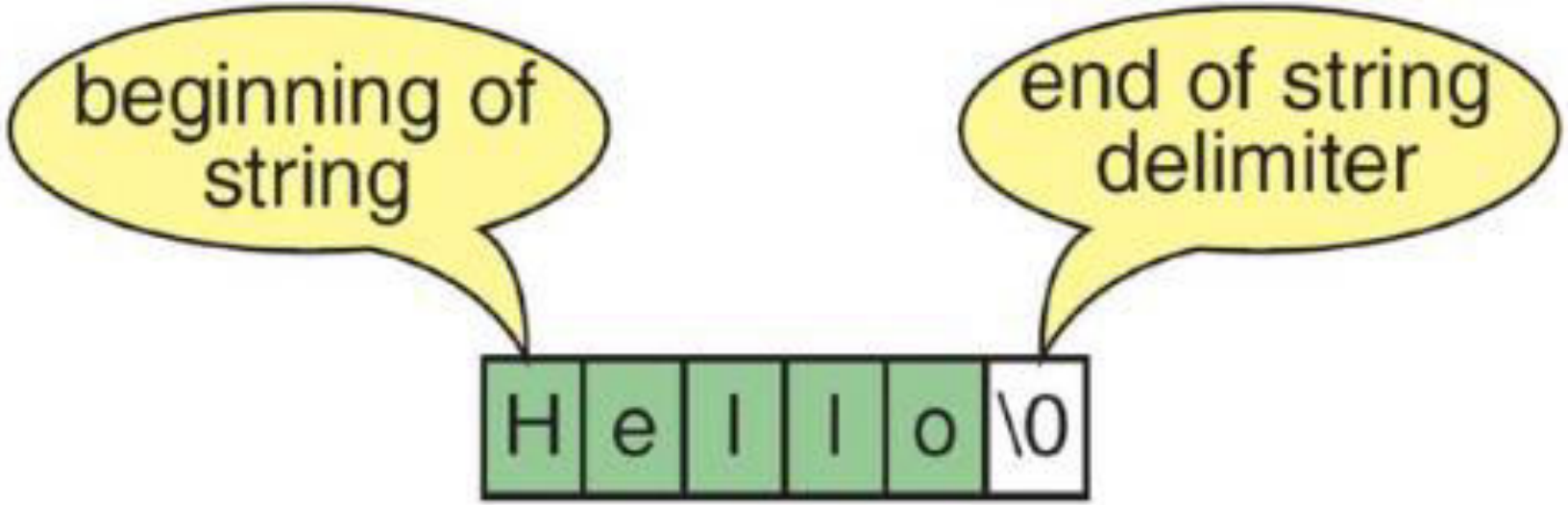
String Formats

C Strings

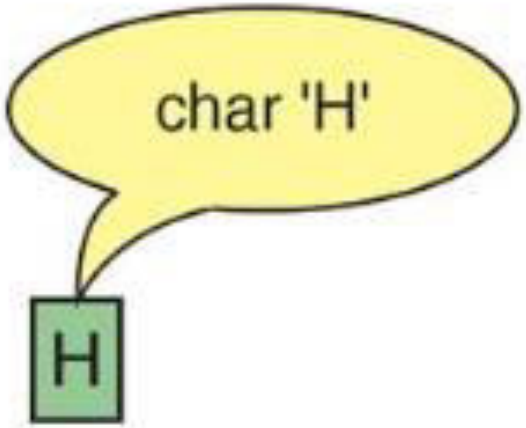
A C string is a variable-length array of characters that is delimited by the null character.

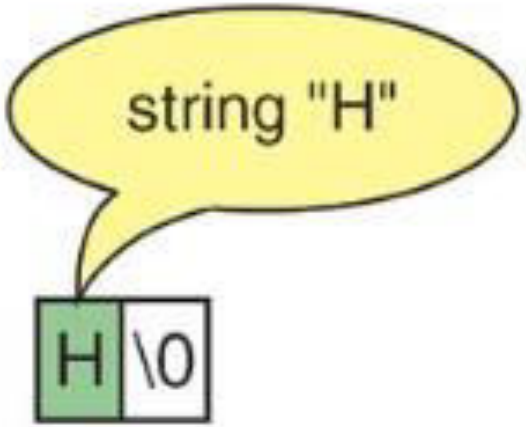
C uses variable-length, delimited strings.

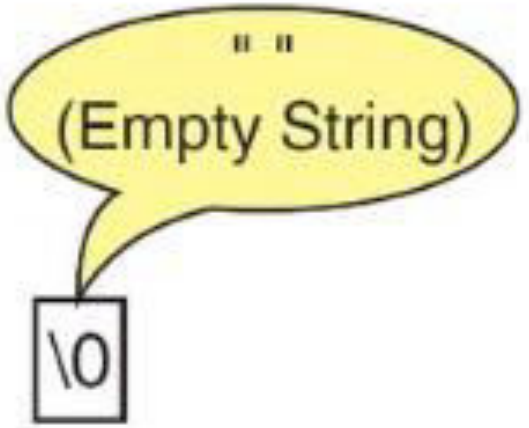
A string literal is enclosed in double quotes.



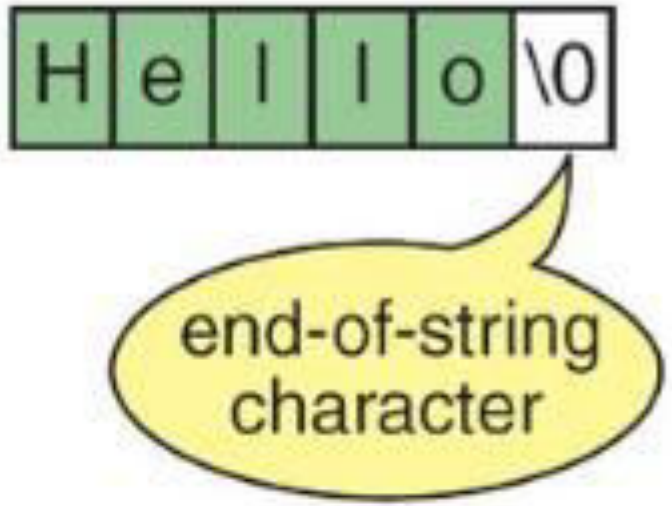
Storing Strings

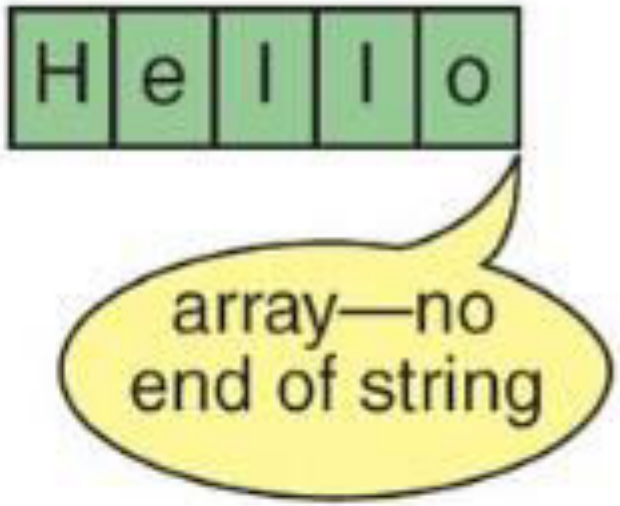






Storing Strings and Characters





Differences Between Strings and Character Arrays

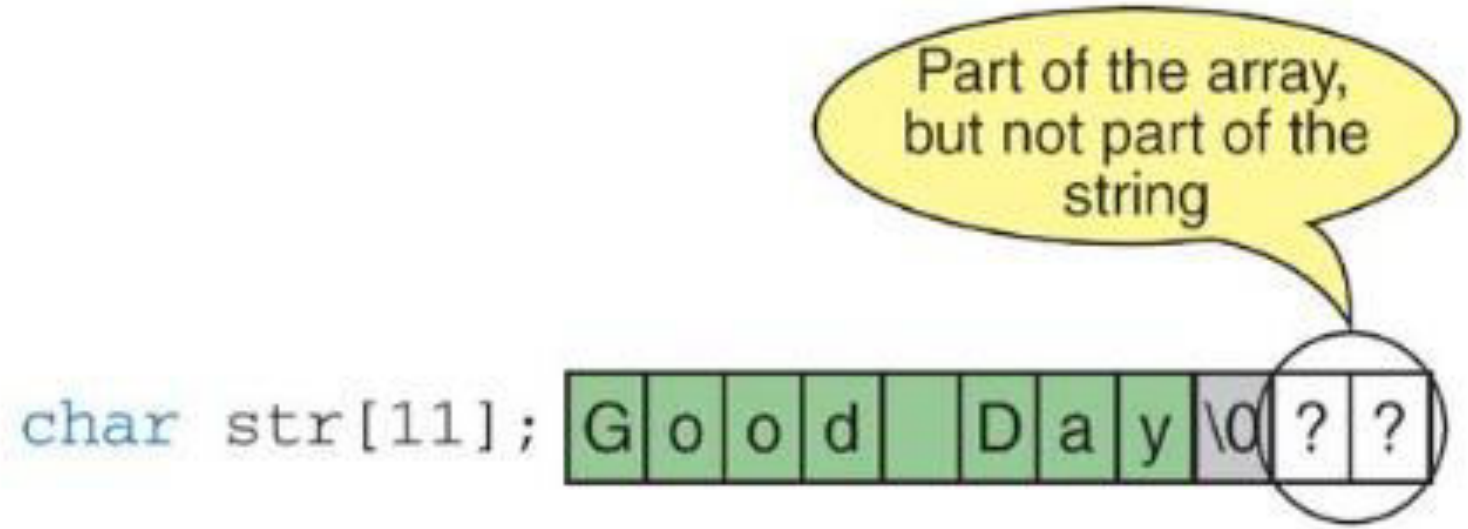
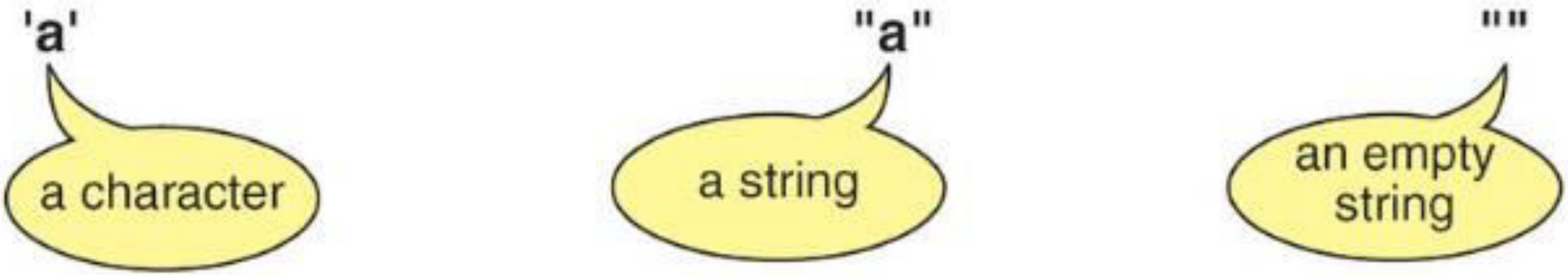
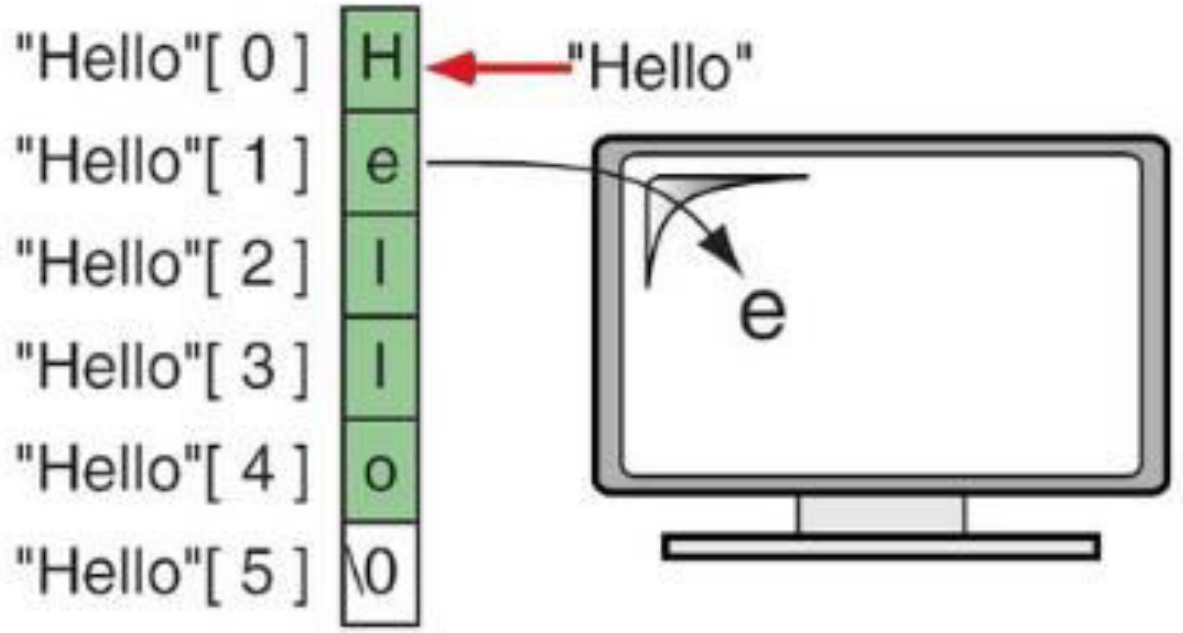


FIGURE 11-6 Strings in Arrays



Character Literals and String Literals

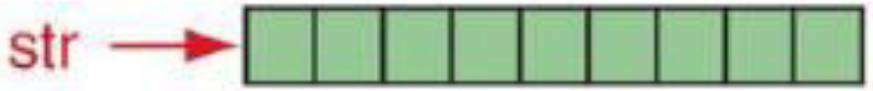
```c
#include <stdio.h>
int main (void)
{
  printf("%c\n", "Hello"[1];
  return 0;
} // main
```



String Literal References

```
// Local Declarations
   char str[9];
```

(a) String Declaration

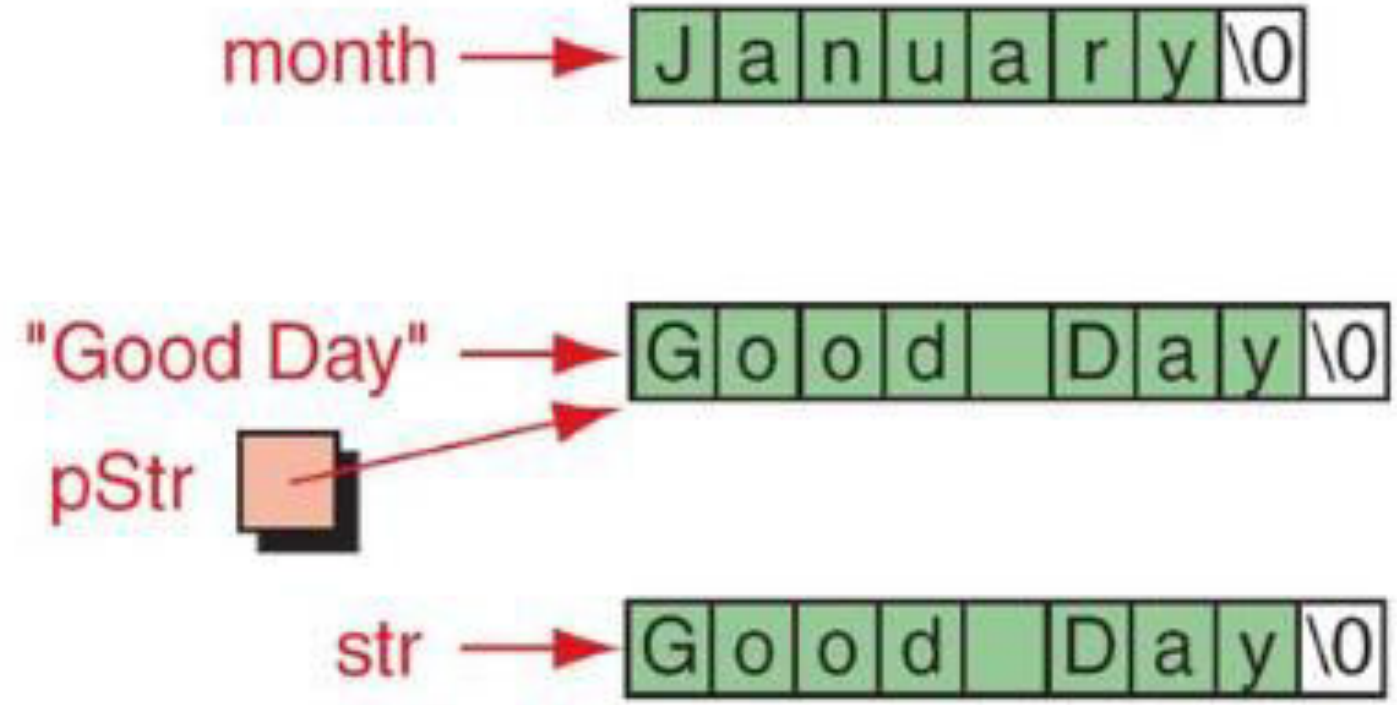



```
// Local Declarations
   char* pStr;
```

(b) String Pointer Declaration

Note

Memory for strings must be allocated before the string can be used.



Initializing Strings

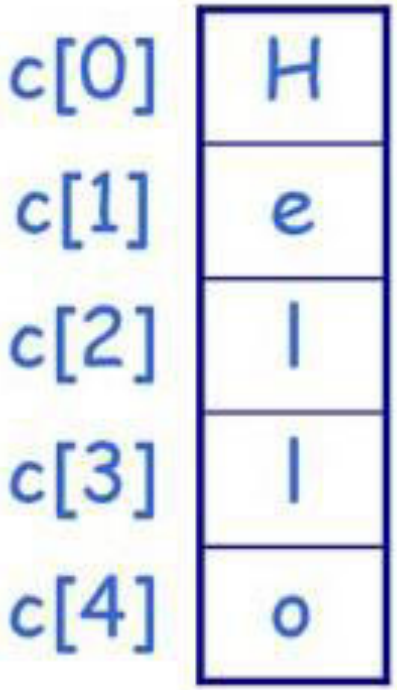
Arrays of Strings

Ragged arrays are very common with strings. Consider, for example, the need to store the days of the week in their textual format. We could create a two-dimensional array of seven days by ten characters, but this wastes space.

Character Arrays

- In a character array, each element stores one character.

e.g. `char c[5];`




```
c[0]='H'; c[1]='e'; c[2]=c[3]='l'; c[4]='o';

c[0]=72; c[1]=101; c[2]=c[3]=108; c[4]=111;
```

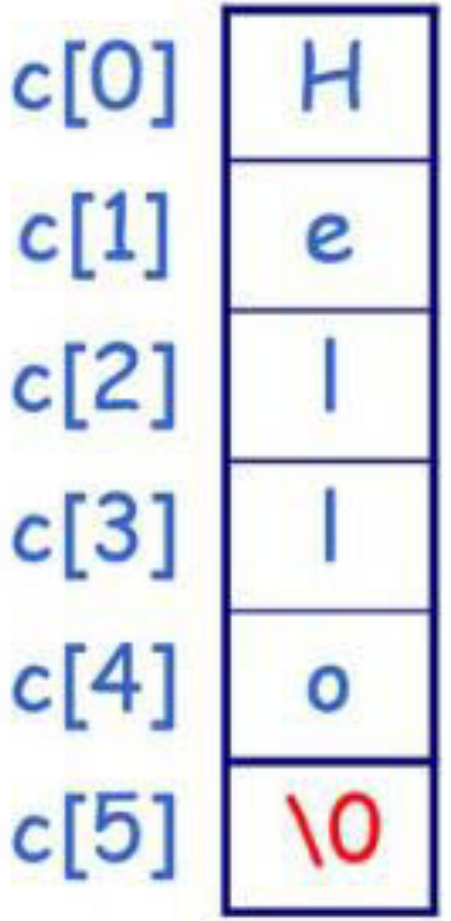
Character Arrays - Initialization

- Like other type one-dimensional arrays, the character array can be initialized in the same way.

  e.g. `char c[5] = { 'H', 'e', 'l', 'l', 'o' };`

- If the number of initializers is less than the declared size, the remaining elements are initialized to null character ('\0').

  e.g. `char c[6] = { 'H', 'e', 'l', 'l', 'o' };`



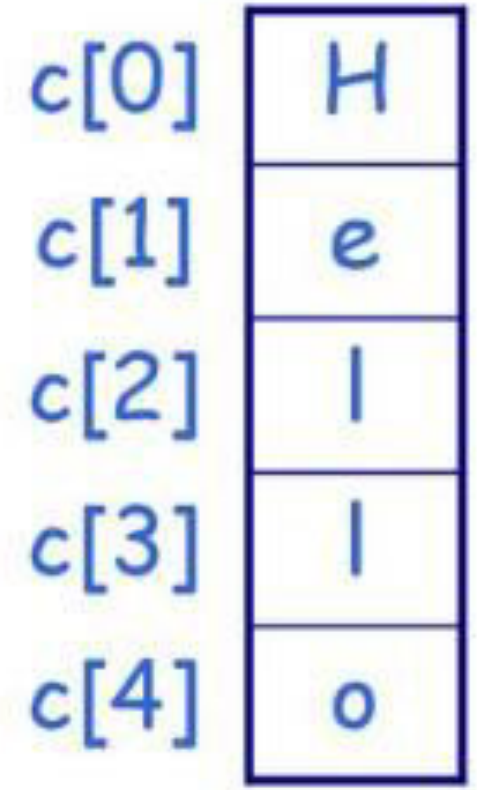
Character Arrays - Initialization

- Like other type one-dimensional arrays, the character array can be initialized in the same way.

  e.g. `char c[5] = { 'H', 'e', 'l', 'l', 'o' };`

- If the number of initializers is **equal to** the declared size, the size may be omitted.

  e.g. `char c[ ] = { 'H', 'e', 'l', 'l', 'o' };`



Character Arrays

- The character array can be used as other type one-dimensional arrays.

I am happy

```
main()
{  char  c[10] = { 'I', '␣', 'a', 'm', '␣', 'h', 'a', 'p', 'p', 'y' };
   int  i;

   for ( i = 0; i < 10; i++ )
        printf ( "%c", c[i] );
}
```

Strings

- A string is a character array.
  - char c[10] = { 'I', '␣', 'a', 'm', '␣', 'h', 'a', 'p', 'p', 'y' };

    The actual length of the string is **not always** equal to the size of the character array.
  - In fact, we often concern the actual length of a string rather than the size of a character array.

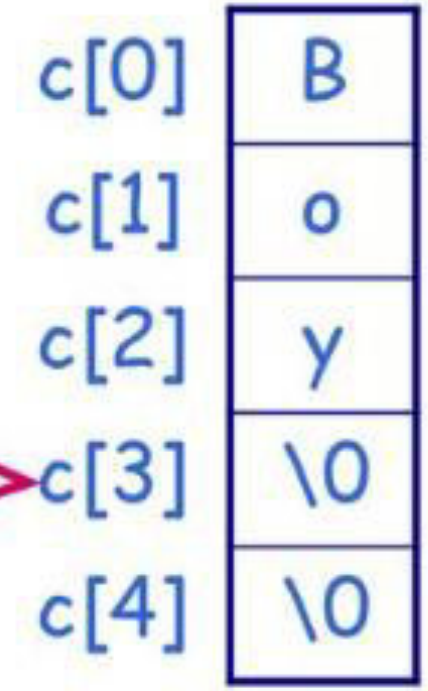
    char c[100] = { 'I', '␣', 'a', 'm', '␣', 'h', 'a', 'p', 'p', 'y' };

    The size of the character array is 100, but the actual length of the string is only 10.

Strings

- The array size is often much larger than the size of the string stored in the array. In order to represent the end of the string, the null character '\0' is used as the "end-of-string" marker.

```
char c[5] = { 'B', 'o ', 'y' };
```



c[3] is the "end-of-string" marker '\0', so the elements before it compose the string, and the length of the string is 3.

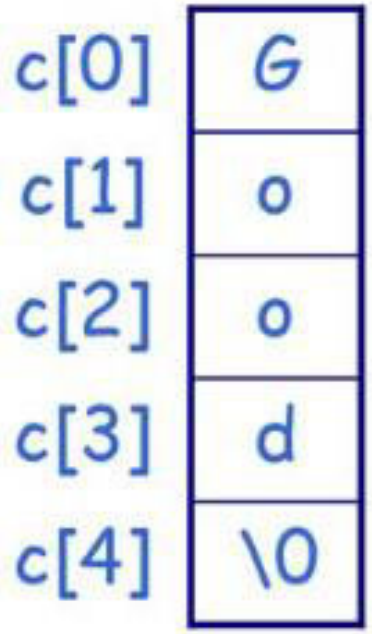
Strings

- When a **string constant** is stored in memory, it also uses the null character '\0' as the "end-of-string" marker.
  - For example: When the string **"Hello"** is stored in memory, it requires 6 bytes storage, and the last byte is used to store the null character '\0'.

Strings - Initialization

- Using characters.

```
char c[5] = { 'G', 'o ', 'o', 'd' };
```



- Using string.

```
char c[5] = { "Good" };
char c[5] = "Good";
char c[ ] = "Good";
```

Notice: The array c consists of 5 elements but not 4 elements! This declaration is equivalent to:

```
char c[5] = "Good";
```

It differs with

```
char c[4] = "Good";
```

Strings - Initialization

- The initialization of table of strings.

```
char fruit[ ][7] = { "Apple", "Orange",
                     "Grape", "Pear", "Peach"};
```

fruit[0]	A	p	p	l	e	\0	\0
fruit[1]	O	r	a	n	g	e	\0
fruit[2]	G	r	a	p	e	\0	\0
fruit[3]	P	e	a	r	\0	\0	\0
fruit[4]	P	e	a	c	h	\0	\0

11-3 String Input/Output Functions

C provides two basic ways to read and write strings. First, we can read and write strings with the formatted input/output functions, *scanf/fscanf* ***and*** *printf/fprintf.* ***Second, we can use a special set of string-only functions, get string (****gets/fgets****) and put string (*** *puts/fputs* ***).***

```
// Read and print only second integer
while (fscanf(spData,
        " %*d%d%*[^\n]", &amount) != EOF)
    printf("Second integer: %4d\n", amount);
```

```
Input:
123 456 7.89
987 654 3.21

Results:
Second integer:  456
Second integer:  654
```

- gets( )
  - Form: gets (str)
  - It is used to read a line of text.

    Only the "newline" character marks the end of the input string. The "blank space" and "tab" will be read in as a character in t

```
main()
{       char c[50];
        gets ( c );
        puts ( c );
}
```

```
How are you?
How are you?
_
```

```
fgets (str, sizeof (str), stdin);
printf("Here is your string: \n\t%s", str);
```

```
fputs(pStr, stdout);
fputs("\n", stdout);
```

Function	Work of Function
strlen()	computes string's length
strcpy()	copies a string to another
strcat()	concatenates(joins) two strings
strcmp()	compares two strings
strlwr()	converts string to lowercase
strupr()	converts string to uppercase

STRING FUNCTIONS

- C provides a wide range of string functions for performing different string tasks
- Examples

  strlen(str) - calculate string length

  strcpy(dst,src) - copy string at src to dst

  strcmp(str1,str2) - compare str1 to str2
- Functions come from the utility library string.h
  - #include <string.h> to use

STRING LENGTH

Syntax: int strlen(char *str)

returns the length (integer) of the string argument

counts the number of characters until an \0 encountered

does not count \0 char

Example:

char str1 = “hello”;

strlen(str1) would return 5

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char a[20]="Program";
    char b[20]={'P','r','o','g','r','a','m','\0'};

    // using the %zu format specifier to print size_t
    printf("Length of string a = %zu \n",strlen(a));
    printf("Length of string b = %zu \n",strlen(b));

    return 0;
}
```

COPYING A STRING

Syntax:

char *strcpy(char *dst, char *src)

- copies the characters (including the \0) from the source string (src) to the destination string (dst)
- dst should have enough space to receive entire string (if not, other data may get written over)
- if the two strings overlap (e.g., copying a string onto itself) the results are unpredictable
- return value is the destination string (dst)

char *strncpy(char *dst, char *src, int n)

- similar to strcpy, but the copy stops after n characters
- if n non-null (not \0) characters are copied, then no \0 is copied

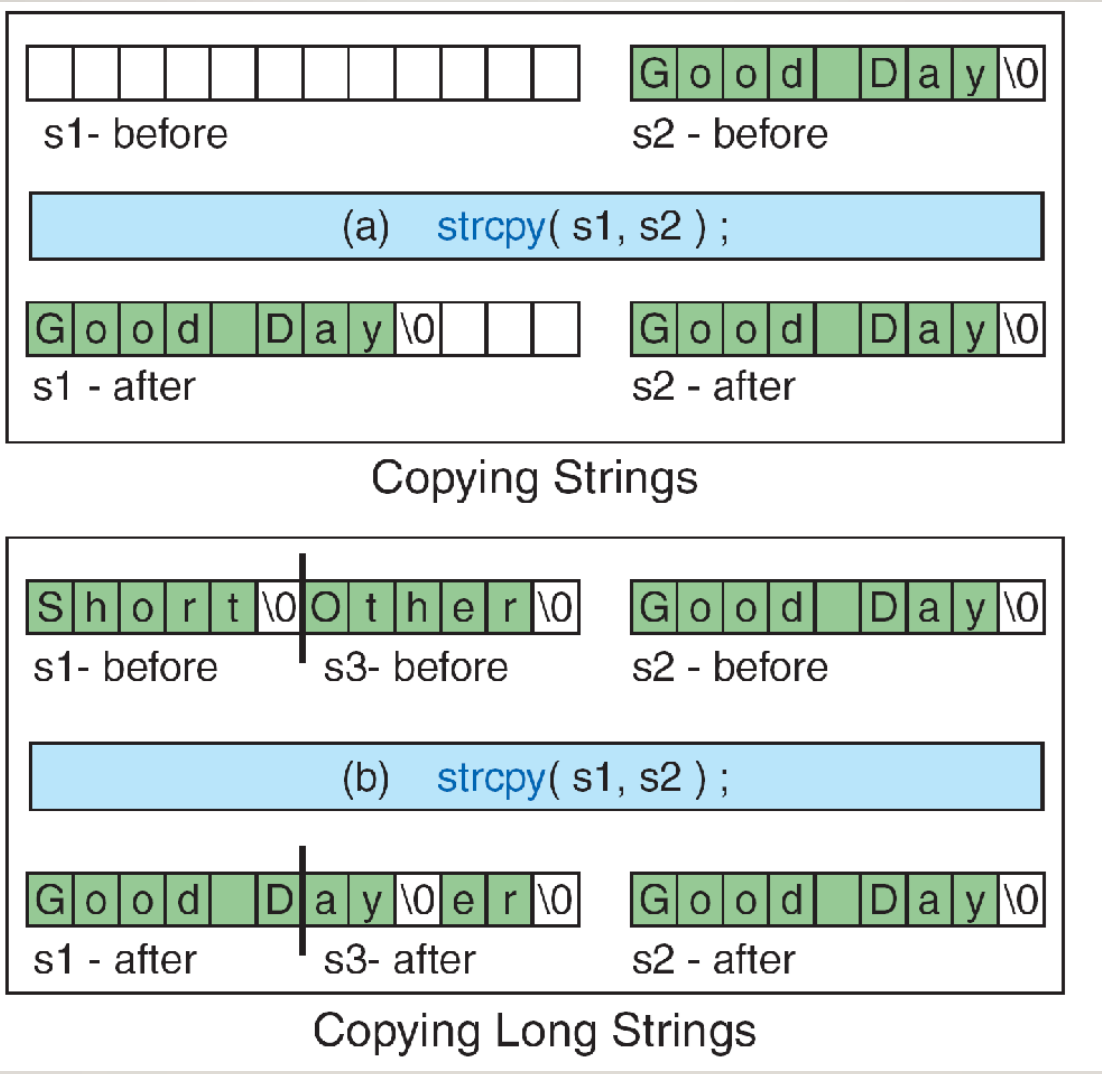


FIGURE 11-14 String Copy

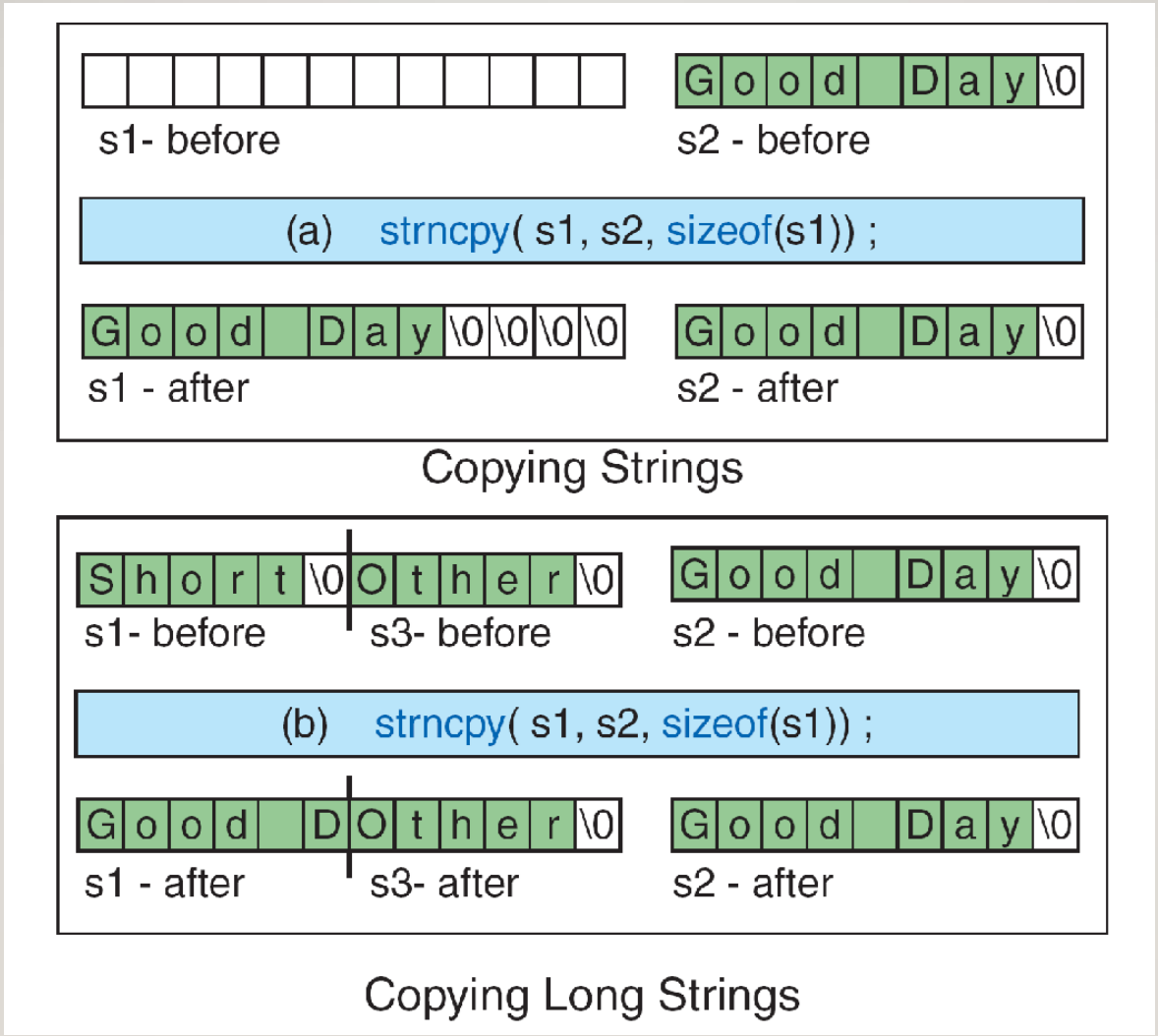


FIGURE 11-15 String-number Copy

```c
#include <stdio.h>
#include <string.h>

int main() {
   char str1[20] = "C programming";
   char str2[20];

   // copying str1 to str2
   strcpy(str2, str1);

   puts(str2); // C programming

   return 0;
}
```

STRING COMPARISON

Syntax:

int strcmp(char *str1, char *str2)

compares str1 to str2, returns a value based on the first character they differ at:

less than 0

if ASCII value of the character they differ at is smaller for str1

or if str1 starts the same as str2 (and str2 is longer)

greater than 0

if ASCII value of the character they differ at is larger for str1

or if str2 starts the same as str1 (and str1 is longer)

0 if the two strings do not differ

STRING COMPARISON (CONT)

strcmp examples:

strcmp(“hello”,”hello”) -- returns 0

strcmp(“yello”,”hello”) -- returns value > 0

strcmp(“Hello”,”hello”) -- returns value < 0

strcmp(“hello”,”hello there”) -- returns value < 0

strcmp(“some diff”,”some dift”) -- returns value < 0

expression for determining if two strings s1,s2 hold the same string value:

!strcmp(s1,s2)

STRING COMPARISON (CONT)

Sometimes we only want to compare first n chars:

int strncmp(char *s1, char *s2, int n)

Works the same as strcmp except that it stops at the nth character

looks at less than n characters if either string is shorter than n

strcmp(“some diff”,”some DIFF”) -- returns value > 0

strncmp(“some diff”,”some DIFF”,4) -- returns 0

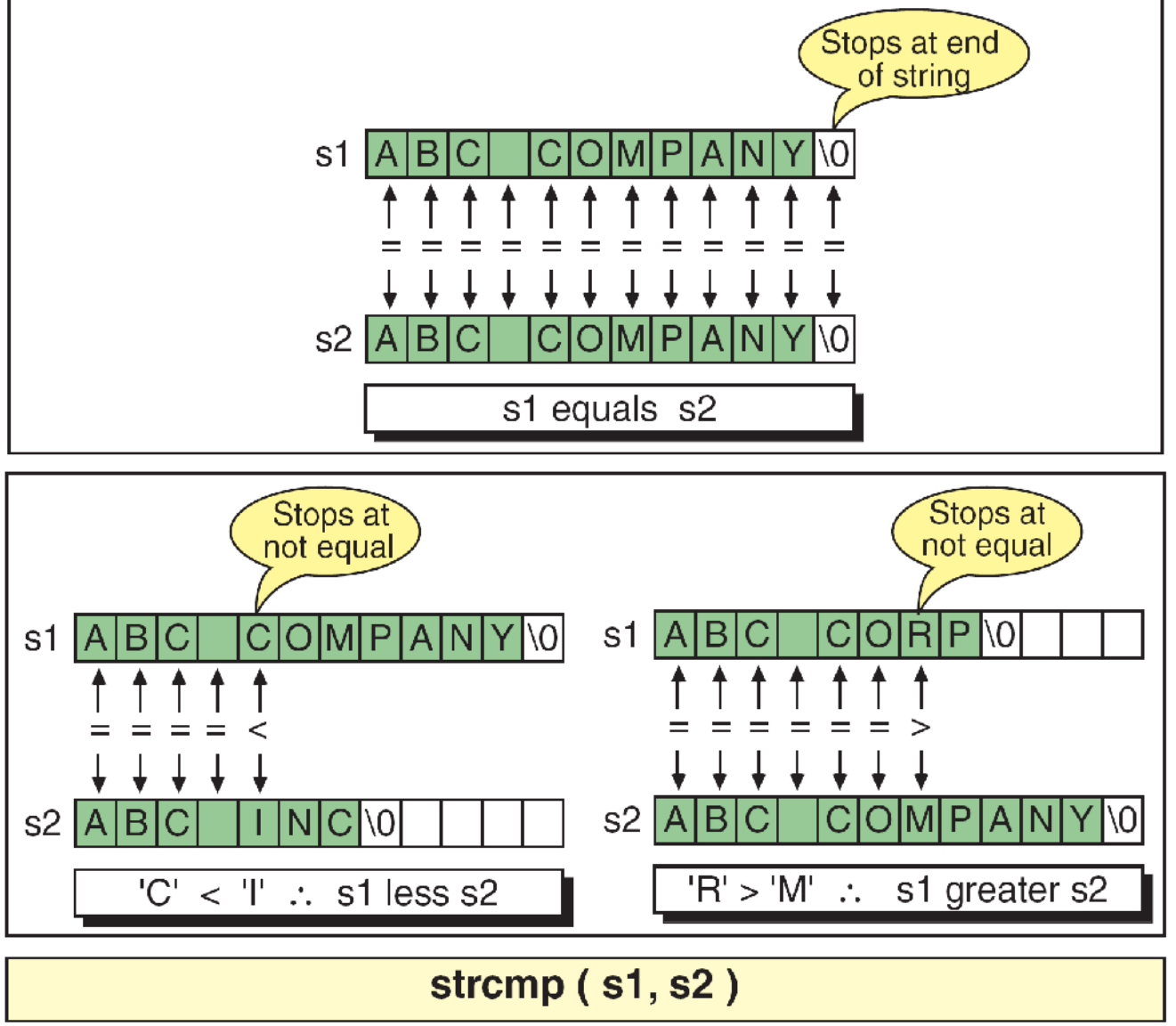


FIGURE 11-17 String Compares

STRING COMPARISON (IGNORING CASE)

Syntax:

int strcasecmp(char *str1, char *str2)

similar to strcmp except that upper and lower case characters (e.g., ‘a’ and ‘A’) are considered to be equal

int strncasecmp(char *str1, char *str2, int n)

version of strncmp that ignores case

```c
#include <stdio.h>
#include <string.h>

int main() {
  char str1[] = "abcd", str2[] = "abCd", str3[] = "abcd";
  int result;

  // comparing strings str1 and str2
  result = strcmp(str1, str2);
  printf("strcmp(str1, str2) = %d\n", result);

  // comparing strings str1 and str3
  result = strcmp(str1, str3);
  printf("strcmp(str1, str3) = %d\n", result);

  return 0;
}
```

STRING CONCATENATION

Syntax:

char *strcat(char *dstS, char *addS)

- appends the string at addS to the string dstS (after dstS's delimiter)
- returns the string dstS
- can cause problems if the resulting string is too long to fit in dstS

char *strncat(char *dstS, char *addS, int n)

- appends the first n characters of addS to dstS
- if less than n characters in addS only the characters in addS appended
- always appends a \0 character

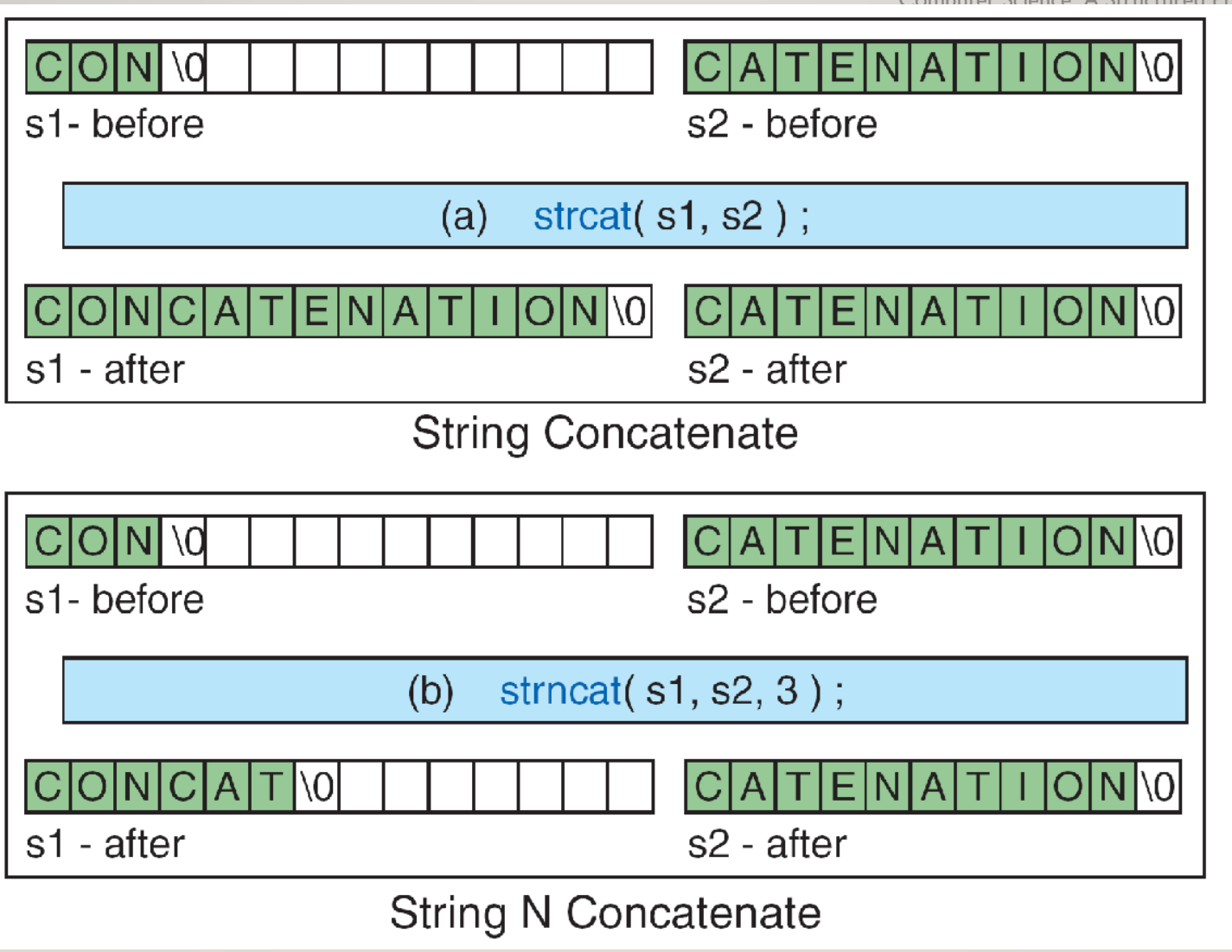


FIGURE 11-18 String Concatenation

```c
#include <stdio.h>
#include <string.h>
int main() {
    char str1[100] = "This is ", str2[] = "programiz.com";

    // concatenates str1 and str2
    // the resultant string is stored in str1.
    strcat(str1, str2);

    puts(str1);
    puts(str2);

    return 0;
}
```

SEARCHING FOR A CHARACTER/STRING

Syntax:

- char *strchr(char *str, int ch)
  - returns a pointer (a char *) to the first occurrence of ch in str
  - returns NULL if ch does not occur in str
  - can subtract original pointer from result pointer to determine which character in array
- char *strstr(char *str, char *searchstr)
  - similar to strchr, but looks for the first occurrence of the string searchstr in str
- char *strrchr(char *str, int ch)
  - similar to strchr except that the search starts from the end of string str and works backward

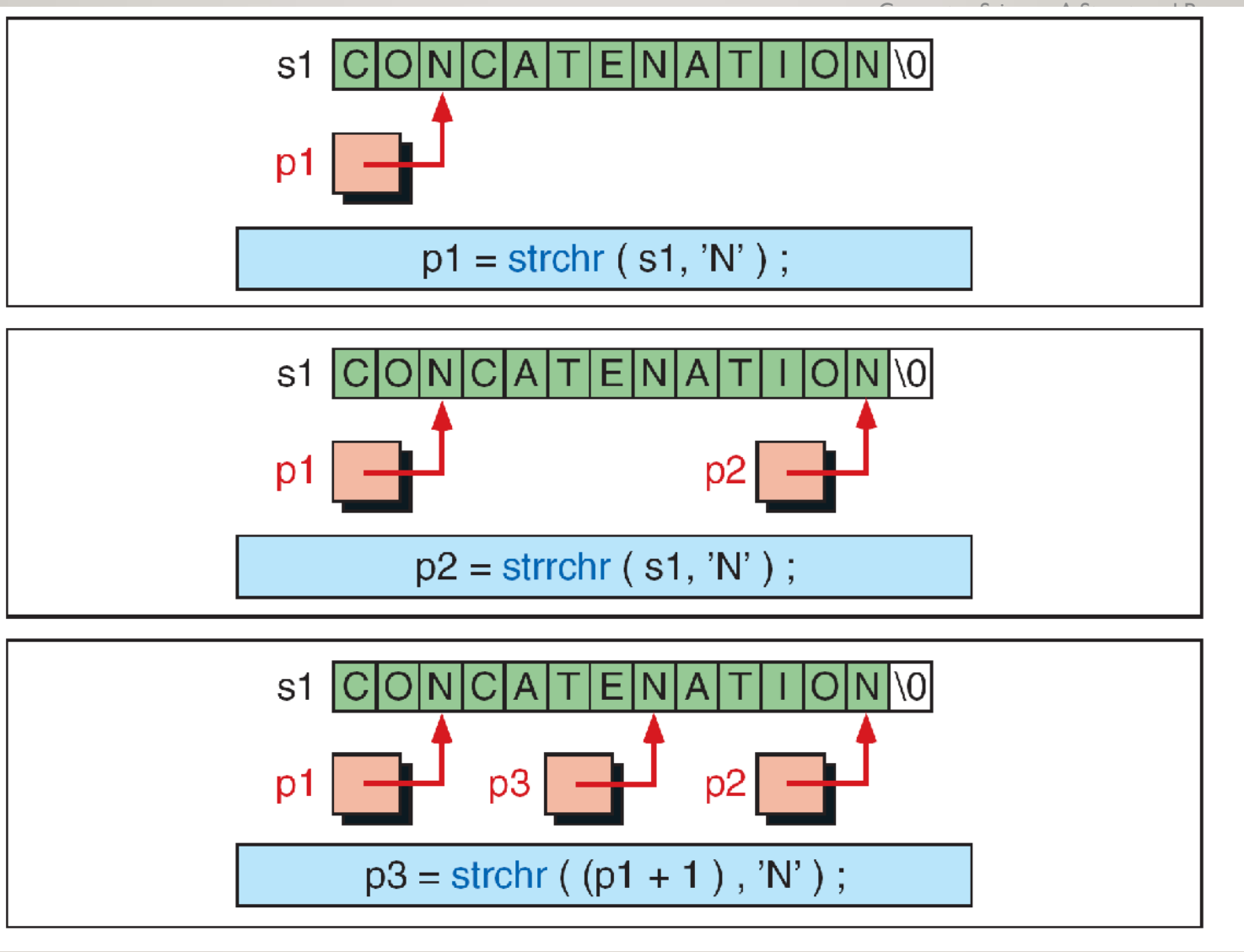


FIGURE 11-19 Character in String (*strchr*)

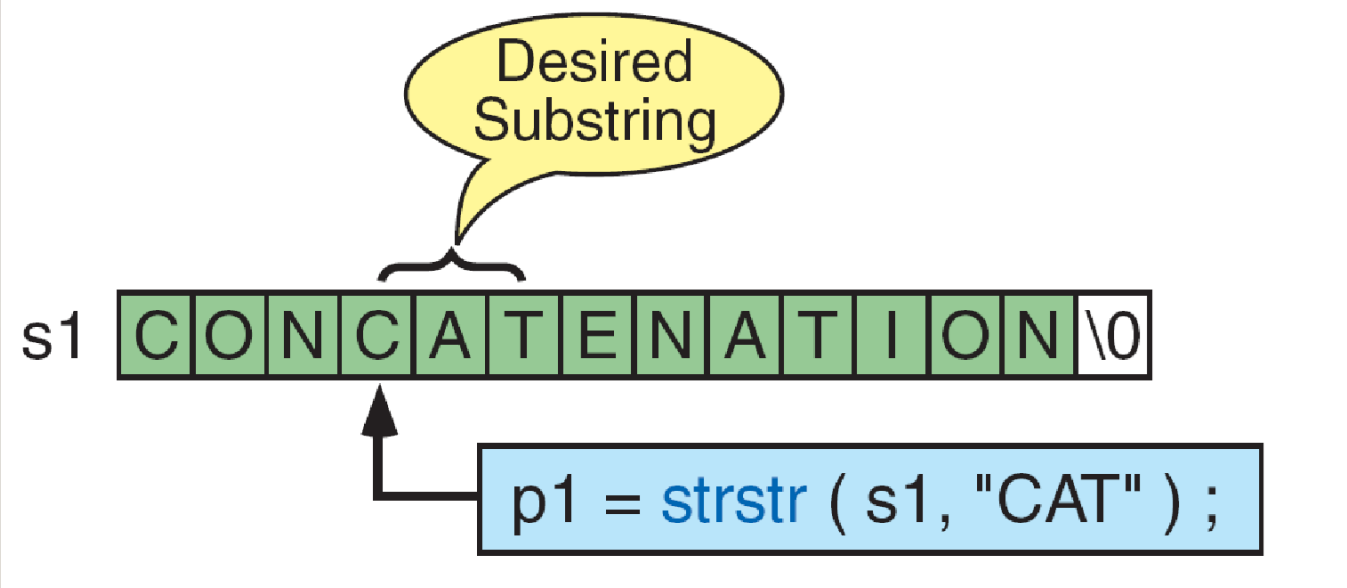


FIGURE 11-20 String in String

STRING SPANS (SEARCHING)

Syntax:

- int strspn(char *str, char *cset)
  - specify a set of characters as a string cset
  - strspn searches for the first character in str that is not part of cset
  - returns the number of characters in set found before first non-set character found
- int strcspn(char *str, char *cset)
  - similar to strspn except that it stops when a character that is part of the set is found

Examples:

- strspn(“a vowel”,”bvcwl”) returns 2
- strcspn(“a vowel”,”@,*e”) returns 5

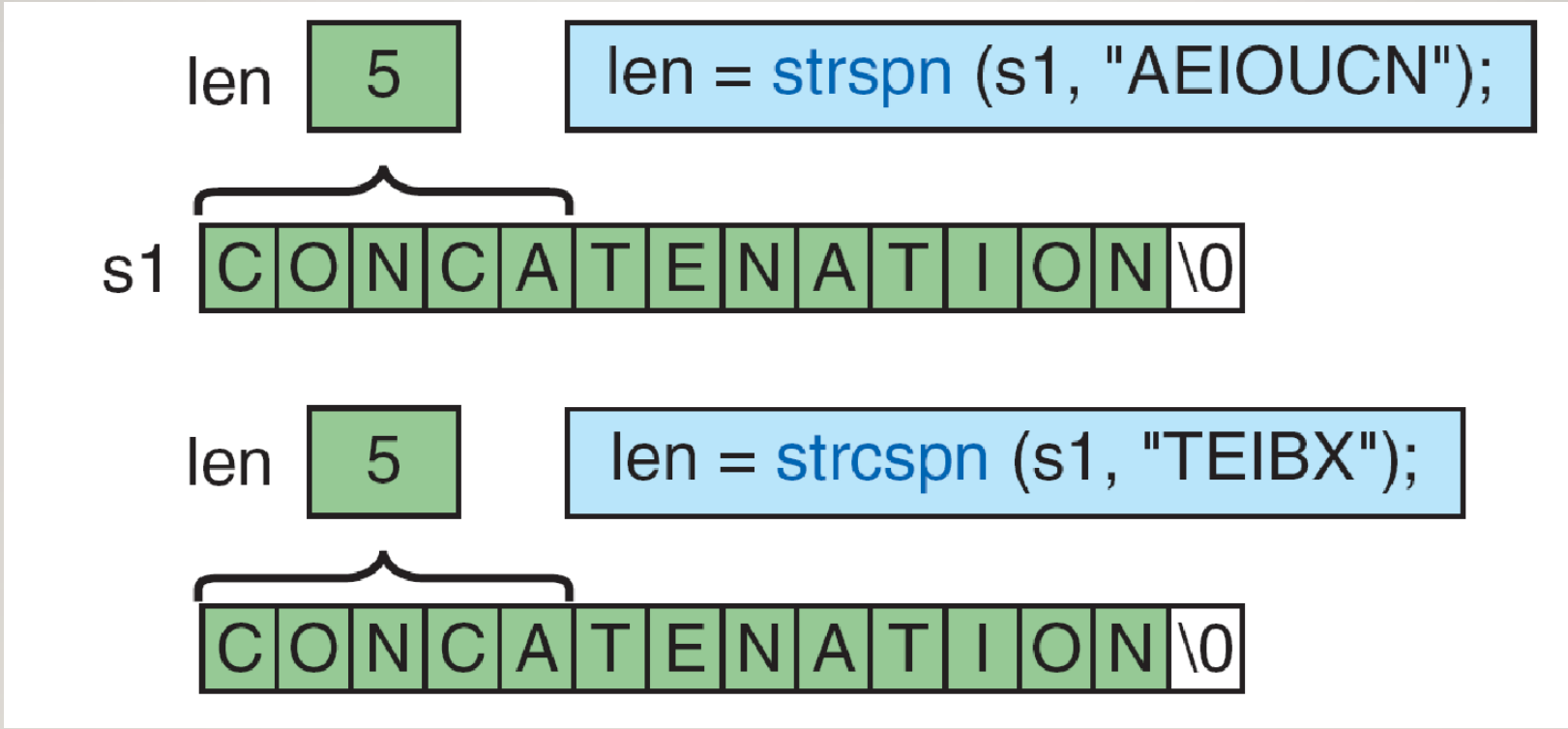


FIGURE 11-21 String Span

PARSING STRINGS

The strtok routine can be used to break a string of characters into a set of tokens

- we specify the characters (e.g., whitespace) that separate tokens
- strtok returns a pointer to the first string corresponding to a token, the next call returns the next pointer, etc.
- example:

  call strtok repeatedly on "A short string\n" with whitespace (space, \t, \n) as delimiters, should return

  first call: pointer to string consisting of "A"
  second call: pointer to string consisting of "short"
  third call: pointer to string consisting of "string"
  fourth call: NULL pointer

PARSING STRINGS

Syntax:

char *strtok(char *str, char *delimiters)

- delimiters is a string consisting of the delimiter characters (are ignored, end tokens)
- if we call strtok with a string argument as the first argument it finds the first string separated by the delimiters in str
- if we call strtok with NULL as the first argument it finds the next string separated by the delimiters in the string it is currently processing
- strtok makes alterations to the string str (best to make a copy of it first if you want a clean copy)

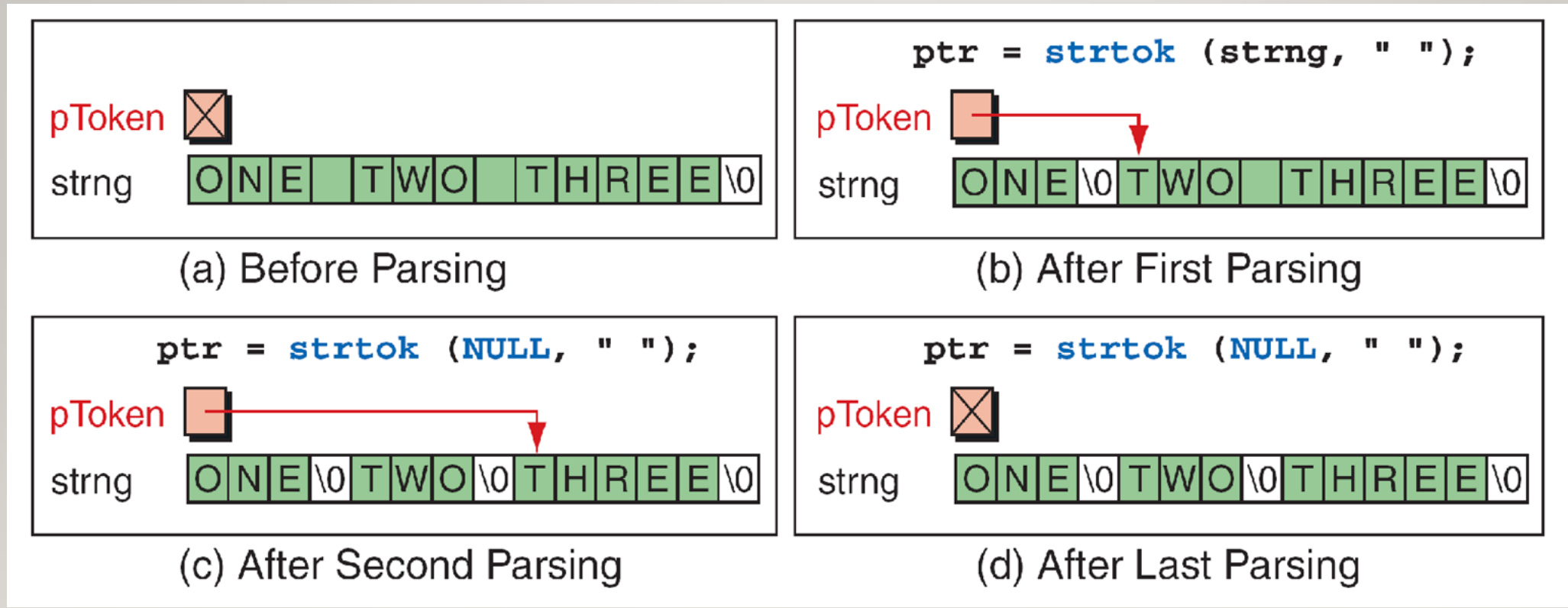
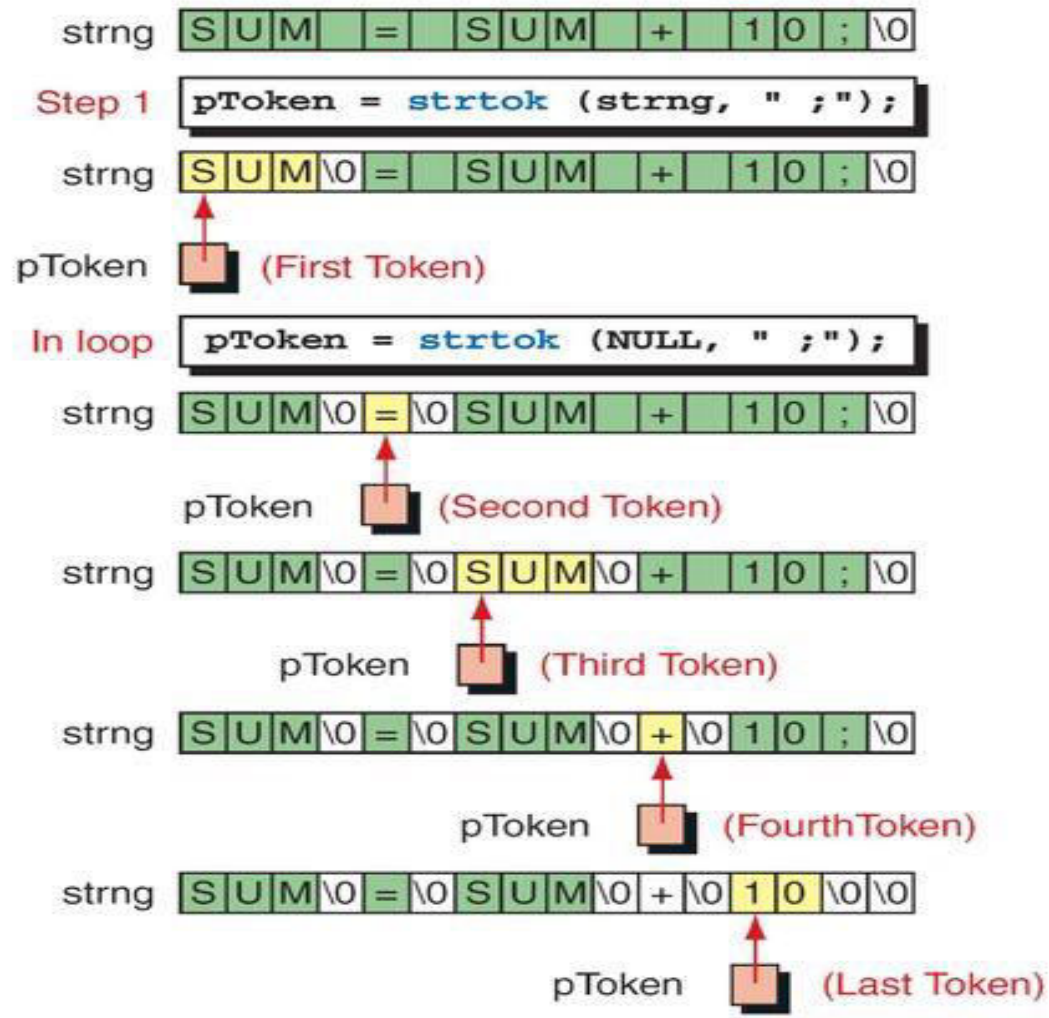


FIGURE 11-22 Streams



Parsing with String Token

Parsing a String with String Token

```
 1  /* Parse a simple algebraic expression.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <string.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     char  strng [16] = "sum = sum + 10;";
12     char* pToken;
13     int   tokenCount;
14
15  // Statements
16     tokenCount = 0;
17     pToken = strtok (strng, " ;");
18
```

Parsing a String with String Token

```
19      while (pToken)
20          {
21           tokenCount++;
22           printf("Token %2d contains %s\n",
23                     tokenCount, pToken);
24           pToken = strtok (NULL, " ;");
25          } // while
26
27      printf("\nEnd of tokens\n");
28      return 0;
29  }   // main
```

```
Results:
Token   1 contains sum
Token   2 contains =
Token   3 contains sum
Token   4 contains +
Token   5 contains 10

End of tokens
```

STRTOL()

- long int strtol(const char *str, char **endptr, int base)
- converts the initial part of the string in str to a long int value according to the given base,
- Base must be between 2 and 36 inclusive, or be the special value 0.

STRTOL()

- str − This is the string containing the representation of an integral number.
- endptr − This is the reference to an object of type char*, whose value is set by the function to the next character in str after the numerical value.
- base − This is the base, which must be between 2 and 36 inclusive, or be the special value 0.
- function returns the converted integral number as a long int value, else zero value is returned.

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

int main () {
   char str[30] = "2030300 This is test";
   char *ptr;
   long ret;

   ret = strtol(str, &ptr, 10);
   printf("The number(unsigned long integer) is %ld\n", ret);
   printf("String part is |%s|", ptr);

   return(0);
}
```

```
The number(unsigned long integer) is 2030300
String part is | This is test|
```

Demonstrate String to Long

```
 1  /* Demonstrate string to long function.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdlib.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     long  num;
12     char* ptr;
13
14  // Statements
15     num = strtol ("12345 Decimal constant: ", &ptr, 0);
16     printf("%s %ld\n", ptr, num);
17
18     num = strtol ("11001 Binary constant : ", &ptr, 2);
19     printf("%s %ld\n", ptr, num);
```

Demonstrate String to Long

```
20
21      num = strtol ("13572 Octal constant   : ", &ptr, 8);
22      printf("%s %ld\n", ptr, num);
23
24      num = strtol (" 7AbC Hex constant     : ", &ptr, 16);
25      printf("%s %ld\n", ptr, num);
26
27      num = strtol ("11001 Base 0-Decimal   : ", &ptr, 0);
28      printf("%s %ld\n", ptr, num);
29
30      num = strtol ("01101 Base 0-Octal     : ", &ptr, 0);
31      printf("%s %ld\n", ptr, num);
32
33      num = strtol ("0x7AbC Base 0-Hex        : ", &ptr, 0);
34      printf("%s %ld\n", ptr, num);
35
36      num = strtol ("Invalid input     : ", &ptr, 0);
37      printf("%s %ld\n", ptr, num);
```

Demonstrate String to Long

```
38
39      return 0;
40  }   // main
```

```
Results:
    Decimal constant:   12345
    Binary constant :   25
    Octal constant  :   6010
    Hex constant    :   31420
    Base 0-Decimal  :   11001
    Base 0-Octal    :   577
    Base 0-Hex      :   31420
    Invalid input   :   0
```

Numeric Format	ASCII Function	Wide-character Function
double	strtod	wcstod
float	strtof	wcstof
long double	strtold	wcstold
long int	strtol	wcstol
long long int	strtoll	wcstoll
unsigned long int	strtoul	wcstoul
unsigned long long int	strtoull	wcstoull

String-to-Number Functions

PRINTING TO A STRING

The sprintf function allows us to print to a string argument using printf formatting rules

First argument of sprintf is string to print to, remaining arguments are as in printf

Example:

```
char buffer[100];
sprintf(buffer,"%s, %s",LastName,FirstName);
if (strlen(buffer) > 15)
  printf("Long name %s %s\n",FirstName,LastName);
```

READING FROM A STRING

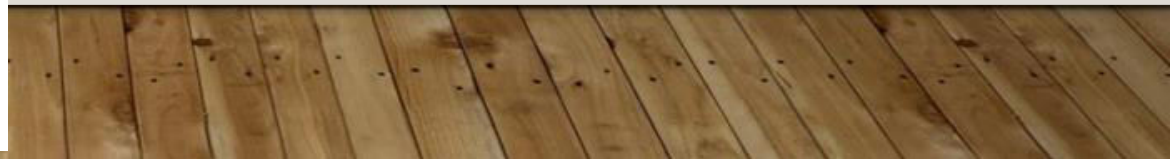
The sscanf function allows us to read from a string argument using scanf rules

First argument of sscanf is string to read from, remaining arguments are as in scanf

Example:

```
char buffer[100] = “A10 50.0”;
sscanf(buffer,”%c%d%f”,&ch,&inum,&fnum);
/* puts ‘A’ in ch, 10 in inum and 50.0 in fnum */
```

```c
#include <stdio.h>
#include <string.h>
int main(void)
{
  //variable
  char str[100], tmp;
  int i, len, mid;

  //input
  printf("Enter a string: ");
  gets(str);

  //find number of characters
  len = strlen(str);
  mid = len/2;

  //reverse
  for (i = 0; i < mid; i++) {
    tmp = str[len - 1 - i];
    str[len - 1 - i] = str[i];
    str[i] = tmp;
  }

  //output
  printf("Reversed string: %s\n", str);

  printf("End of code\n");
  return 0;
}
```



9-1 Introduction

A pointer is a constant or variable that contains an address that can be used to access data. Pointers are built on the basic concept of pointer constants.

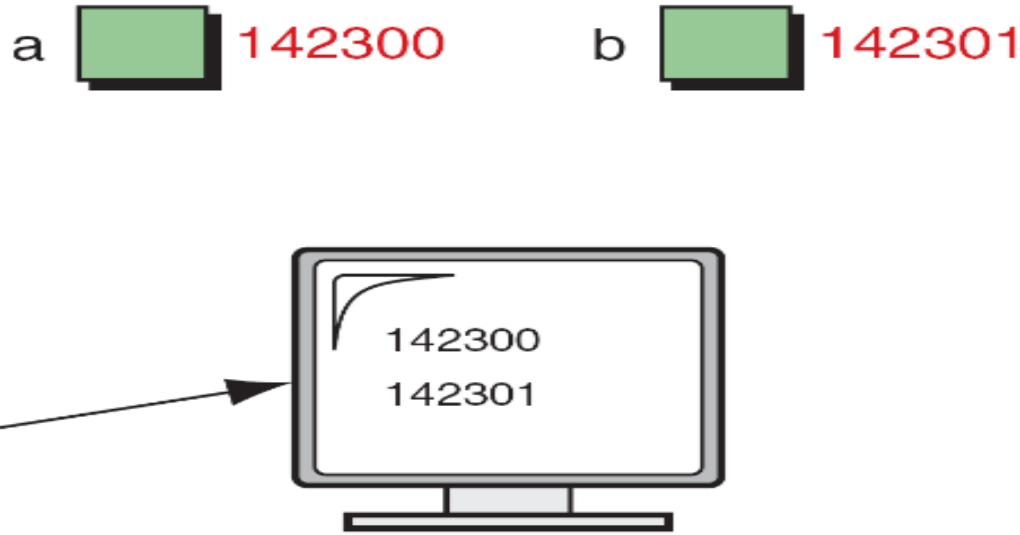
Topics discussed in this section:

Pointer Constants
Pointer Values
Pointer Variables
Accessing Variables Through Pointers
Pointer Declaration and Definition
Declaration versus Redirection
Initialization of Pointer Variables

Note

An address expression, one of the expression types in the unary expression category, consists of an ampersand (&) and a variable name.

```c
// Print character addesses
#include <stdio.h>

int main (void)
{
// Local Declarations
    char a;
    char b;
// Statements
    printf ("%p\n %p\n", &a, &b);
    return 0;
}   // main
```



Note

A variable's address is the first byte occupied by the variable.

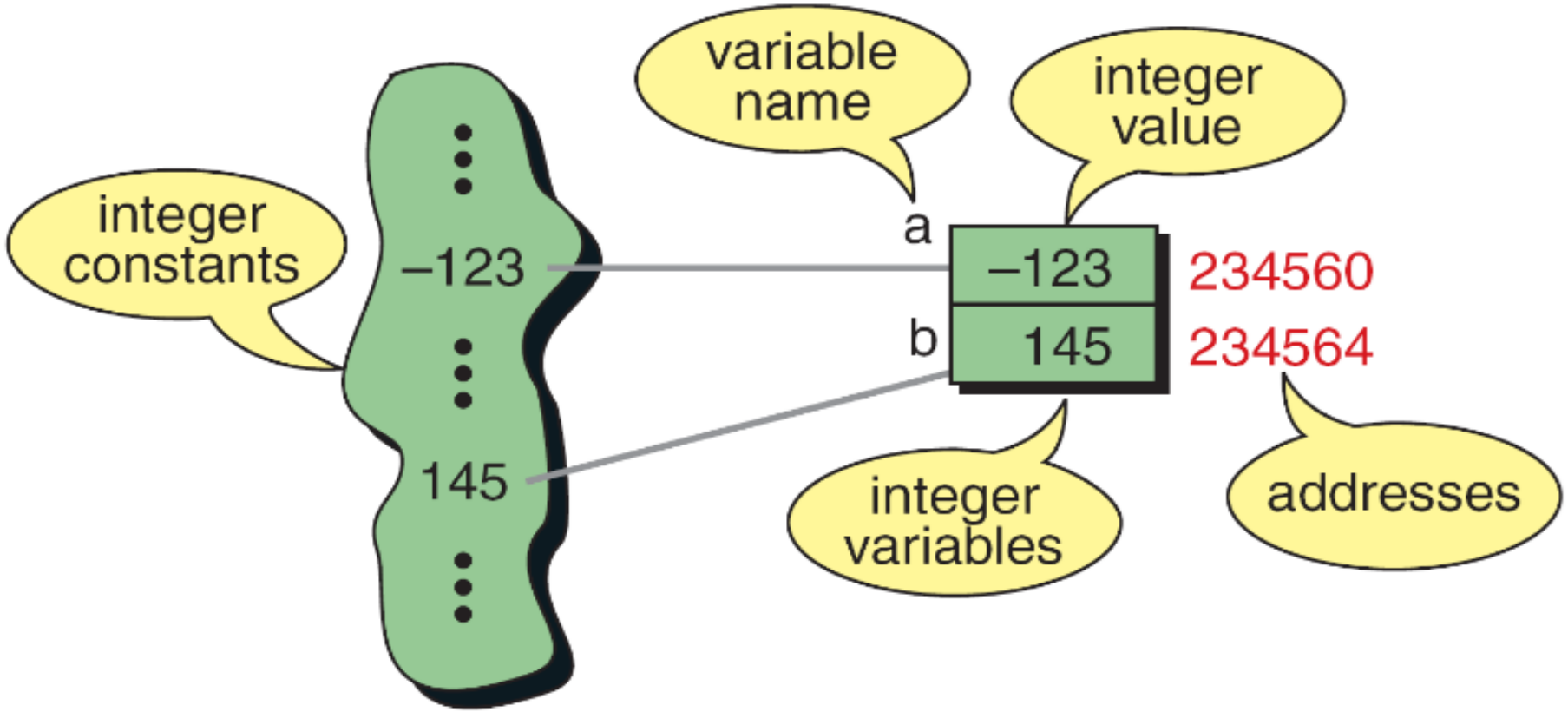
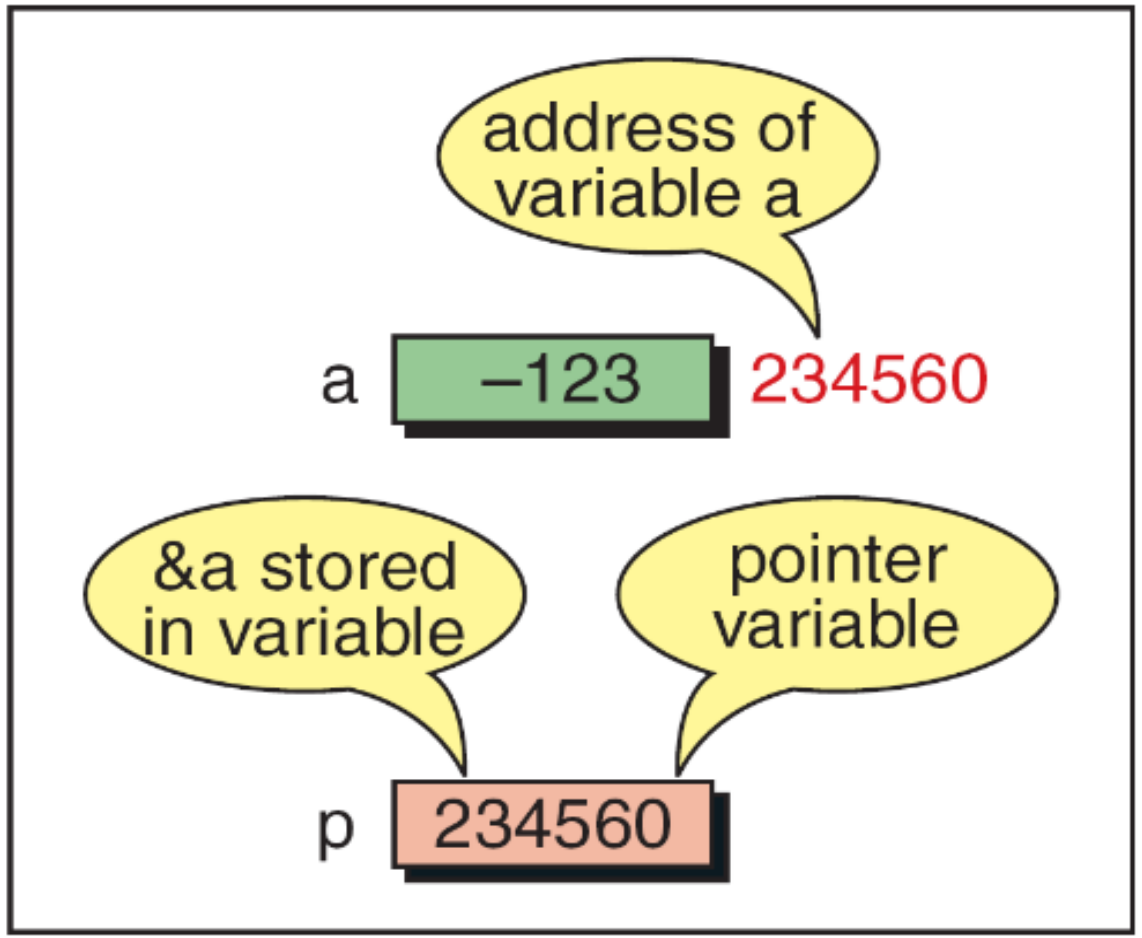
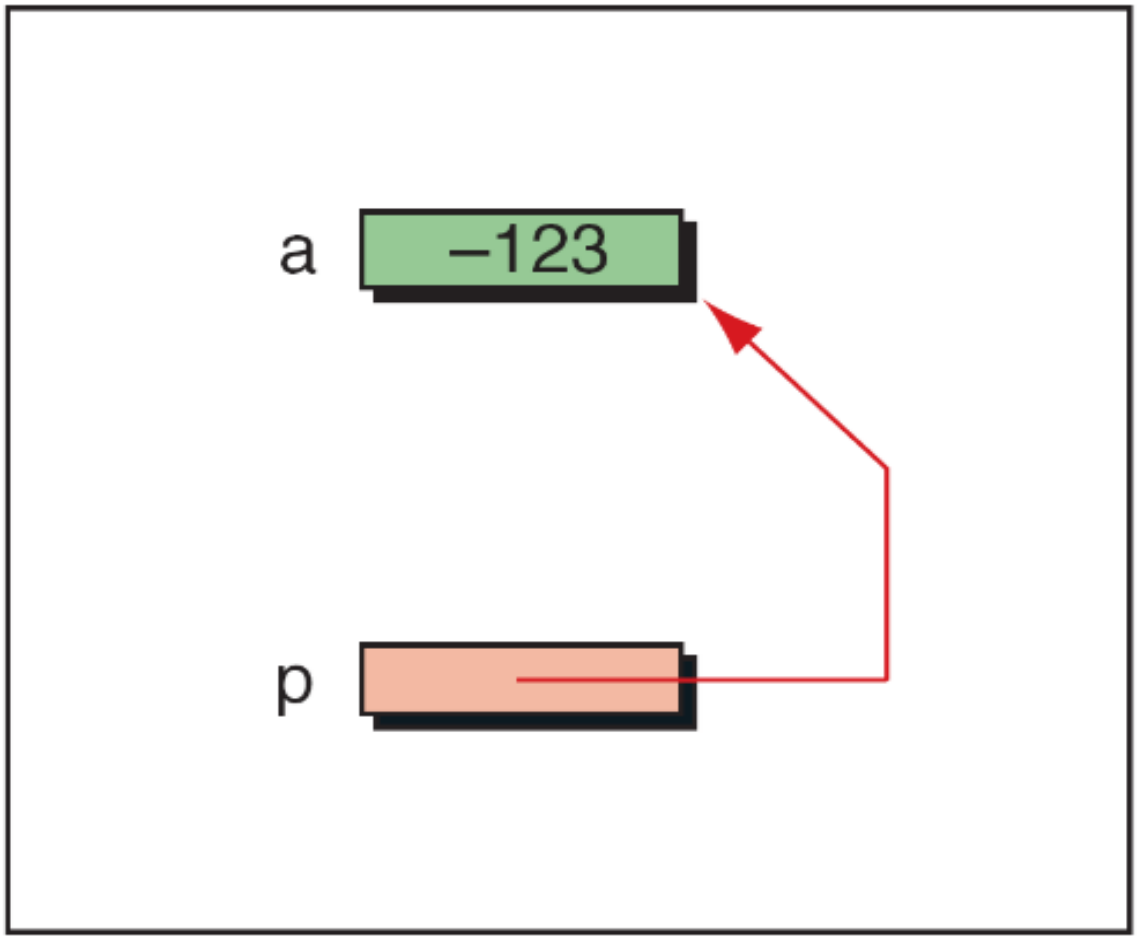


FIGURE 9-5 Integer Constants and Variables



Physical representation



Logical representation

FIGURE 9-6 Pointer Variable

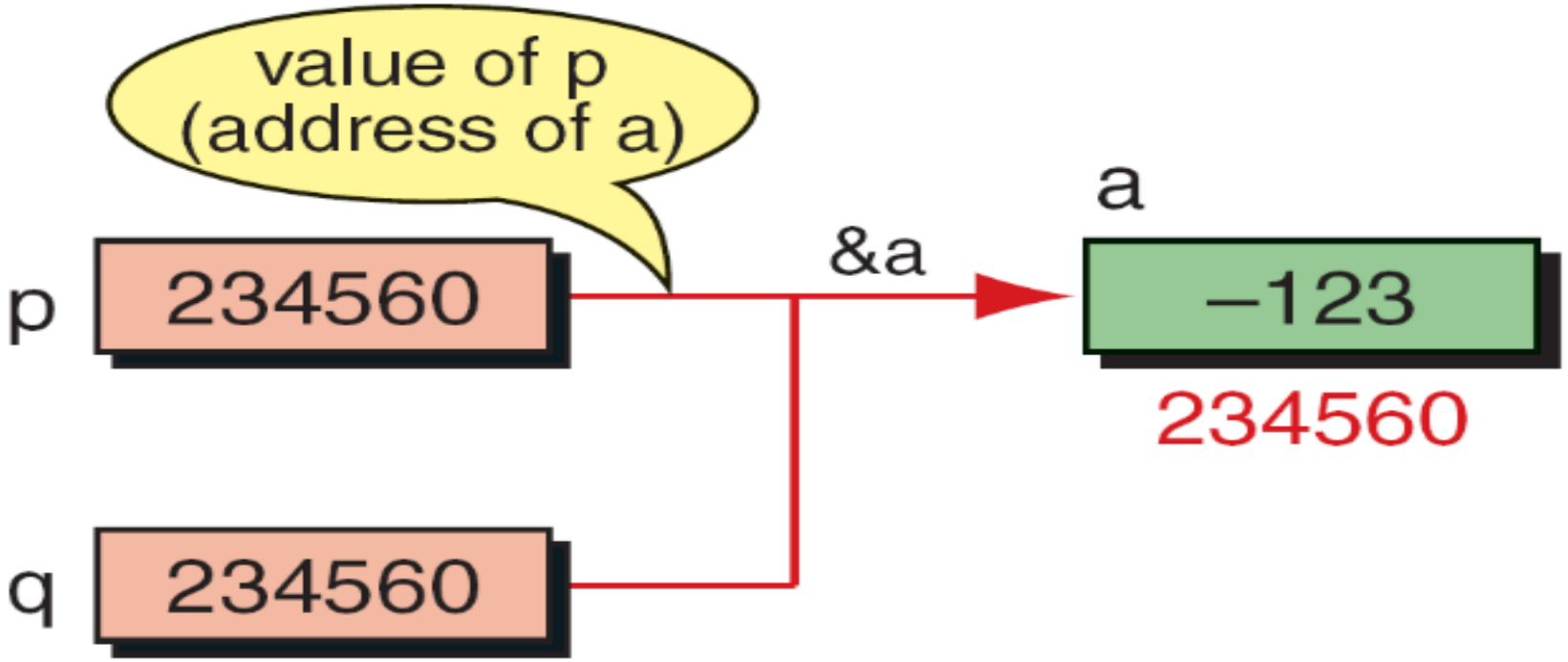


FIGURE 9-7 Multiple

Note

A pointer that points to no variable contains the special null-pointer constant, NULL.

Note

An indirect expression, one of the expression types in the unary expression category, is coded with an asterisk (*) and an identifier.

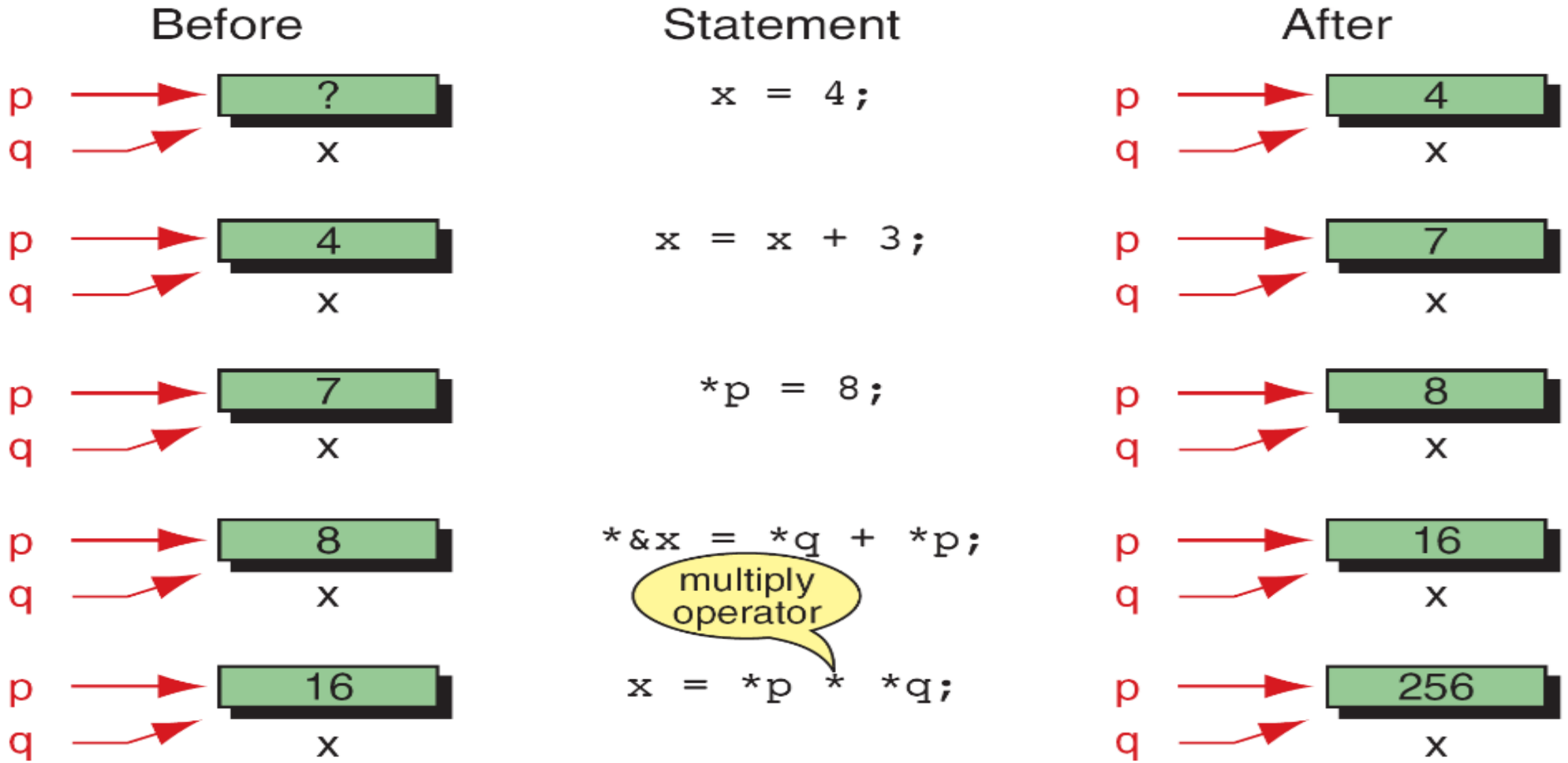


FIGURE 9-8 Accessing Variables Through Pointers

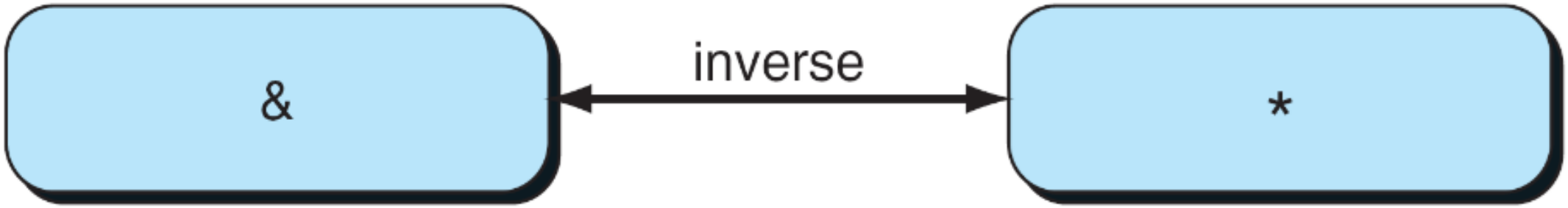


FIGURE 9-9 Address and Indirection Operators

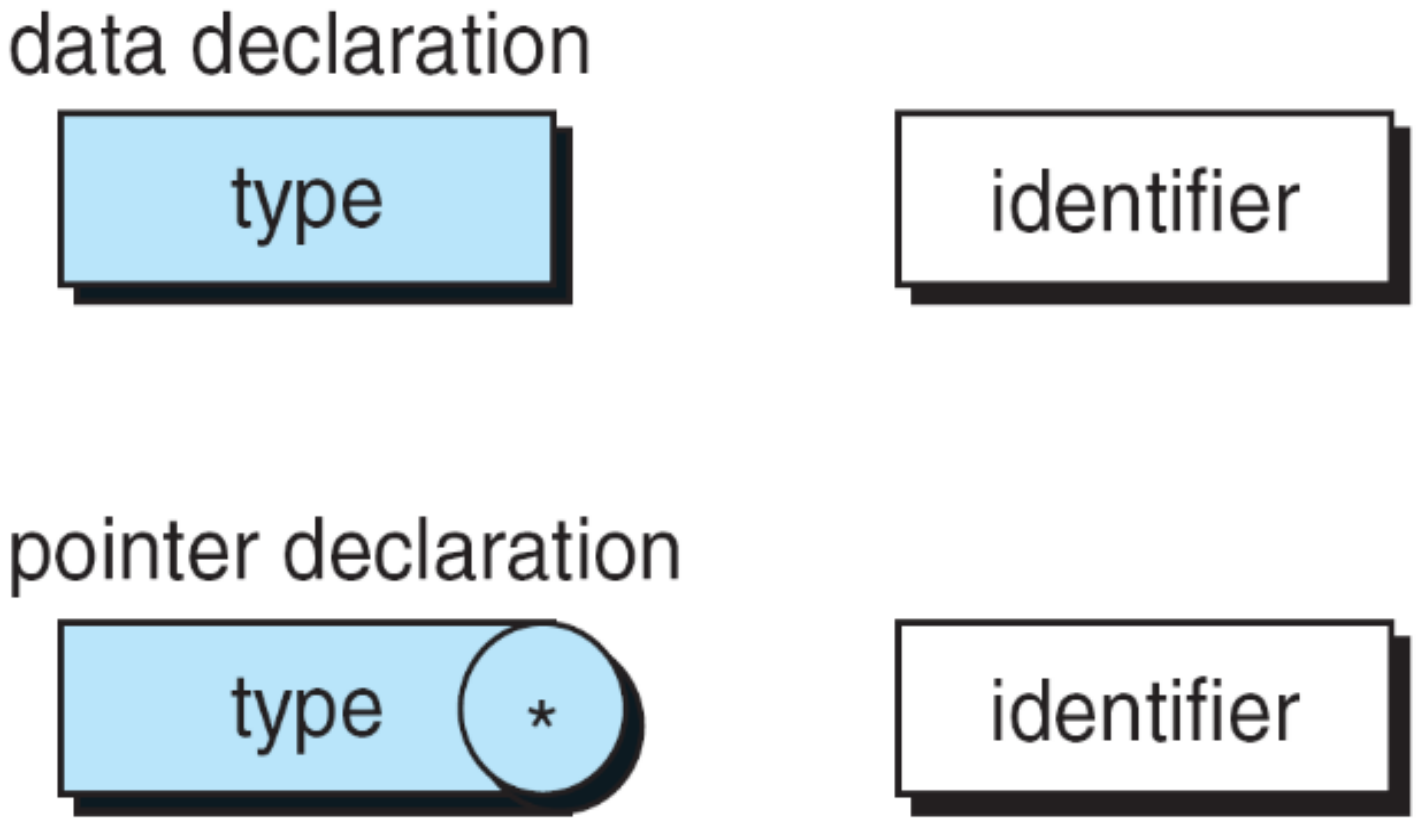


FIGURE 9-10 Pointer Variable Declaration

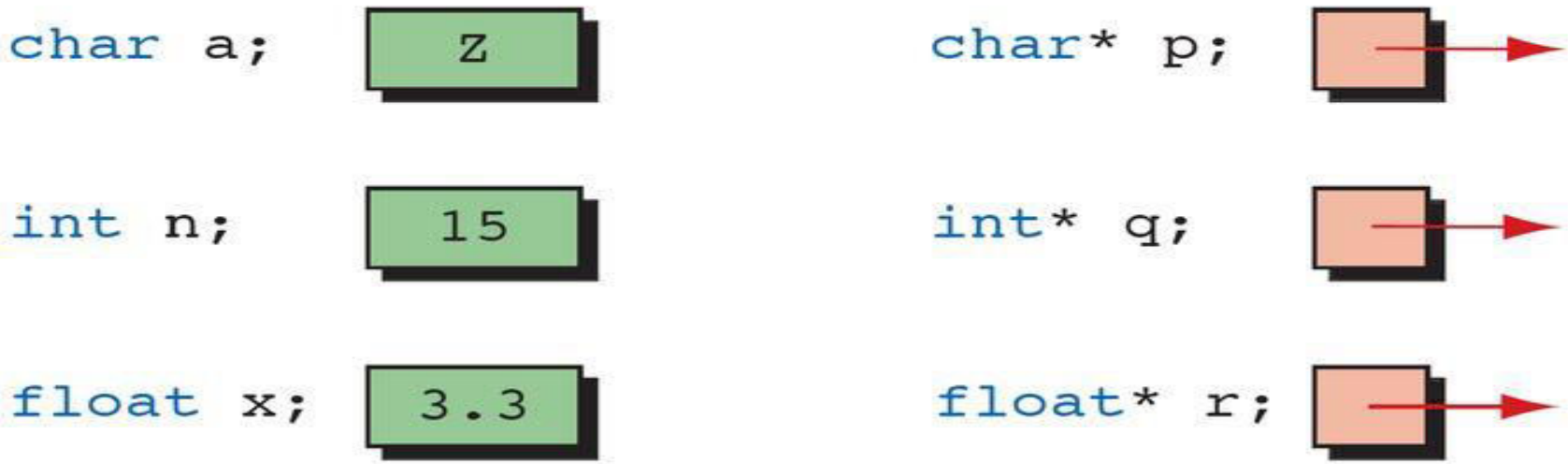
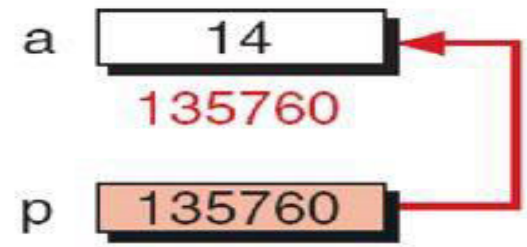


FIGURE 9-11 Declaring Pointer Variables

PROGRAM 9-1 Demonstrate Use of Pointers

```
 1  /* Demonstrate pointer use
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int  a;
11     int* p;
12
13  // Statements
14     a = 14;
15     p = &a;
16
17     printf("%d %p\n", a, &a);
```



PROGRAM 9-1 Demonstrate Use of Pointers

```
18      printf("%p %d %d\n",
19                  p, *p, a);
20
21      return 0;
22  }   // main
```

```
Results:
14 00135760
00135760 14 14
```

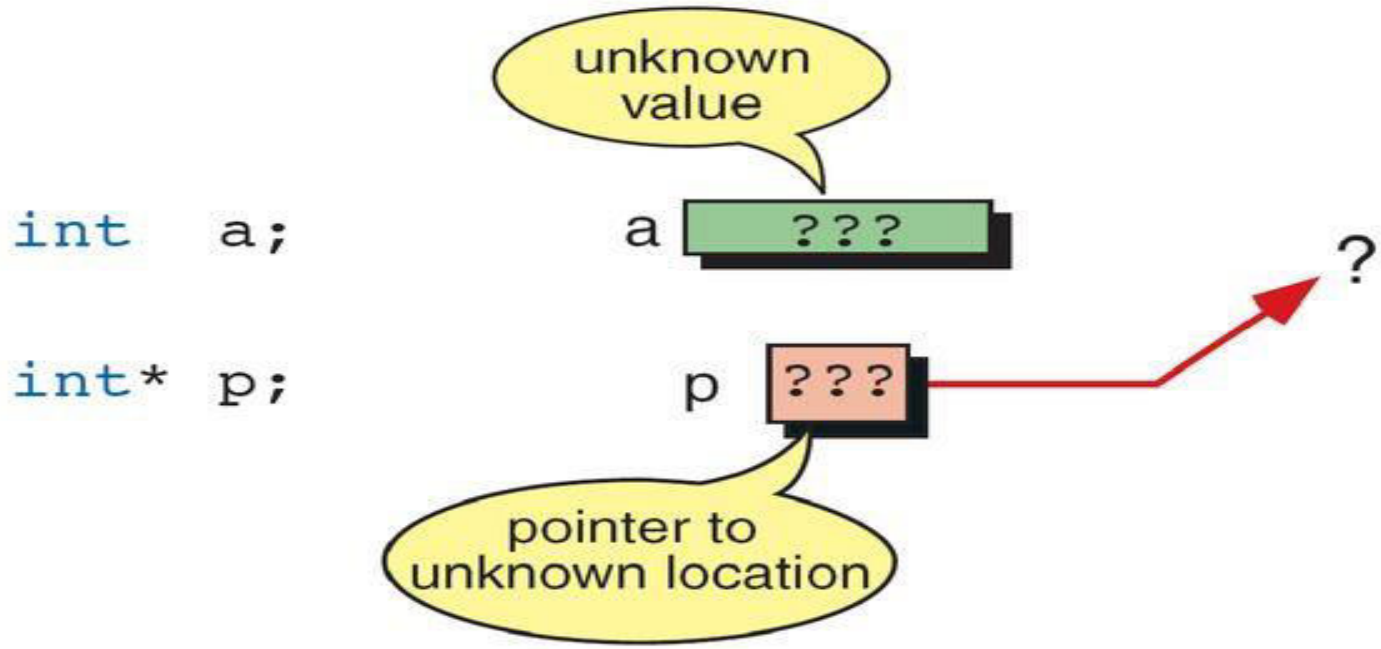


FIGURE 9-12 Uninitialized Pointers

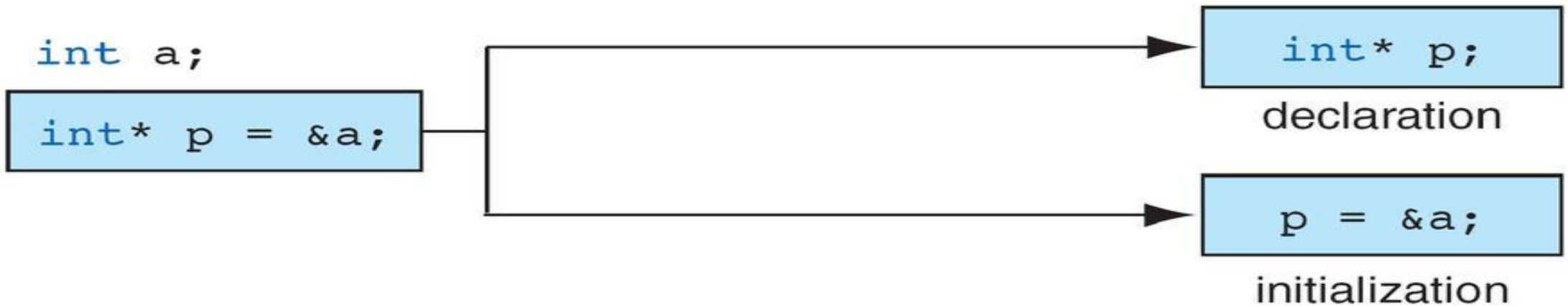
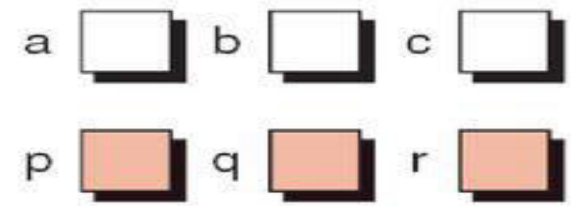
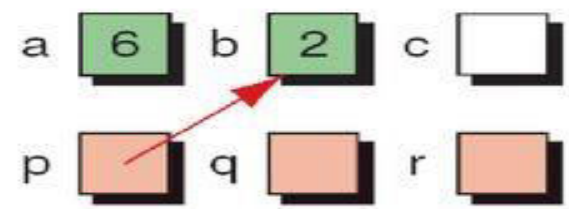


FIGURE 9-13 Initializing Pointer Variables

PROGRAM 9-2 Fun with Pointers

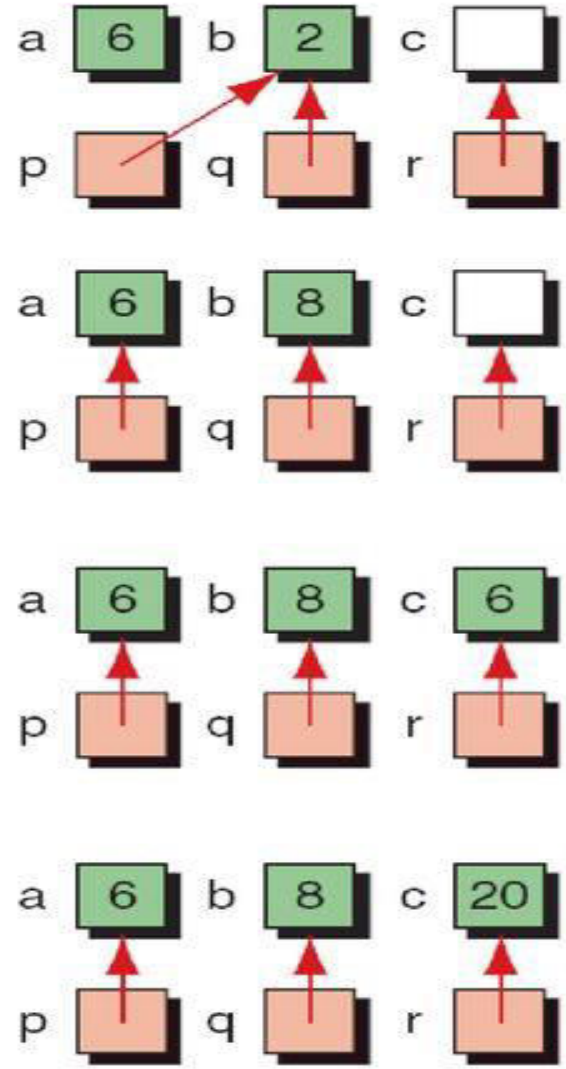
```
 1  /* Fun with pointers
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int   a;
11     int   b;
12     int   c;
13     int*  p;
14     int*  q;
15     int*  r;
16
17  // Statements
18     a = 6;
19     b = 2;
20     p = &b;
21
```





PROGRAM 9-2 Fun with Pointers

```
22      q = p;
23      r = &c;
24
25      p  = &a;
26      *q = 8;
27
28      *r = *p;
29
30      *r = a + *q + *&c;
31
32      printf("%d %d %d \n",
33                 a, b, c);
```



PROGRAM 9-2 Fun with Pointers

```
34      printf("%d %d %d",
35                *p, *q, *r);
36      return 0;
37  }   // main
```

```
Results:
6 8 20
6 8 20
```

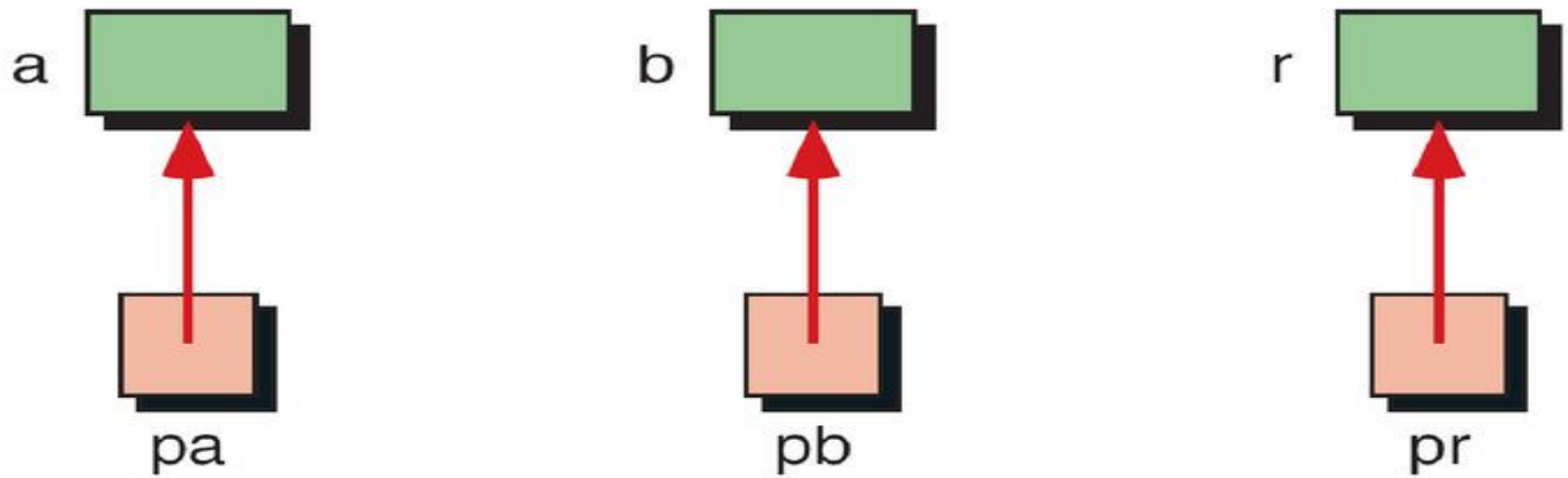


FIGURE 9-14 Add Two Numbers Using Pointers

PROGRAM 9-3 Add Two Numbers Using Pointers

```
 1  /* This program adds two numbers using pointers to
 2     demonstrate the concept of pointers.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     int  a;
12     int  b;
13     int  r;
14     int* pa = &a;
15     int* pb = &b;
16     int* pr = &r;
17
```

PROGRAM 9-3 Add Two Numbers Using Pointers

```
18  // Statements
19     printf("Enter the first number : ");
20     scanf ("%d", pa);
21     printf("Enter the second number: ");
22     scanf ("%d", pb);
23     *pr = *pa + *pb;
24     printf("\n%d + %d is %d", *pa, *pb, *pr);
25     return 0;
26  }  // main
```

```
Results:
   Enter the first number : 15
   Enter the second number: 51

   15 + 51 is 66
```

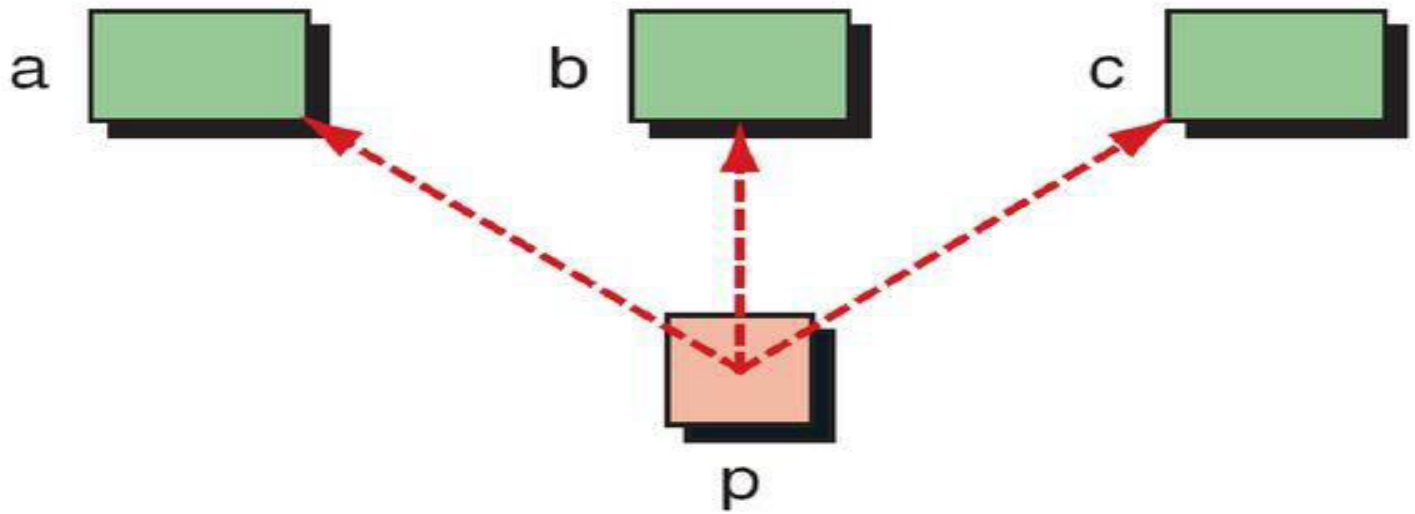


FIGURE 9-15 Demonstrate Pointer Flexibility

PROGRAM 9-4 Using One Pointer for Many Variables

```
 1  /* This program shows how the same pointer can point to
 2     different data variables in different statements.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     int  a;
12     int  b;
13     int  c;
14     int* p;
15
16  // Statements
17     printf("Enter three numbers and key return: ");
18     scanf ("%d %d %d", &a, &b, &c);
```

PROGRAM 9-4 Using One Pointer for Many Variables

```
19      p = &a;
20      printf("%3d\n", *p);
21      p = &b;
22      printf("%3d\n", *p);
23      p = &c;
24      printf("%3d\n", *p);
25      return 0;
26  }   // main
```

```
Results:
Enter three numbers and key return: 10 20 30
 10
 20
 30
```

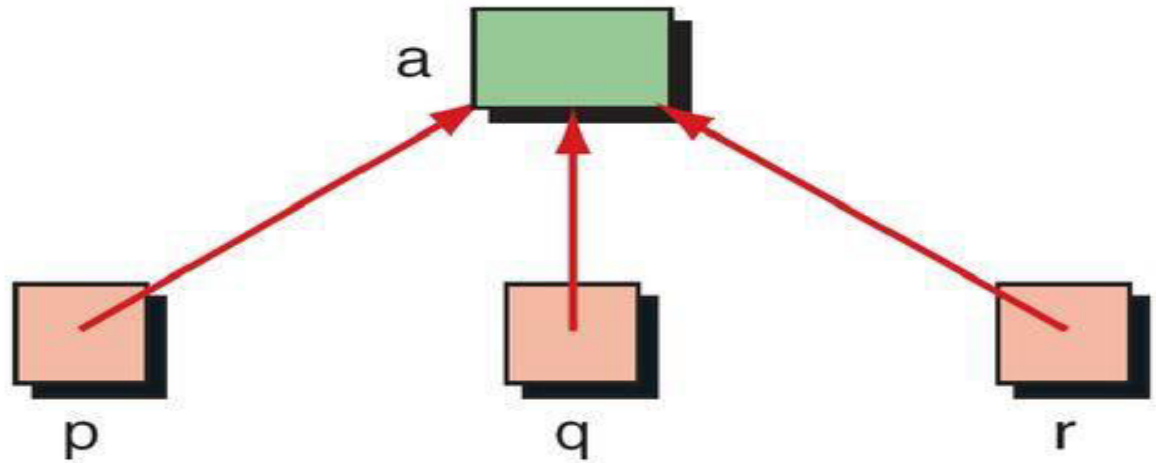


FIGURE 9-16 One Variable with Many Pointers

PROGRAM 9-5 Using A Variable with Many Pointers

```
 1  /* This program shows how we can use different pointers
 2     to point to the same data variable.
 3        Written by:
 4        Date:
 5  */
 6  #include <stdio.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     int  a;
12     int* p = &a;
13     int* q = &a;
14     int* r = &a;
15
16  // Statements
17     printf("Enter a number: ");
18     scanf ("%d", &a);
```

PROGRAM 9-5 Using A Variable with Many Pointers

```
19      printf("%d\n", *p);
20      printf("%d\n", *q);
21      printf("%d\n", *r);
22
23      return 0;
24  }   // main
```

```
Results:
Enter a number: 15
15
15
15
```

9-2 Pointers for Inter-function Communication

One of the most useful applications of pointers is in functions. When we discussed functions in Chapter 4, we saw that C uses the pass-by-value for downward communication. For upward communication, we normally pass an address. In this section, we fully develop the bi-directional communication.

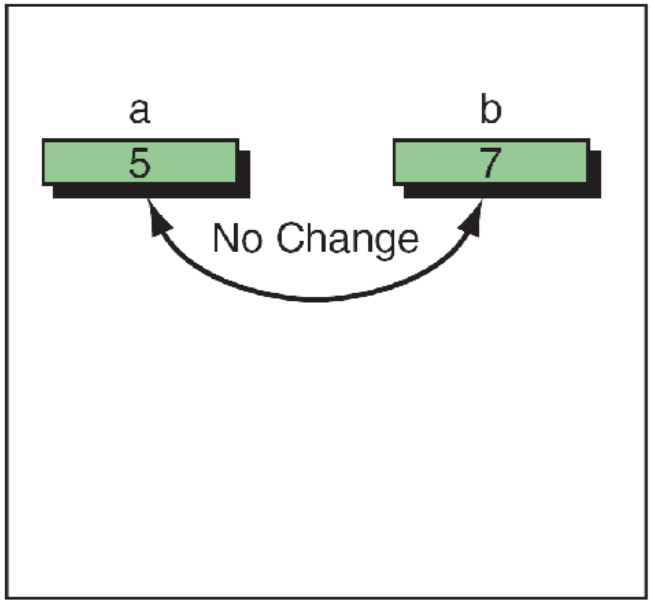
Topics discussed in this section:

Passing Addresses
Functions Returning Pointers

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called call-by-reference (or by address or by location).
- Normally, arguments are passed to a function by value.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.

```c
// Function Declarations
void exchange (int x, int y);

int main (void)
{
   int a = 5;
   int b = 7;
   exchange (a, b);
   printf("%d %d\n", a, b);
   return 0;
} // main
```




```c
void exchange (int x, int y)
{
   int temp;

   temp = x;
   x    = y;
   y    = temp;
   return;
} // exchange
```

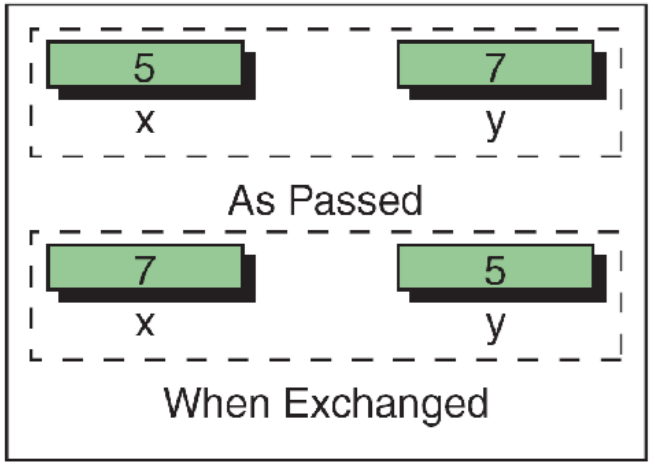


FIGURE 9-17 An Unworkable Exchange

```c
// Function Declaration
void exchange (int*, int*);

int main (void)
{
  int a = 5;
  int b = 7;

  exchange (&a, &b);
  printf("%d %d\n", a, b);
  return 0;
} // main
```

```c
void exchange (int* px, int* py)
{
  int temp;

  temp = *px;
  *px  = *py;
  *py  = temp;
  return;
} // exchange
```

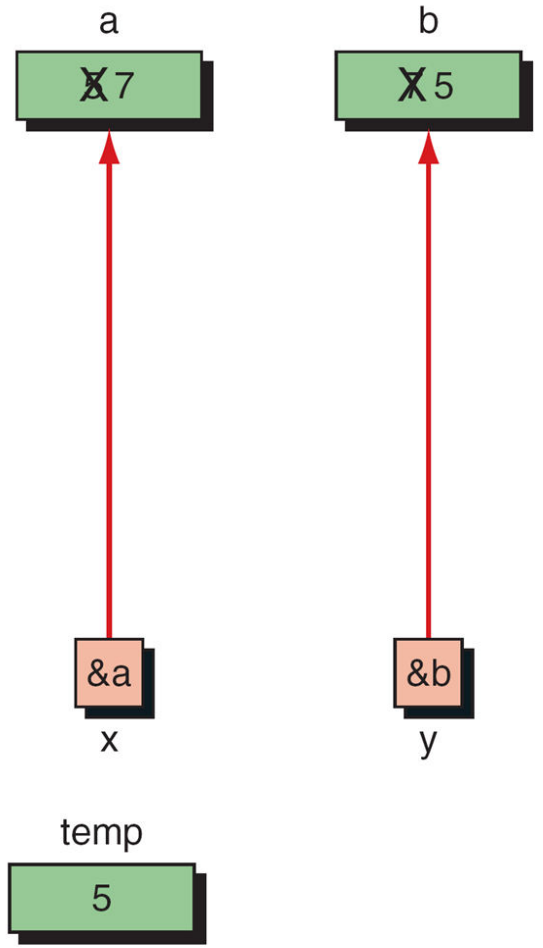


FIGURE 9-18 Exchange Using Pointers

Note

Every time we want a called function to have access to a variable in the calling function, we pass the address of that variable to the called function and use the indirection operator to access it.

```c
// Prototype Declarations
int* smaller (int* p1, int* p2);

int main (void)
  …
  int  a;
  int  b;
  int* p;
  …
  scanf ( "%d %d", &a, &b );
  p = smaller (&a, &b);
  …
```

```c
int* smaller (int* px, int* py)
{
  return (*px < *py ? px : py);
} // smaller
```

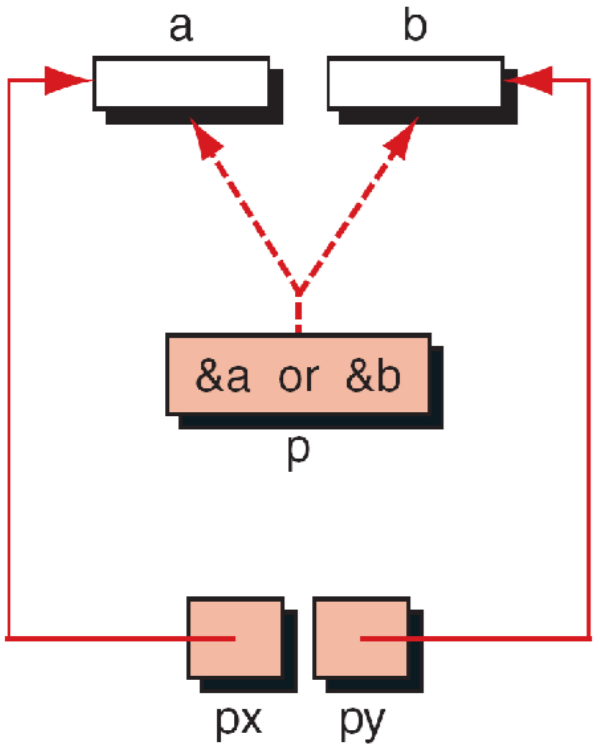


FIGURE 9-19 Functions Returning Pointers

Note

It is a serious error to return a pointer to a local variable.

9-3 Pointers to Pointers

So far, all our pointers have been pointing directly to data. It is possible—and with advanced data structures often necessary—to use pointers that point to other pointers. For example, we can have a pointer pointing to a pointer to an integer.

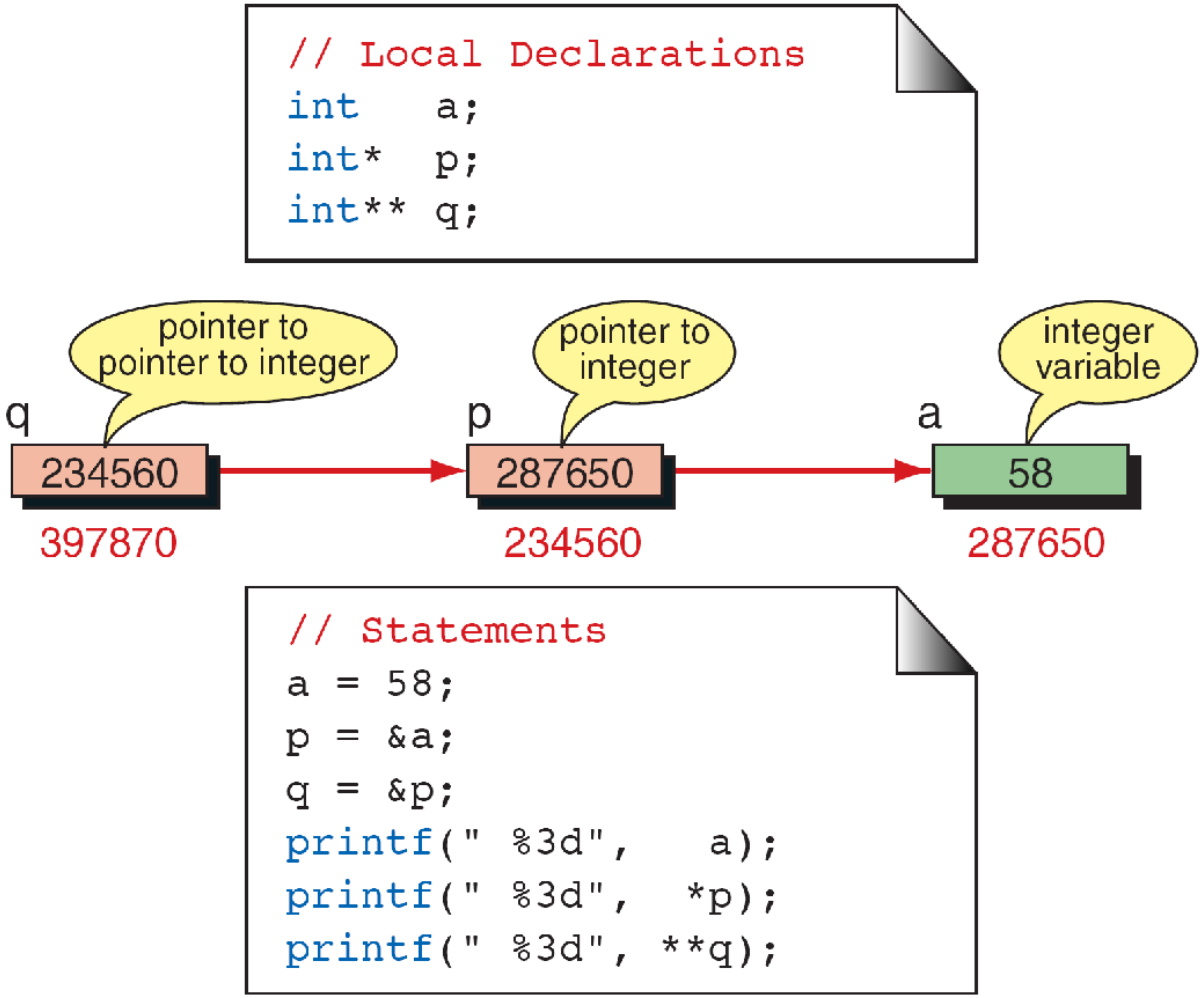


FIGURE 9-20 Pointers to Pointers

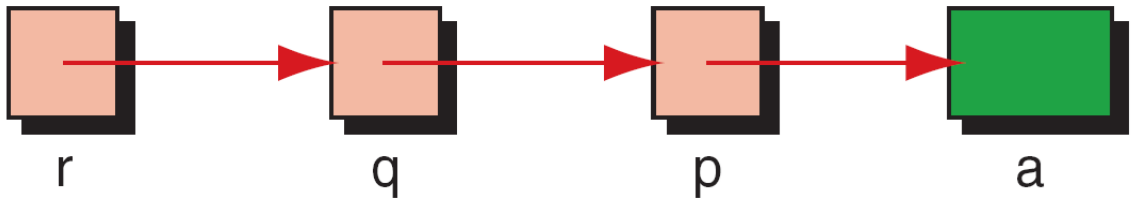


FIGURE 9-21 Using Pointers to Pointers

PROGRAM 9-6 Using pointers to pointers

```
1  /* Show how pointers to pointers can be used by different
2     scanf functions to read data to the same variable.
3        Written by:
4        Date:
5  */
6  #include <stdio.h>
7
8  int main (void)
9  {
10 // Local Declarations
11    int    a;
12    int*   p;
13    int**  q;
14    int*** r;
15
16 // Statements
17    p = &a;
```

PROGRAM 9-6 Using pointers to pointers

```
18      q = &p;
19      r = &q;
20
21      printf("Enter a number: ");
22      scanf ("%d", &a);                          // Using a
23      printf("The number is : %d\n", a);
24
25      printf("\nEnter a number: ");
26      scanf ("%d", p);                           // Using p
27      printf("The number is : %d\n", a);
28
29      printf("\nEnter a number: ");
30      scanf ("%d", *q);                          // Using q
31      printf("The number is : %d\n", a);
32
33      printf("\nEnter a number: ");
34      scanf ("%d", **r);                         // Using r
35      printf("The number is : %d\n", a);
36
37      return 0;
38  }   // main
```

PROGRAM 9-6 Using pointers to pointers

```
Results:
Enter a number: 1
The number is : 1

Enter a number: 2
The number is : 2

Enter a number: 3
The number is : 3

Enter a number: 4
The number is : 4
```

10-1 Arrays and Pointers

The name of an array is a pointer constant to the first element. Because the array's name is a pointer constant, its value cannot be changed. Since the array name is a pointer constant to the first element, the address of the first element and the name of the array both represent the same location in memory.

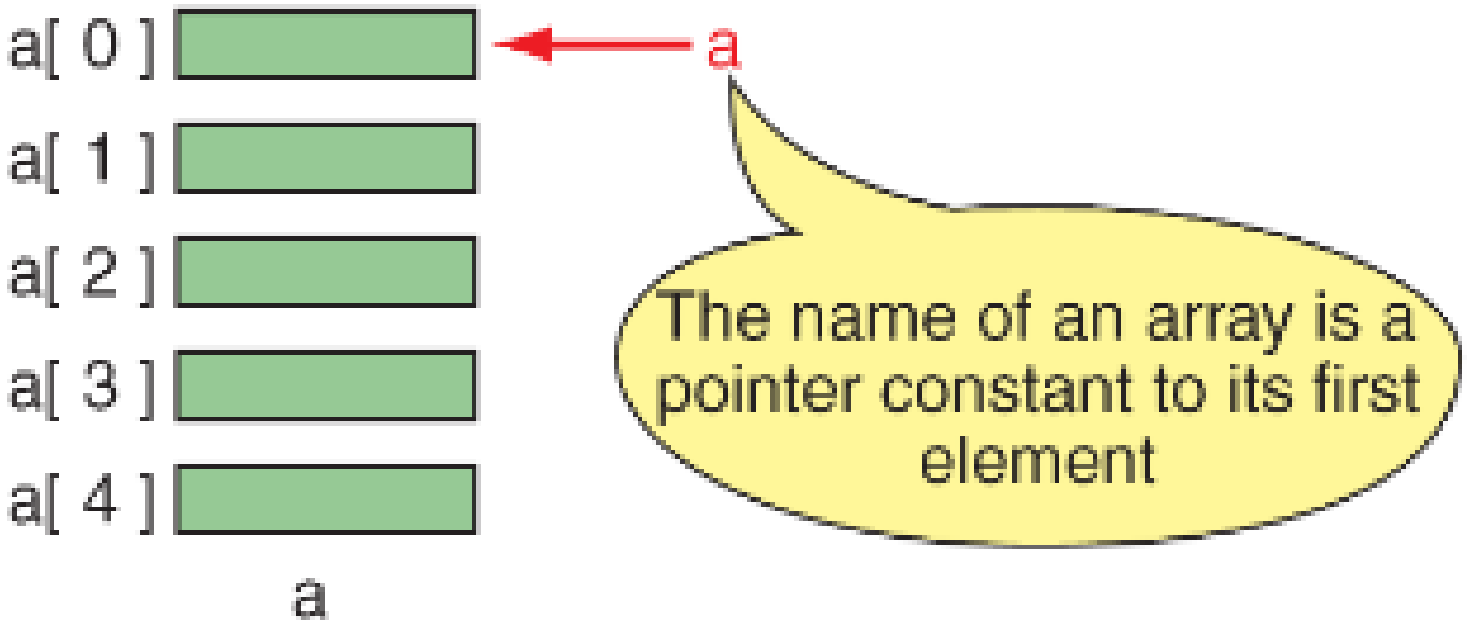


FIGURE 10-1 Pointers to Arrays

Note

same

a ⟷ &a[0]

***a* is a pointer only to the first element—not the whole array.**

Note

The name of an array is a pointer constant;
it cannot be used as an *lvalue*.

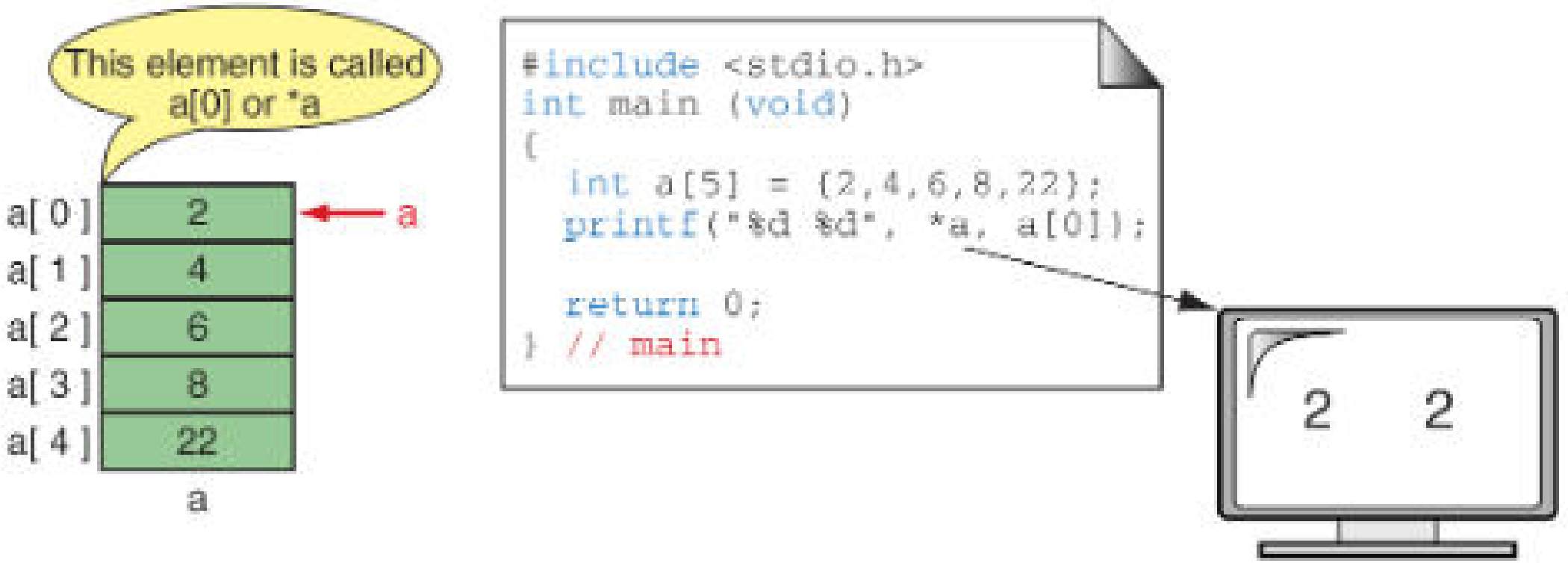


FIGURE 10-2 Dereference of Array Name

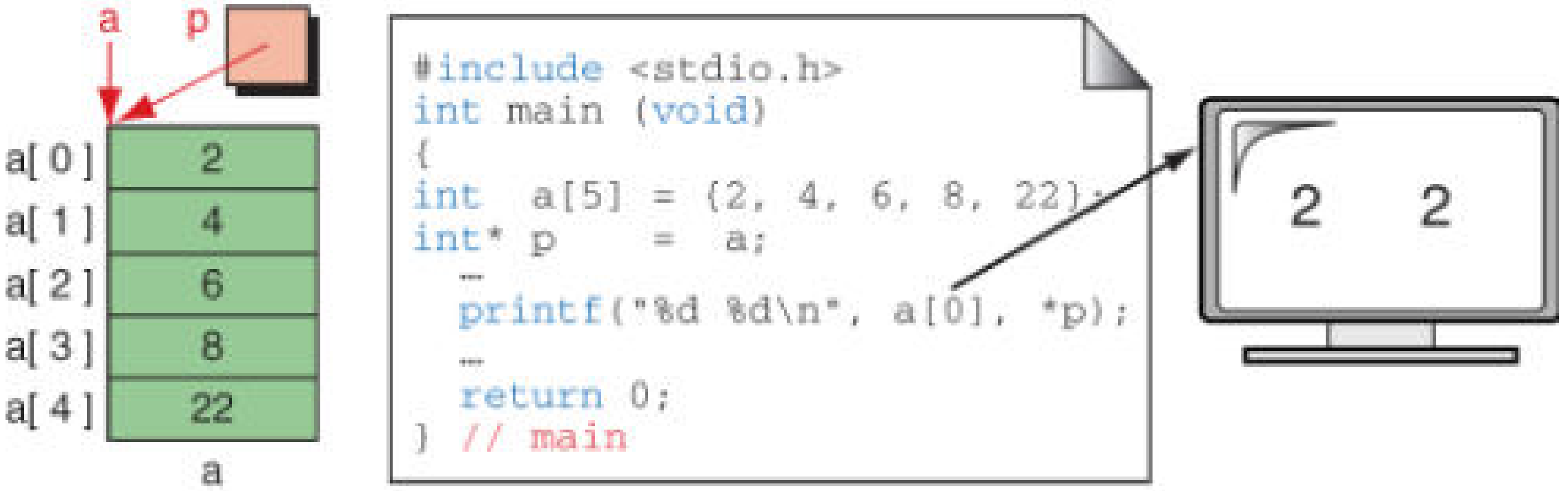


FIGURE 10-3 Array Names as Pointers

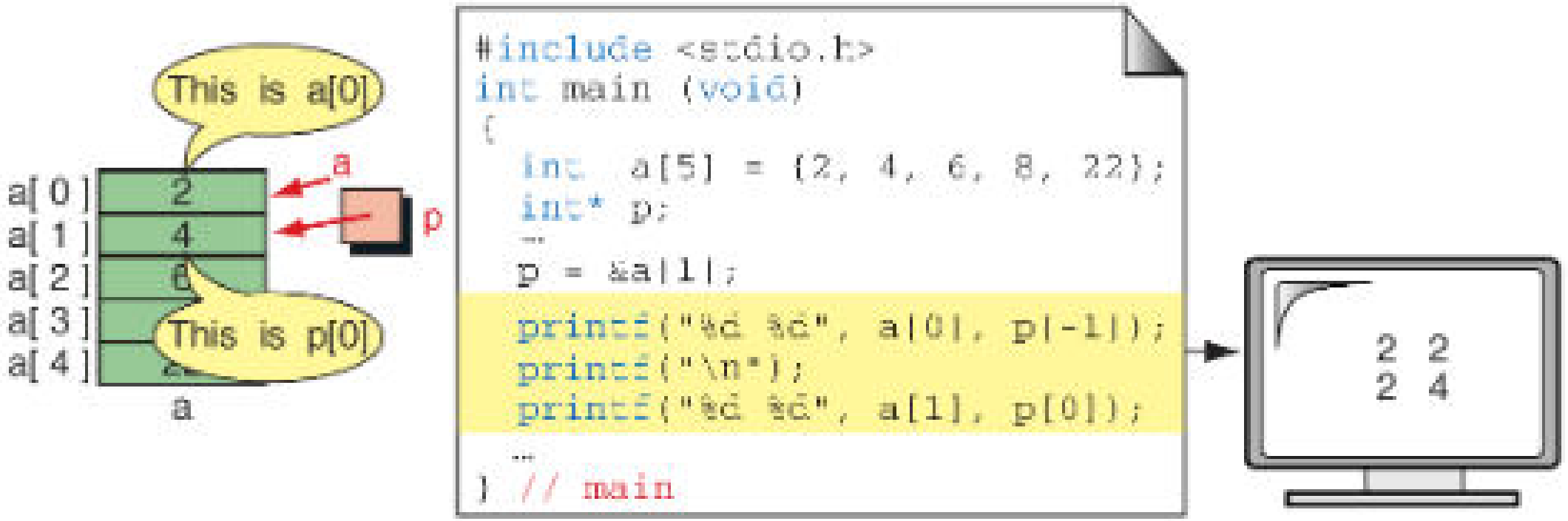


FIGURE 10-4 Multiple Array Pointers

Note

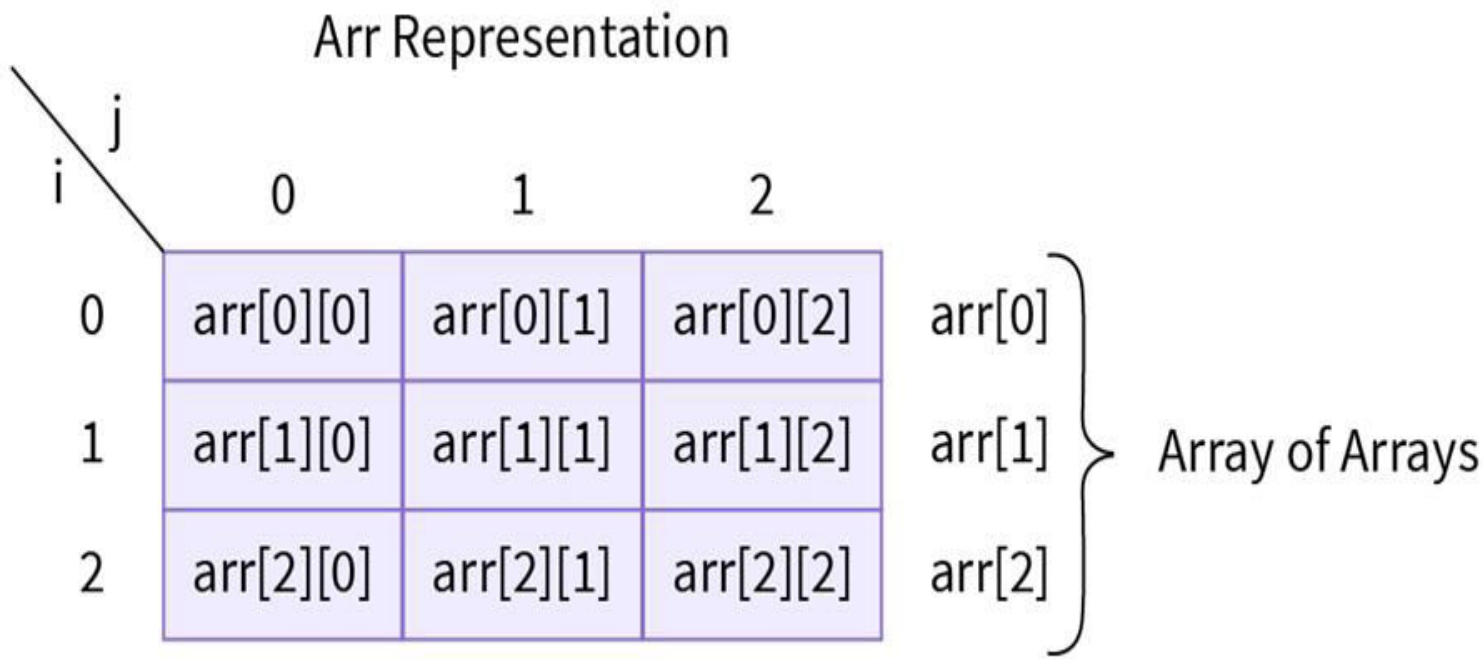
To access an array, any pointer to the first element can be used instead of the name of the array.

```
#include <stdio.h>
int main( )
{
   /*Pointer variable*/
   int *p;

   /*Array declaration*/
   int val[7] = { 11, 22, 33, 44, 55, 66, 77 } ;

   p = &val[0];   // * p = val;

for ( int i = 0 ; i<7 ; i++ )
   {
      printf("val[%d]: value is %d and address is %p\n", i, *p, p);
      p++;
   }
   return 0;
}
```

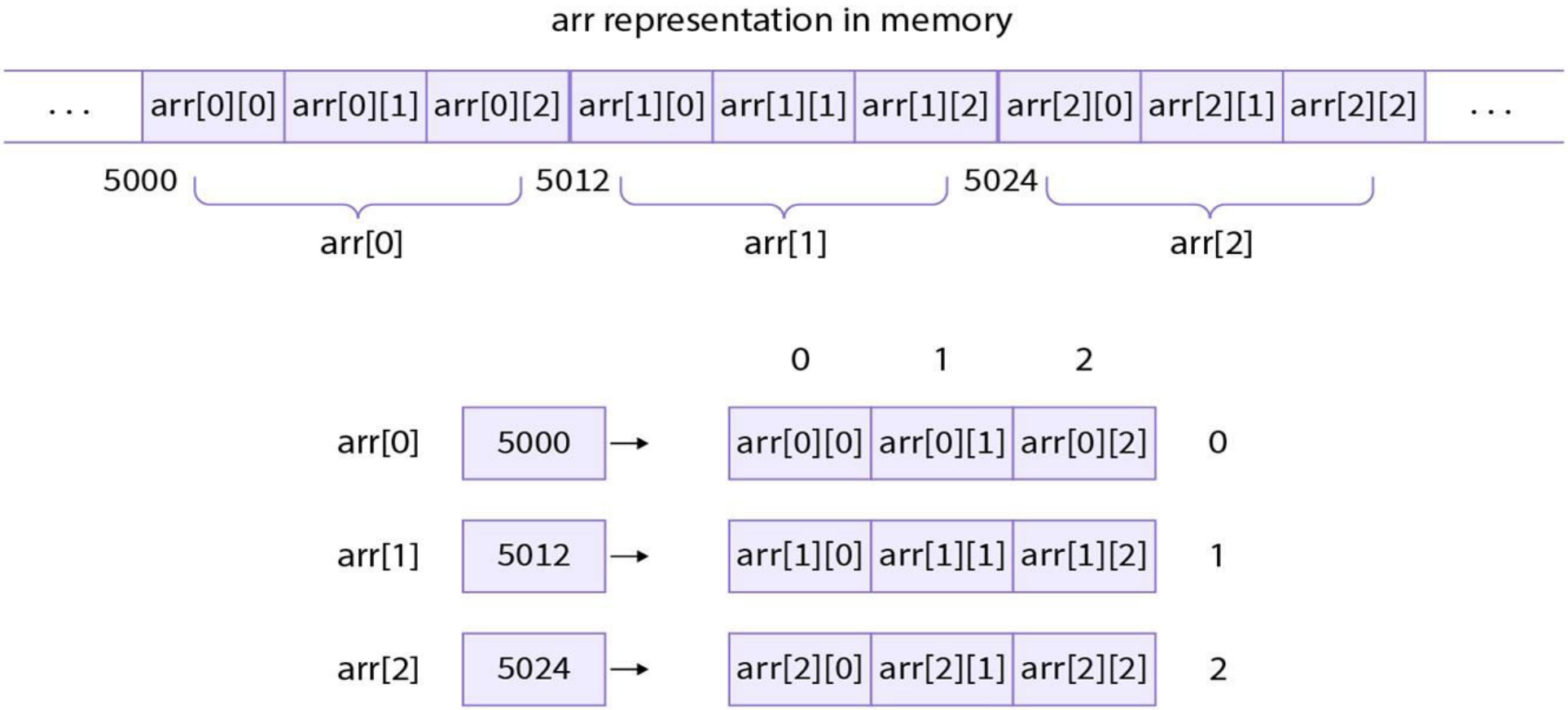
```c
#include <stdio.h>
int main() {

  int i, x[6], sum = 0;

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {
  // Equivalent to scanf("%d", &x[i]);
      scanf("%d", x+i);

  // Equivalent to sum += x[i]
      sum += *(x+i);
  }

  printf("Sum = %d", sum);

  return 0;
}
```

```c
#include <stdio.h>
int main() {

   int x[5] = {1, 2, 3, 4, 5};
   int* ptr;

   // ptr is assigned the address of the third element
   ptr = &x[2];

   printf("*ptr = %d \n", *ptr);   // 3
   printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
   printf("*(ptr-1) = %d", *(ptr-1));  // 2

   return 0;
}
```

```c
#include <stdio.h>
int sum( int *a, int l) {
int i, s = 0;
for(i = 0; i < l; i++)
s += a[i];
return s;
}

int main()
{
int nums[5] = {1, 2, 3, 4, 5};
int size = sizeof(nums)/sizeof(nums[0]);
printf("Sum is %d\n", sum(nums, size));  //15
return 0;
}
```



SCALER Topics



SCALER Topics

```
*(*(arr + i) + j)
```

Note : *(*(arr + i) + j) represents the element of an array arr at the index value of i^{th} row and j^{th} column; it is equivalent to the regular representation of 2-D array elements as arr[i][j].

```
#include <stdio.h>

int main()
{
        int arr[3][3] = {{2, 4, 6},
                         {0, 1, 0},
                         {3, 5, 7}};
        int i, j;

printf("Addresses : \n");
        for(i = 0; i < 3; i++)
        {
                for(j = 0; j < 3; j++)
                {
                        printf("%u[%d%d] ", (*(arr + i) +
j), i, j);
                }
                printf("\n");
```

```
printf("Values : \n");
        for(i = 0; i < 3; i++)
        {
                for(j = 0; j < 3; j++)
                {
                        printf("%d[%d%d] ", *(*(arr + i) + j), i,
j);
                }
                printf("\n");
        }

        return 0;
}
```

```
[Success] Your code was executed successfully

Addresses :
4201367232[00] 4201367236[01] 4201367240[02]
4201367244[10] 4201367248[11] 4201367252[12]
4201367256[20] 4201367260[21] 4201367264[22]
Values :
2[00] 4[01] 6[02]
0[10] 1[11] 0[12]
3[20] 5[21] 7[22]
```

```c
#include <stdio.h>

int main(void) {

  // string variable
  char str[6] = "Hello";

  // pointer variable
  char *ptr = str;

  // print the string
  while(*ptr != '\0') {
    printf("%c", *ptr);

    // move the ptr pointer to the next memory location
    ptr++;
  }

  return 0;
}
```