Introduction to Computers & Programming

Comp 1330/ First Semester 2024/2025

Instructor: Saif Harbia

Faculty of Engineering and Technology
Department of Computer Science

Chapter 08

Strings

Chapter Objectives:

1. Understand how a string constant is stored in an array of characters.

2. Learn about the placeholder **%s** and how it is used in **printf** and **scanf** operations.

3. Learn some of the operations that can be performed on strings. (1)

4. Understand how C compares two strings to determine their relative order.

5. Learn some of the operations that can be performed on individual characters using functions from the library **ctype**.

6. How to write your own functions that perform some of the basic operations of a text editor program

STRINGS

- **A string: is a grouping of characters, a data structure.**
- **C implements the string data structure using arrays of type char** (1)
- **Strings are important in computer science** (2)
- **Strings play an important role in science as well** (3)
- **We need strings as the vehicle for communication between the computer system and the human user.**

8.1 STRING BASICS

- We have already used **string constants** in our earlier work with **printf** and **scanf (1)**

```
printf("Average = %.2f", avg);
```

- The first argument is the string constant **"Average = %.2f",** a string of 14 characters
- Notice that the blanks in the string are characters.
- A string constant can be associated with a symbolic name using the **#define** directive (2)

```
#define ERROR "*****Error - "
#define STD_NAME "Student Name"
```

DECLARING AND INITIALIZING STRING VARIABLES

- A string in C is implemented as an array, so declaring a string variable is the same as declaring an array of type **char** .

```
char string_var[30];
```

- The variable **string_var** will hold strings from 0 to 29 characters long (1)

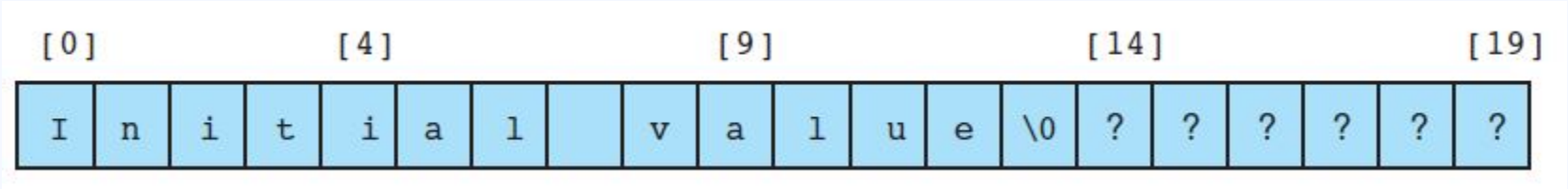
- C permits initialization of string variables using a string constant

```
char str[20] = "Initial value";
```

DECLARING AND INITIALIZING STRING VARIABLES

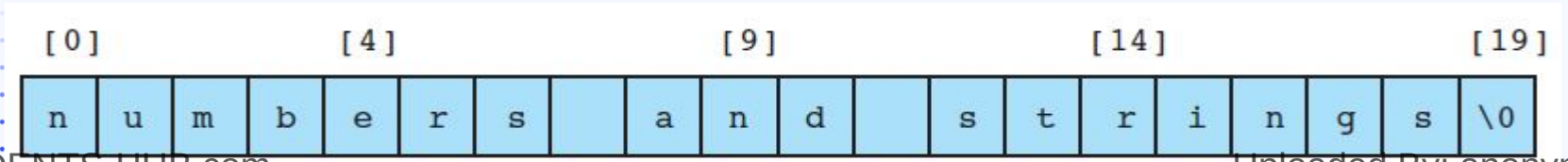
```
char str[20] = "Initial value";
```

- **str** in memory after this declaration with initialization.



- **str[13]** contains the character **'\0'** , the null character that marks the end of a string. (1)

```
char str[20] = "Numbers and strings"; (2)
```



ARRAYS OF STRINGS

- Because one string is an array of characters, an array of strings is a **two-dimensional array of characters** in which each row is one string.

```
#define NUM_PEOPLE 30
#define NAME_LEN 25
. . .
char names[NUM_PEOPLE][NAME_LEN]; (1)
```

```
char month[12][10] = {"January", "February", "March", "April",
"May", "June", "July", "August", "September", "October", "November",
"December"};
```

INPUT/OUTPUT WITH PRINTF AND SCANF

- Use the placeholder **%s** in the format string with **printf** and **scanf** to handle string argument.

```
printf("Topic: %s\n", string_var);
```

- The **printf** function, depends on finding a *null character* in the character array to mark the end of the string. (1)
- When we write our own string-building functions, we must be sure that a null character is inserted at the end of every string. (2)
- This inclusion of the null character is automatic for constant strings.
- The %s placeholder in a **printf** format string can be used with a minimum field width (3)

```
printf ("%8s%3s***\n", "Short", "Strings");
```

INPUT/OUTPUT WITH PRINTF AND SCANF

➢ Placing **a minus sign** prefix on a placeholder's field width causes left justification of the value displayed.

```
printf("%-20s\n", "Name");   (1)
```

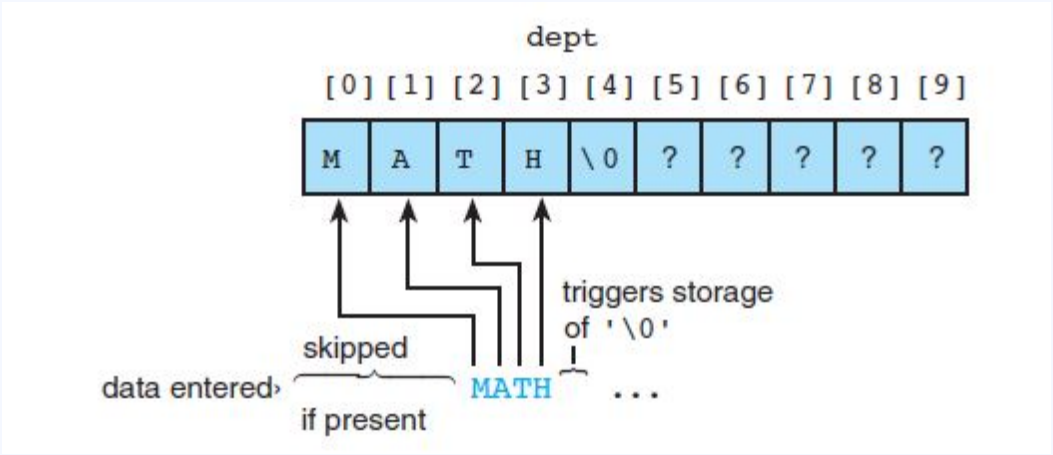
- **Scanf Function:**

- **Remember:** when we call **scanf** with a string variable as an argument, we must remember that array output arguments are always passed to functions by sending the address of the initial array element. (2)

- **Figure 8.2 p. 457**

INPUT/OUTPUT WITH PRINTF AND SCANF

Figure 8.3



- **scanf** skips leading whitespace characters such as blanks, newlines, and tabs.(1)
- Starting with the first nonwhitespace character, **scanf** copies the characters it encounters into successive memory cells of its character array argument.
- When it comes across a whitespace character, scanning stops, **scanf** places the *null character* at the end of the string in its array argument.

INPUT/OUTPUT WITH PRINTF AND SCANF

- For example, the data could have been entered one value per line with extra whitespace.

```
> MATH
1270
TR
1800
```

- or two values per line:

```
> MATH 1270
TR 1800
```

- But: (1)

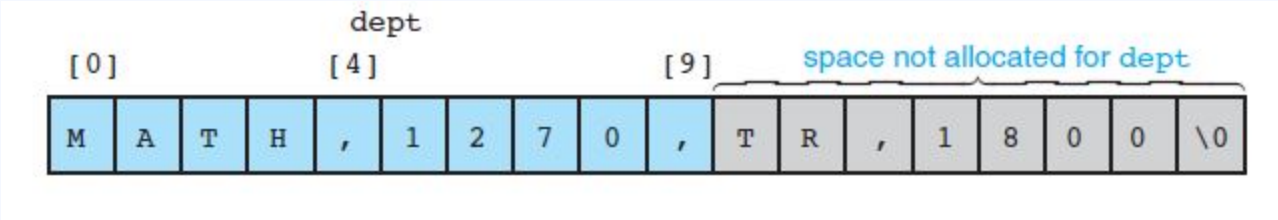
```
> MATH1270 TR 1800    (2)
```

INPUT/OUTPUT WITH PRINTF AND SCANF

> MATH,1270,TR,1800 (1)

=> 17 characters

Figure 8.4



- **EXAMPLE 8.1 p.459**

8.2 STRING LIBRARY FUNCTIONS: ASSIGNMENT AND SUBSTRINGS

- Using the assignment symbol in a declaration of a string variable with initialization, is the only one in which the operator means to copy the string that is the right operand into the variable that is the left operand.

```
char one_str[20] = "Test String");
```

- Array address is constant and cannot be changed through assignment **(1)**

```
char one_str[20];
one_str = "Test string"; /* Does not work */
```

a compiler error

Invalid target of assignment :

8.2 STRING LIBRARY FUNCTIONS: ASSIGNMENT AND SUBSTRINGS

- **C Library Functions:**

➢ Along with assignment functions, the library **string.h** provides functions for **substring, concatenation, string length, string comparison, and whole line input operations [See Table 8.1 p.461]**

➢ The data type of the value returned by each string-building function is **the pointer type char * .**

➢ **Reminder**: An array is being represented by the *address of* its initial value.

8.2 STRING LIBRARY FUNCTIONS: ASSIGNMENT AND SUBSTRINGS

- **String Assignment:**

➢ Function **strcpy** copies the string that is its second argument into its first argument

```
char one_str[20];
strcpy(one_str, "Test String");     (1)
```

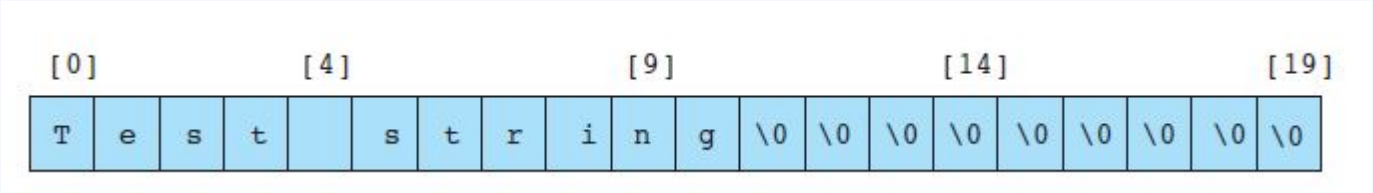
➢ Like a call to **scanf** with a **%s** placeholder, a call to **strcpy** can easily overflow the space allocated for the destination variable (2)

```
strcpy(one_str, "A very long test string");  (3)
```

8.2 STRING LIBRARY FUNCTIONS: ASSIGNMENT AND SUBSTRINGS

- **strncpy** that takes an argument specifying the number of characters to copy (call this number **n**). (1)

strncpy(one_str, "Test string", 20); (2)



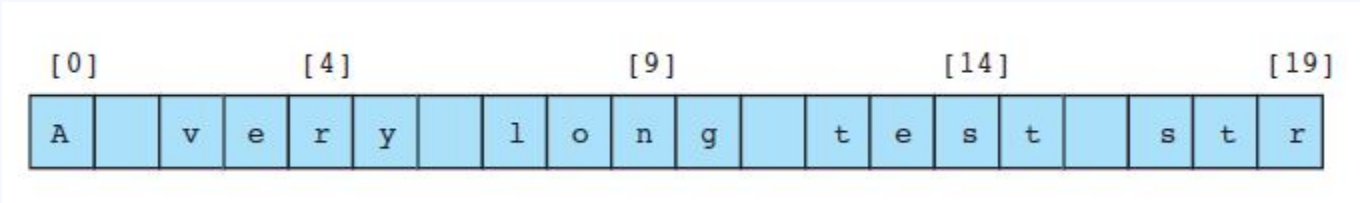
- The effect is the same as the call

strcpy(one_str, "Test string"); (3)

8.2 STRING LIBRARY FUNCTIONS: ASSIGNMENT AND SUBSTRINGS

- when the source string is longer than n characters, only the first n characters are copied.

```
strncpy(one_str, "A very long test string", 20); (1)
```



- In general,

```
strncpy(dest, source, dest_len - 1); (2)
dest[dest_len - 1] = '\0';
```

- Both **strcpy** and **strncpy** mimic the assignment operator in that the value they return is the string assigned. (3)

SUBSTRINGS

- For example, we might want to examine the “**133**" in the string “**Comp133**" or the "**Jan**" in the string "**Jan. 30, 1996**”.

- Assuming that the prototype of **strncpy** is (1)

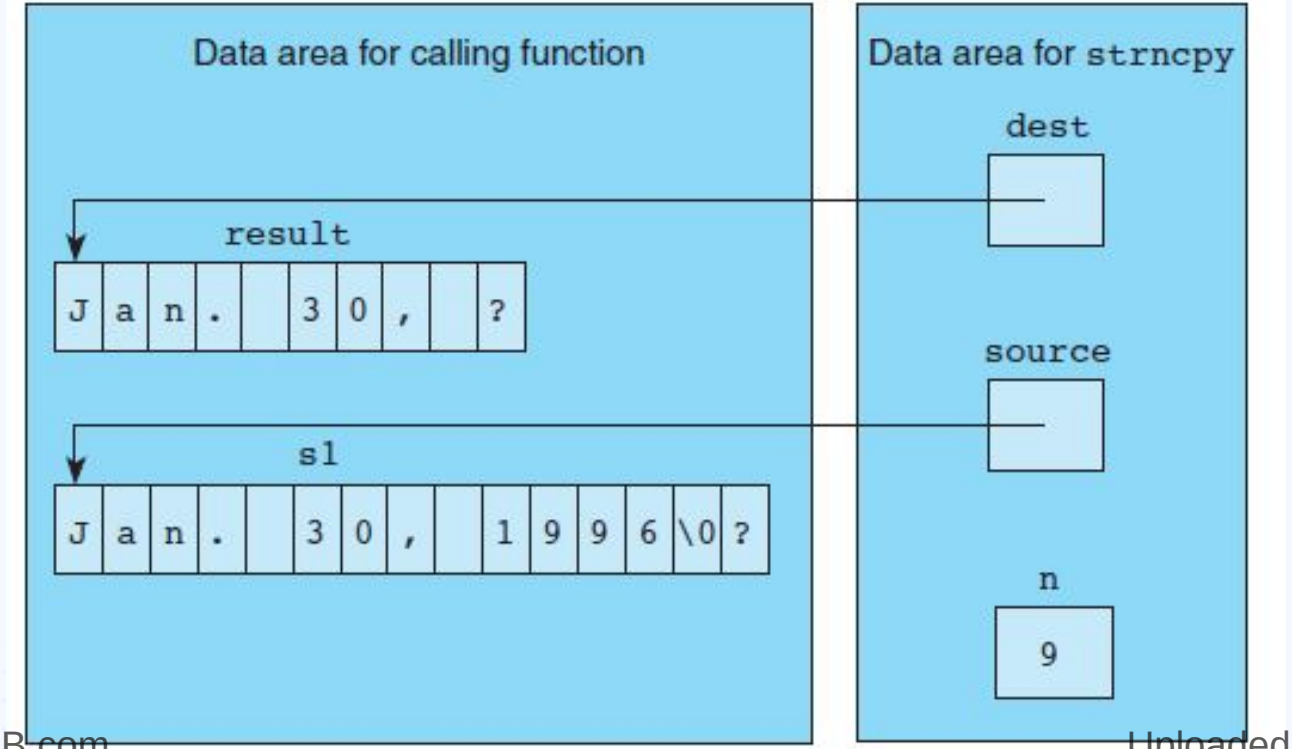
```
char *strncpy(char *dest, const char *source, size_t n);
```

```
char result[10], s1[15] = "Jan. 30, 1996";
strncpy(result, s1, 9);
result[9] = '\0';
```

SUBSTRINGS

Figure 8.5

```
char result[10], s1[15] = "Jan. 30, 1996";
strncpy(result, s1, 9);
result[9] = '\0'; (1)
```



SUBSTRING
S

- If we wish to use **strncpy** to extract a middle substring, we must call the function with the address of the first character to copy.

```
strncpy(result, &s1[5], 2);
result[2] = '\0';    (1)
```

```
strcpy(result, &s1[9]); (2)
```

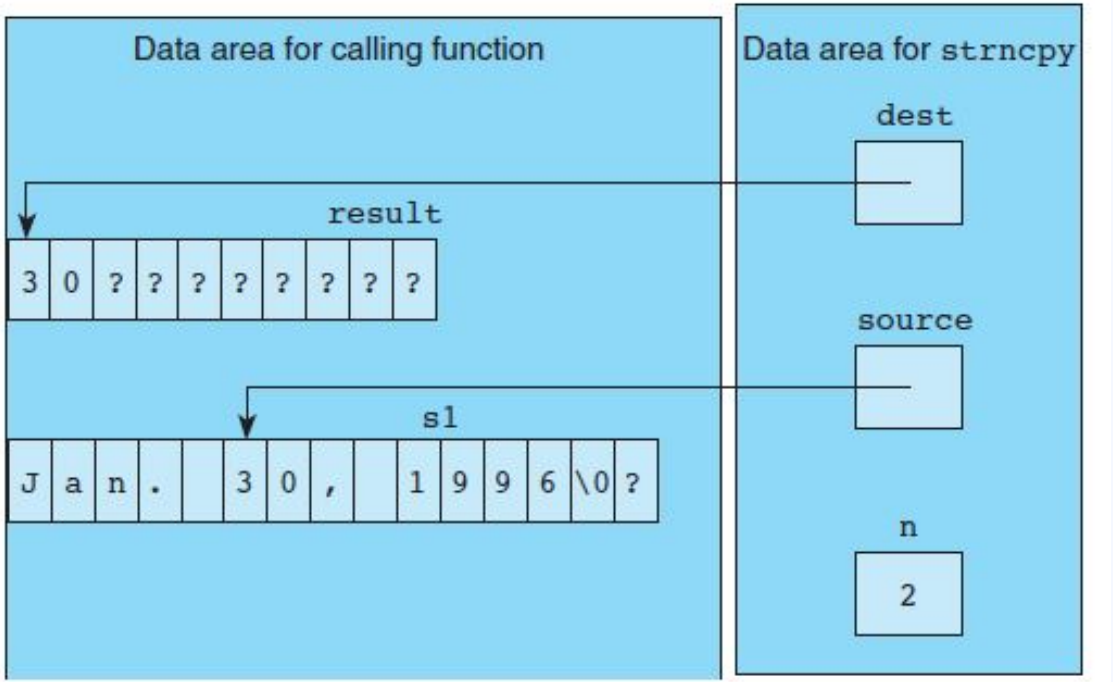
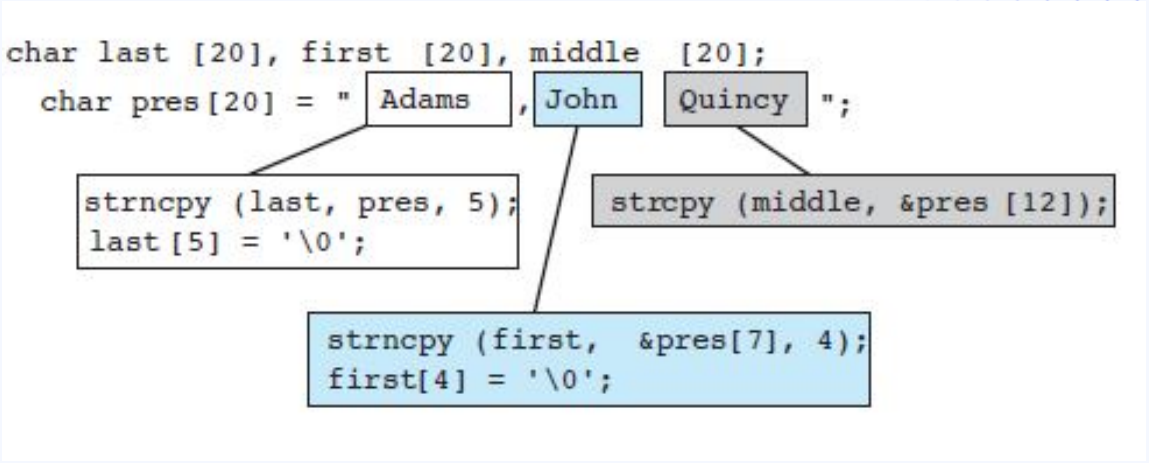


Fig. 8.6

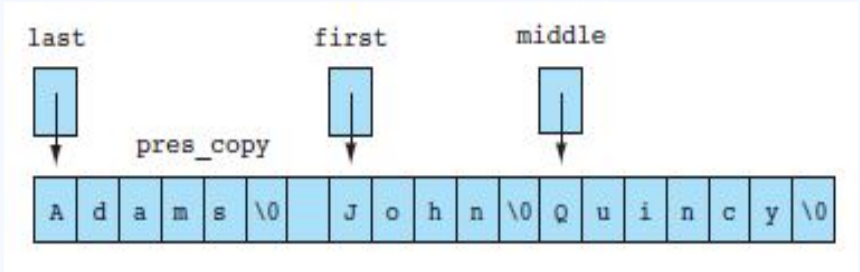
SUBSTRINGS

- **EXAMPLE 8.2 (1)**
- **1st Solution (2)**



- **2nd Solution: (3)**

```
char *last, *first, *middle;
char pres[20] = "Adams, John Quincy";
char pres_copy[20];
strcpy(pres_copy, pres);
last = strtok(pres_copy, ",  ");
first = strtok(NULL, ",  ");
middle = strtok(NULL, ",  ");
```



SUBSTRING
S

- **EXAMPLE 8.3, Figure 8.7**

```
printf("%s\n", strcpy(elem, &compound[first]));
```

d to display the last

- **Instead of: (1)**

Since **strcpy** returns as its value the

```
strcpy(elem, &compound[first]);
printf("%s\n", elem);
```

stored in **elem**

- The use of **CMP_LEN** and **strlen(compound)**: the declared size of the array is no longer the effective size once data have been stored in the array (2)

8.3 LONGER STRINGS: CONCATENATION AND WHOLE-LINE INPUT

- **Concatenation:** String library functions **strcat** and **strncat** modify their first string argument by adding all or part of their second string argument at the end of the first argument.

- Both functions assume that sufficient space is allocated for the first argument to allow addition of the extra characters.

EXAMPLE 8.4

```
#define STRSIZ 15
char f1[STRSIZ] = "John ", f2[STRSIZ] = "Jacqueline ",
last[STRSIZ] = "Kennedy";
strcat(f1, last);   => “John Kennedy”          12 characters + “\0”
strcat(f2, last);=> “Jacqueline Kennedy”   18 characters + “\0”   /* invalid overflow of f2 */
//As an alternative to the second call to strcat (2 )
strncat(f2, last, 3);                              /*Jacqueline Ken*/
```

8.3 LONGER STRINGS: CONCATENATION AND WHOLE-LINE INPUT

- **Concatenation:**
  - Take note that the call to **strcat** modifies the value of **s1** even when embedded in other function calls:

```
printf("%s\n", strncat(f2, last, 3));
```

  - Both **strcat** and **strncat** return as the function value the modified first argument.

  - If **s1** and **s2** are both character strings declared to hold **STRSIZ** characters (1)

```
if (strlen(s1) + strlen(s2) < STRSIZ) {
strcat(s1, s2);
} else {
strncat(s1, s2, STRSIZ - strlen(s1) - 1);
s1[STRSIZ - 1] = '\0';
}
```

8.3 LONGER STRINGS: CONCATENATION AND WHOLE-LINE INPUT

- **String Length:** function **strlen** returns the length of the value of its string argument, not counting the '**\0**' character.
- **For example**: this code fragment displays the numbers 8 and 18 followed by the phrase **Birzeit University** .

```
#define STRSIZ 20
char s1[STRSIZ] = “Birzeit  ",
s2[STRSIZ] = “University";
printf("%d %d\n", strlen(s1), strlen(strcat(s1, s2)));
printf("%s\n", s1);
```

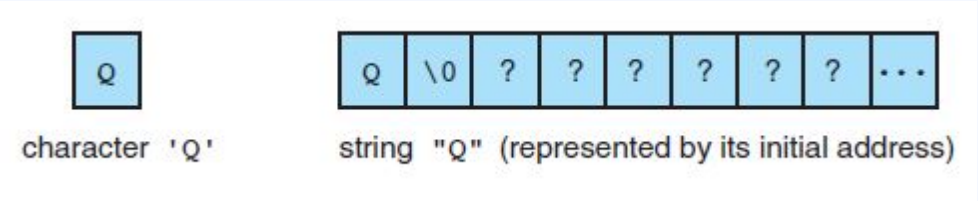
- Notice that the blank at the end of the initial value of **s1** is counted in **s1’**s length.
- Two critical questions that must always be in the mind when working with ***strcpy, strncpy , strcat*** , and ***strncat:***

1. **Is there enough space in the output parameter for the entire string being created?**
2. **Does the created string end in '\0' ?**

DISTINCTION BETWEEN CHARACTERS AND STRINGS

- A type char value is not a valid argument for a function with a corresponding parameter of type char * (1)

- Note the difference internally between the representations of the **character 'Q'** and the **string "Q"** .



- View the string as an array and use assignment to subscripted elements for access. (2)
- Be sure to include the null character at the end of the string.

SCANNING A FULL LINE

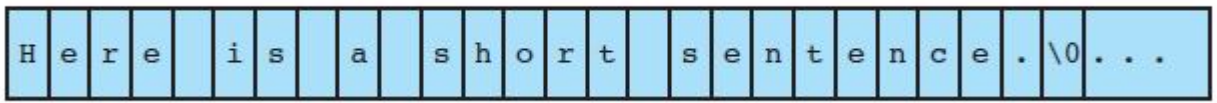
- Although blanks are natural separators to place between numeric data values, viewing them as delimiters (1) when processing strings may not make sense (2)

➢ The **stdio** library provides the function **gets** for interactive input of one complete line of data

```
char line[80];
printf("Type in a line of data.\n> ");
gets(line);
```

```
Type in a line of data.
> Here is a short sentence.
```

(3)



SCANNING A FULL LINE

- The **\n** character representing the **<return>** or **<enter>** key pressed at the end of the sentence is not stored.
- Like **scanf** , **gets** can overflow its string argument if the user enters a longer data line than will fit.

- Function **fgets** takes three arguments: (1)

1. The output parameter string,
2. A maximum number of characters to store (**n**) (2)
3. And the file pointer to the data source

- The next to last character may or may not be **'\n'** :

4. If **fgets** has room to store the entire line of data it will include **'\n'** before **'\0'** .
5. If the line is truncated, no **'\n'** is stored.

SCANNING A FULL LINE

- **fgets** stores the string created in its first argument and returns this string argument (i.e., *its address*) as its value (1)

- When a call to **fgets** encounters the end of file, the value returned is the address 0, which is considered the null pointer.

- **FIGURE 8.8** (2)

8.4 STRING COMPARISON

- We studied the fact that characters are represented by numeric codes, and we used relational and equality operators to compare characters.

```
letter == 'C'
```

```
ch1 < ch2
```

- Unfortunately, these operators cannot be used for comparison of strings. (1)

- if **str1** and **str2** are string variables, the condition is **not** checking whether **str1** precedes **str2** alphabetically (2)

```
str1 < str2
```

- The standard string library provides the **int** function **strcmp** for comparison of two strings that we will refer to as **str1** and **str2**.

8.4 STRING COMPARISON

- Function **strcmp** separates its argument pairs into three categories:

TABLE 8.2 Possible Results of strcmp(str1, str2)

Relationship	Value Returned	Example
`str1` is less than `str2`	negative integer	`str1` is `"marigold"` `str2` is `"tulip"`
`str1` equals `str2`	zero	`str1` and `str2` are both `"end"`
`str1` is greater than `str2`	positive integer	`str1` is `"shrimp"` `str2` is `"crab"`

- **str1[n] < str2[n] if**:

```
str1 t h r i l l
str2 t h r o w
         ^
First 3 letters match.
str1[3] < str2[3]
    'i' < 'o'
```

```
str1 e n e r g y
str2 f o r e
     ^
First 0 letters match.
str1[0] < str2[0]
    'e' < 'f'
```

OR

```
str1   j o y
str2   j o y o u s
```

8.4 STRING COMPARISON

- The string library also provides an analogous function **strncmp** that bases its comparison on only the first **n** characters of the two strings, where **n** is the third argument.

- if **str1** is "**cab**inet" and **str2** is "**cab**le " , the function calls

strncmp(str1, str2, 1) = 0
strncmp(str1, str2, 2) = 0
strncmp(str1, str2, 3) = 0 (1)

- **strncmp(str1, str2, 4)** would return a negative integer (2)

8.4 STRING COMPARISON

EXAMPLE 8.5

FIGURE 8.9 Numeric and String Versions of Portions of Selection Sort That Compare and Exchange Elements

Comparison (in function that finds index of "smallest" remaining element)

Numeric

```
if (list[i] < list[first])
     first = i;
```

String

```
if (strcmp(list[i], list[first]) < 0)
     first = i;
```

Exchange of elements

```
temp = list[index_of_min];
list[index_of_min] = list[fill];
list[fill] = temp;
```

```
strcpy(temp, list[index_of_min]);
strcpy(list[index_of_min], list[fill]);
strcpy(list[fill], temp);
```

EXAMPLE 8.6, FIGURE 8.10

8.5 ARRAYS OF POINTERS

FIGURE 8.11 Exchanging String Elements of an Array

```
1. strcpy(temp, list[index_of_min]);
2. strcpy(list[index_of_min], list[fill]);
3. strcpy(list[fill], temp);
```

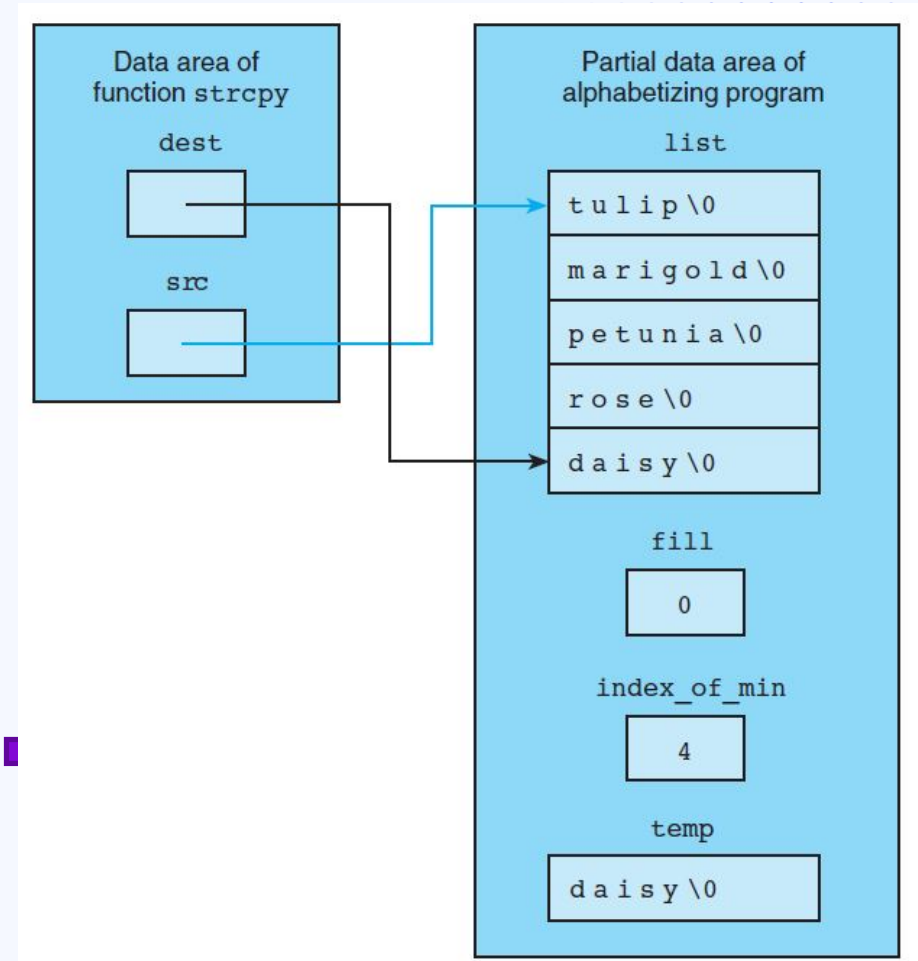
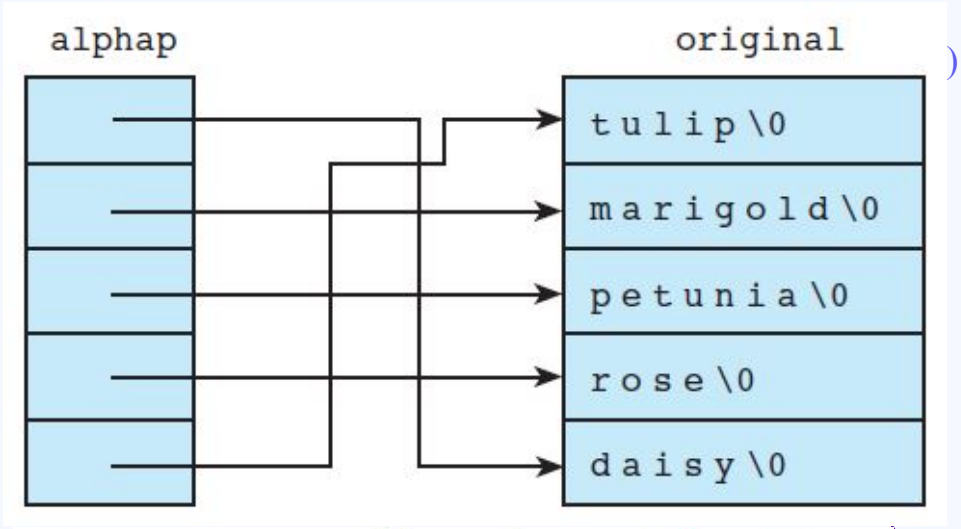


Figure 8.12

8.5 ARRAYS OF POINTERS

➢ C's use of pointers to represent arrays presents us with an opportunity to develop an alternate approach to our sorting problem.




```
alphap[0]    address of    "daisy"
alphap[1]    address of    "marigold"
alphap[2]    address of    "petunia"
alphap[3]    address of    "rose"
alphap[4]    address of    "tulip"
```

```
for (i = 0; i < 5; ++i)
printf("%s\n", alphap[i]);
```

8.5 ARRAYS OF POINTERS

➢ When **printf** sees a **%s** specifier, it always expects to receive the starting address of a string as the corresponding input argument, so **alphap[i]** is just as legitimate an argument as **original[i]** .

➢ Declaring the array of pointers **alphap** (1)

```
char *alphap[5];
```

➢ **EXAMPLE 8.7, FIGURE 8.14** (2)

- **Advantages of this approach:** (3)

1. A pointer (an integer address) requires less storage space than a full copy of a character string.
2. The sorting function executes faster when it copies only pointers and not complete arrays of characters
3. Because the strings themselves are stored only once, a spelling correction made in the original list would automatically be reflected in the other orderings as well.

ARRAYS OF STRING CONSTANTS

- C also permits the use of an array of pointers to represent a list of string constants.

- Two alternatives for representing the list of month names (1)

```
char month[12][10] = {"January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"};
```

```
char *month[12] = {"January", "February", "March", "April", "May",
"June", "July", "August", "September", "October", "November", "December"};
```

- initialization list also implies this value.

8.6 CHARACTER OPERATIONS

➢ When we develop programs that involve string processing, often we must work with the individual characters that make up the string.

➢ **C** provides character input/output routines as part of the **<stdio.h>** library, and an extensive collection of facilities for character analysis and conversion is available in the library we **#include** as **<ctype.h>.**

- **Character Input/Output**

o The **stdio** library includes a routine named **getchar** that is used to get the next character from the standard input source (1).

o **getchar** takes no arguments and returns the character as its result. (2)

scanf("%c", &ch) ⟺ **ch = getchar()** (3)

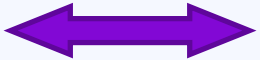
8.6 CHARACTER OPERATIONS

- There is, a difference between the two fragments: (1)
- **scanf** returns an integer representing the number of values read or **EOF**. (2)
- The call to **getchar** returns an integer values representing the character that **getchar** found (3).

- What if there were no characters for **getchar** to take? What if **getchar** came across the end of the data? (4)

- We use a type **int** variable to store the result of a call to **getchar** , at least until we verify that **getchar** did not return **EOF** . (5)

- To get a single character from a file, use the facility **getc**
- **getc** is comparable to a call to **getchar** , except that the character retur**getc(inp)**om the file accessed by file pointer **inp** .



- **EXAMPLE 8.8, FIGURE 8.15**

8.6 CHARACTER OPERATIONS

- The standard library's single-character output facilities are:

1. **putchar** (for display on the standard output device)

```
putchar('a');
```

2. **putc** (for files). (1)

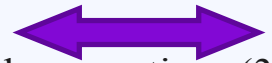
```
putc('a', outp);
```

- **Character Analysis and Conversion:**

- The library we **#include as <ctype.h> : (2)**

3. Helps us decide if a character is a **letter**? a **digit**? A **punctuation mark?**.

4. It also provides routines to do common character conversions like uppercase to lowercase or lowercase to uppercase.



- **Table 8.3 p.485**: lists a number of these routines (3)
- **EXAMPLE 8.9, FIGURE 8.16**

8.7 STRING-TO-NUMBER AND NUMBER-TO-STRING CONVERSIONS

- Common conversions done by **scanf** and **prinft**: (1)

"3.14159"	3.14159	–36	"–36"

TABLE 8.4 Review of Use of scanf

Declaration	Statement	Data (␣ means blank)	Value Stored
char t	scanf("%c", &t);	␣g	
		\n	\n
		A	A
int n	scanf("%d", &n);	␣32␣	32
		␣␣–8.6	–8
		␣+19␣	19
double x	scanf("%lf", &x);	␣␣␣4.32␣	4.32
		␣–8␣	–8.0
		␣1.76e–3␣	.00176
char str[10]	scanf("%s", str);	␣␣hello\n	hello\0
		overlengthy␣	overlengthy\0 (overruns length of str)

TABLE 8.5 Placeholders Used with printf

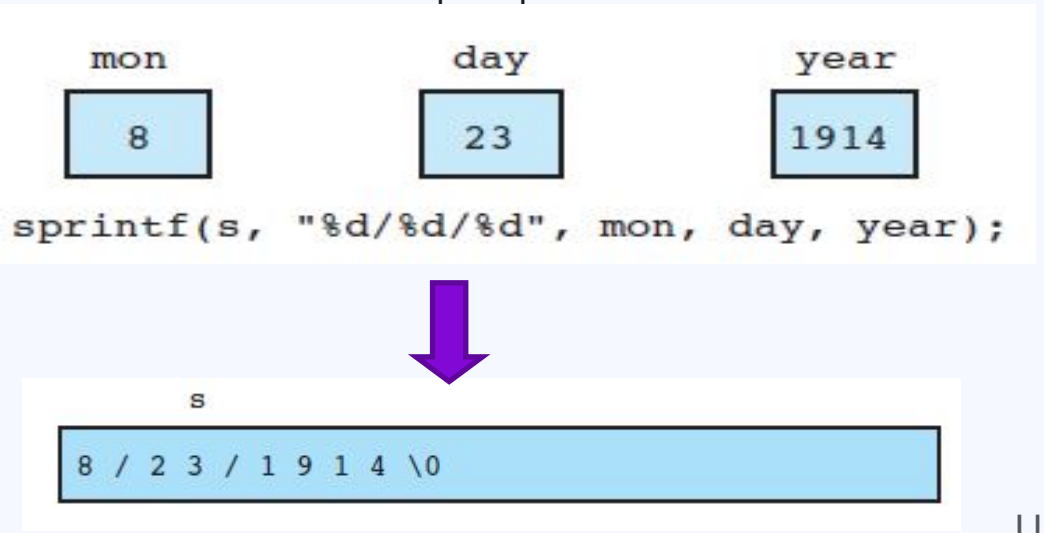
Value	Placeholder	Output (▮ means blank)
'a'	%c	a
	%3c	▮▮a
	%-3c	a▮▮
-10	%d	-10
	%2d	-10
	%4d	▮-10
	%-5d	-10▮▮
49.76	%.3f	49.760
	%.1f	49.8
	%10.2f	▮▮▮▮49.76
	%10.3e	▮4.976e+01
"fantastic"	%s	fantastic
	%6s	fantastic
	%12s	▮▮▮fantastic
	%-12s	fantastic▮▮▮
	%3.3s	fan

8.7 STRING-TO-NUMBER AND NUMBER-TO-STRING CONVERSIONS

➢ The **stdio** library also give us **sprintf** and **sscanf** functions:

- The **sprintf** function requires space for a string as its first argument.
- Instead of printing the result, **sprintf** stores it in the character array accessed by its initial argument. (1)

o assume that s has been declared as char s[100]

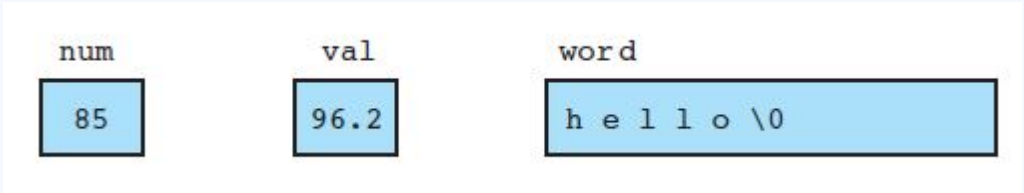


8.7 STRING-TO-NUMBER AND NUMBER-TO-STRING CONVERSIONS

➢ The **sscanf** function takes data from the string that is its first argument (1)

```
sscanf(“  85  96.2  hello", "%d%lf%s", &num, &val, word);
```

stores values from the first string.



- **EXAMPLE 8.10, figure 8.17**
- **Read EXAMPLE 8.11 p.490**

CASE STUDY (Homework)
P.495 - 503

Text Editor

8.9 COMMON PROGRAMMING ERRORS

- String functions do not actually return a string value in the same way that an **int** function returns an integer value.
- Another error is the misuse or neglect of the **&** operator with string use (1).
- Another problem with string use is the overflow of character arrays allocated for strings
- A relatively minor error that can lead to difficult bugs is forgetting that all strings end with the null character.
- It is easy to slip and use equality or relational operators when comparing strings or the assignment operator for copying them.
- Read details of handling these errors p.504

Refernces

Problem Solving and Program Design in C, 7th Ed., by Jeri R. Hanly and Elliot B. Koffman