6
Character Strings

1

Review

- What have you learnt so far:
  - Basic Sequential Programming (data, operators, simple I/O)
  - Branching (if-else, switch, conditional operator)
  - Looping (for, while, do…while)
  - Functions (function definition and call by value)
  - Pointers (call by reference for function communication)
  - 1-D Arrays (+ using pointers for 1D arrays)
  - **2-D (Multi-dimensional) Arrays (+ using pointers for 2D arrays)**

- Arrays are list of data of the same data type.
- In many application, we will need to use character strings such as “Hello World!” as message to interact with user.
- In C, we can declare character strings as arrays of characters.
- In this lecture, we discuss about … **Character Strings** ….

2

This Lecture

- At the end of this lecture, you will be able to understand the following:
  - String Declaration, Initialization and Operations
  - String Input and Output
  - String Functions
  - The ctype.h Character Functions
  - String to Number Conversions
  - Formatted String I/O
  - Arrays of Character Strings

3

Character Strings

In addition to handling numerical data, programs are also required to deal with alphabetic data. Strings are arrays of characters. C libraries provide a number of functions for performing operations on strings. In this chapter, string constants and string variables are first introduced. The different commonly used string functions from C libraries are discussed.

Character Strings

- **String Declaration, Initialization and Operations**
- String Input and Output
- String Functions
- The ctype.h Character Functions
- String to Number Conversions
- Formatted String I/O
- Arrays of Character Strings

4

Character Strings

- A **string** is **an <u>array</u> of characters** terminated by a **NULL** character ('\0').

a	b	c	d	e	f	1	2	3	4	5	O	K	'\0'	

Null character

5

<u>Character Strings</u>

A string is an array of characters terminated by a null character, '**\0**'.

String Constants

- **String constant** is a set of characters in double quotes:
  e.g. **"C Programming"** - is an array of characters and automatically terminated with the null character **'\0'**
- Using #define to define a string constant:
  e.g. #define NTU **"Nanyang Technological Uni."**
- Used in function arguments, e.g. printf() and puts():
  e.g. printf(**"Hello, how are you?"**);

String Constants

A string constant is a series of characters in double quotes:

"C Programming"

When the compiler encounters a string constant, it allocates space for each individual character of the string and adds a terminating null character at the end of the string. A pointer pointing to its first character is returned.

A string constant is essentially a pointer to the first character of an array of characters. We can use **#define** to define string constants. For example,

```
#define NTU "Nanyang Technological University"
```

String constants can also be used as function arguments for **printf()** and **puts()** functions:

```
printf("Welcome to Programming with C.\n");
puts("Welcome to Programming with C.");
```

It is important to distinguish the difference between the character constant '**X**' and a string constant "**X**". The character constant '**X**' consists of a single character of data type **char**, while the character string constant "**X**" consists of two characters, i.e. the character '**X**' and the null character '**\0**', and is an array of type **char**.

String Variables: Declaration using Array Notation

- **String variables**: can be declared using **array notation**

```
char str[ ] = "some text";      // string
char str[10] = "yes";           // legal
char str[4] = "four";           // incorrect -> null character missing
char str[ ] = {'a','b','c','\0'}; // equivalent to:  char str[ ] = "abc";
```

Note: **'\0'** differentiates a character string from an array of characters.

7

String Variables: Declaration

A string is an array of characters. Therefore, we can also define strings using the **array notation**, in addition to the pointer notation.

To declare strings using array notation, we can use the following declarations:

char str[] = "some text";

char str[10] = "yes";

However, the following declaration

char str[4] = "four";

is incorrect. The array **str** only allocates 4 elements to hold its data, which is not enough as it needs an additional element to hold the null character '**\0**' that is automatically added at the end of the sequence of characters.

We can also create strings as follows:

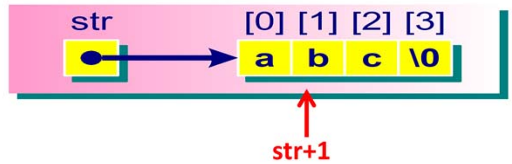
char str[] = {'a','b','c','\0'};

An array of four elements of type **char** is created. However, it is tedious to initialize array by listing all the characters in braces. We can simply enclose the characters using double quotes as if they were a string constant. This is equivalent to

char str[] = "abc";

String Variables: Declaration using Array Notation

- Therefore, just like other kinds of arrays, the array name **str** gives the **address** of the 1st element of the array:



char str[ ] = "abc";

str = = &str[0]
***str** = = 'a'
***(str+1)** = = str[1] = = 'b'

8

String Variables

As a string constant returns a pointer to its first character, the array name **str** contains the address of the first element of the array as other kinds of arrays:

•**str == &str[0] -** the string **str** contains the address of the first element of the array of characters.

•***str == 'a' -** we can use ***str** to retrieve the first element of the string.

•***(str+1) == str[1] == 'b'** - similarly, we can use ***(str+1)** to retrieve the second element of the character string.

String Variables: Declaration using Pointer Notation

- **String variable** can also be declared using the **pointer notation.**
- When declaring string variable using pointer notation, we assign a **string constant** to a **pointer** that points to char:

char *str = "C Programming";

- What does this imply?
When a **string constant** is assigned to a **pointer**, C compiler will:
    1. allocate **memory space** to hold the string constant;
    2. store the **starting address** of the string in the pointer variable;
    3. terminate the string with **NULL** ('\0') character.

9

String Variables: Declaration

Typically, we can declare a string variable by assigning the pointer of the string constant to a **pointer variable**:

char *str = "C Programming";

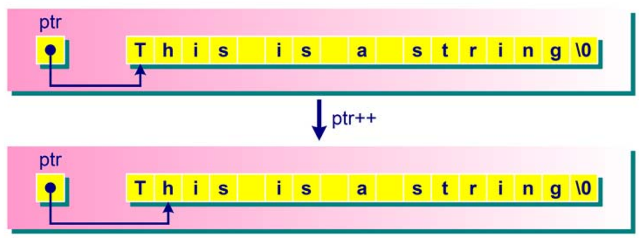
The string variable **str** will then contain the **address** of the first element of the string constant.

String Variables: Declaration using Pointer Notation

- Example:

```
    char *ptr;                    // declared using pointer notation
    ptr = "This is a string“;     // assign a string constant to the ptr variable
or  char *ptr = "This is a string";  // declaration with initialization
```

- For the statement: **ptr++;**
  – it means the pointer variable **ptr** will be updated and points to the next array location:



10

String Variables

We can assign a string constant to a pointer that points to type **char**:

char *ptr = "This is a string";

When a string constant is assigned to a pointer, the C compiler will allocate memory space to hold the string constant, store the starting address of the string (i.e. the address of the character '**T**') in the pointer, and terminate the string with a null character.

However, **ptr** is a **pointer variable** that can be changed. For example, the statement

ptr++;

changes the **ptr** variable to point to the next character (i.e. '**h**') in the string "**This is a string**".

String Variables: Array vs Pointer Declaration

- As can be seen earlier, there are two ways to declare a string:

  char **str1[ ]** = “How are you?"; // using **array** declaration

and

  char ***str2** = “How are you?"; // using **pointer** declaration

- Can you tell the difference between the two declarations?

11

String Variables

It is important to distinguish the declaration of string variables that use array or pointer. For example, the following strings are defined as:

```
char str1[] = "How are you?";      /* using array */
char *str2 = "How are you?";       /* using pointer */
```

The **str1** declaration creates an array of type **char**. The array has been allocated with memory to hold 13 elements including the null character. The C compiler also creates a pointer constant that is initialized to point to the first element of the array, **str1[0]**.

In the **str2** declaration, the C compiler creates a pointer variable **str2** that points to the first character of the string. It contains the memory address of the first character of the string '**H**'.

Question: Can you tell the difference between the two declarations?

String Variables: Array vs Pointer Declaration

- As can be seen earlier, there are two ways to declare a string:

  char **str1[ ]** = “How are you?"; // using **array** declaration

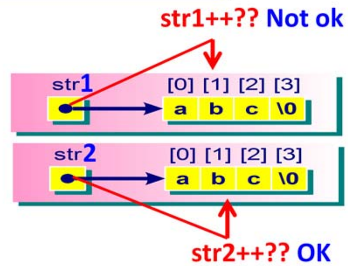
and

  char ***str2** = “How are you?"; // using **pointer** declaration

- Can you tell the difference between the two declarations?
- **str1**: **address constant**, **str2**: **pointer variable**.

Therefore,

```
++str1;        // not allowed
++str2;        // allowed
str1 = str2;   // not allowed
str2 = str1;   // allowed
```



12

String Variables

Both **str1** and **str2** are pointers. However, the difference between the two declarations is that **str1** is a **pointer (or address) constant**, while **str2** is a **pointer variable**. Pointer constant means that the value cannot be changed, while pointer variable allows its value to be changed.

As **str1** is a pointer (or address) constant, we cannot change its value. Therefore, the following statements

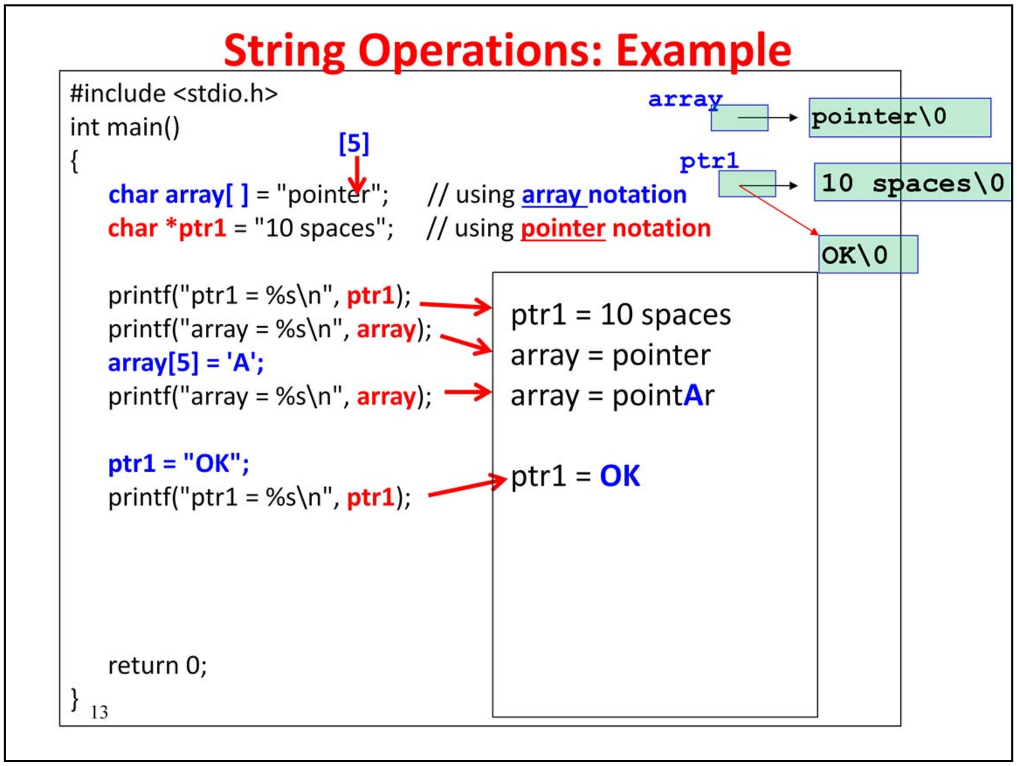
```
str1++;
++str1;
str1 = str2;
```

are invalid.

However, it is valid to have:

```
++str2;
str2=str1;
```

as **str2** is a pointer variable.



String Operations: Example

The declaration

char array[] = "pointer";

declares a string variable **array** using array notation which is initialized with 7 characters plus the null character ('\0').

The declaration

char *ptr1 = "10 spaces";

declares a string variable **ptr1** using pointer notation which is initialized and pointed to the string "**10 spaces**".

When **ptr1** is printed, the string **"10 spaces"** will be displayed.

When **array** is printed, the string **"pointer"** will be displayed.

The statement

array[5] = 'A';

is valid and the string is updated accordingly.

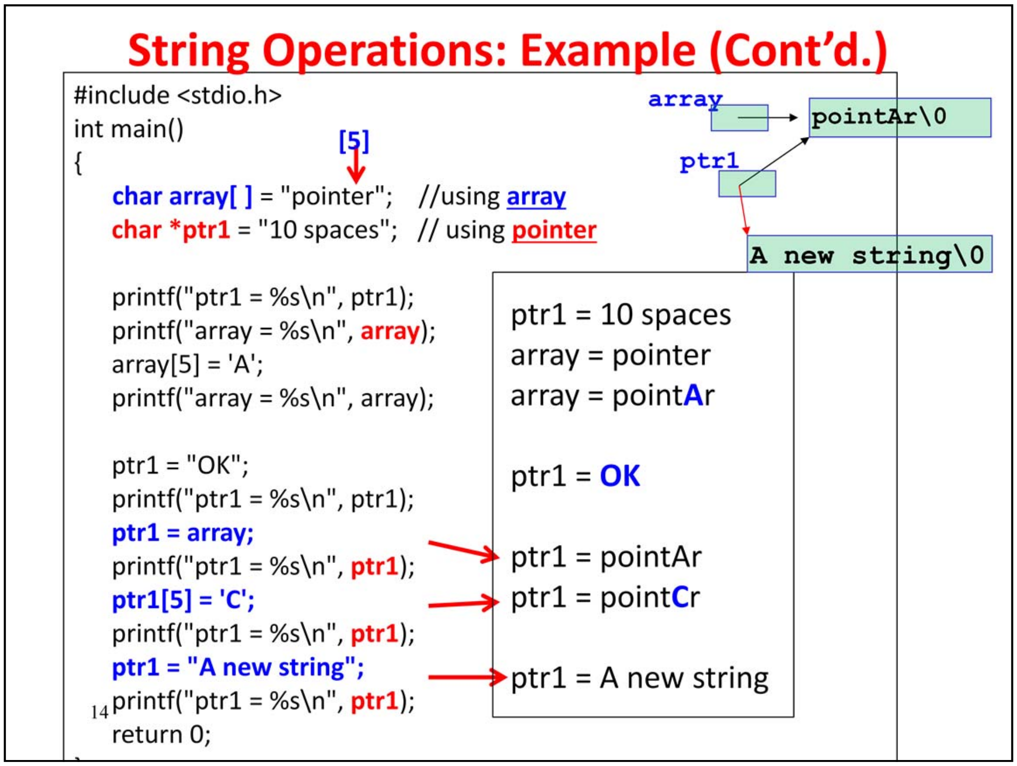
When **array** is printed, the string **"pointAr"** will be displayed.

The statement

ptr1 = "OK";

will update **ptr1** to point to a new string **"OK"**.

When **ptr1** is printed, the string **"OK"** will be displayed.



String Operations: Example

Can we update as an array?
is string mutable?

The statement

ptr1 = array;

changes the pointer variable **ptr1** to point to the same address contained in **array**.

However, if we have

array = ptr1;

which will be invalid because an error on type mismatch will occur. Also, we are not allowed to change the base address of **array**.

When **ptr1** is printed, the string **"pointAr"** will be displayed.

The statement

ptr1[5] = 'C';

is valid and the string is updated accordingly.

When **ptr1** is printed, the string **"pointCr"** will be displayed.

The statement

ptr1 = "A new string";

is valid which changes the pointer variable **ptr1** to point to the new string "**A new string**".

Character Strings

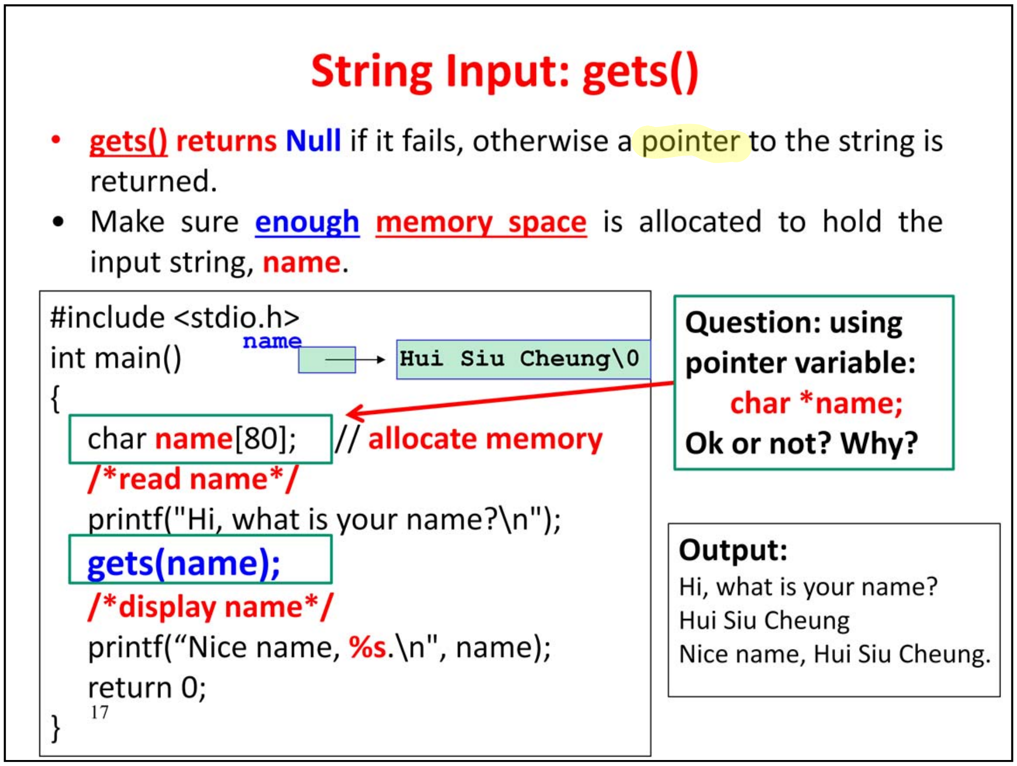
- String Declaration, Initialization and Operations
- **String Input and Output**
- String Functions
- The ctype.h Character Functions
- String to Number Conversions
- Formatted String I/O
- Arrays of Character Strings

15

String Input/Output

- There are 4 C library functions that can be used for string input/output:
  - gets()
  - puts()
  - scanf()
  - printf()
- What are the differences between these functions?

16



String Input

The two most commonly used standard library functions for reading strings are **gets()** and **scanf()**. For printing strings, the two standard library functions are **puts()** and **printf()**.

Creating Memory Space

Before reading a string, it is important to allocate enough storage space to store the string. To create space for a string, we may include an explicit array size as shown in the following declaration:

char name[80];

This declaration statement creates a character array **name** of 80 elements. Once the storage space has been acquired, we can read in the string through the C library functions.

The gets() Function

The **gets()** function gets a string from the standard input device. It reads characters until it reaches a newline character (**\n**). A newline character is generated when the **<Enter>** key is pressed. The **gets()** function reads all the characters up to and including the newline character, replaces the newline character with a null character and passes them to the calling function as a string.

The function prototype of **gets()** is

char *gets(char *ptr);

The **gets()** function returns a null pointer if it fails, otherwise the same pointer **ptr** is returned. It is important to make sure that sufficient storage is allocated to hold the input string. The null pointer contains a null address that is the symbolic constant **NULL** defined in the header file *stdio.h*. This is different from the null character ('**\0**') in that the null character is a data object of type **char** with ASCII value 0, whereas the null pointer contains an address. In addition, the size of the null character is 1 byte, while the null pointer is 4 bytes long.

In the program, it reads in a string and prints the string to the screen. It is important to ensure that the array size of **name** is big enough to hold the input string. Otherwise, the extra characters can overwrite the adjacent memory variables.

String Output: puts()

```c
#include <stdio.h>
int main( )
{
    char str[80];          // string with allocated memory
    printf("Enter a line of string: ");
    if (gets(str) == NULL) {
        printf("Error\n");
    }
   puts(str);
    return 0;
}
```

Input: **0123456789 OK**

0	1	2	3	4	5	6	7	8	9		O	K	'\0'		

18 ***Output***: **0123456789 OK**

The puts() Function

The function prototype for the **puts()** function is

int puts(const char *ptr);

It prints a string on the standard output device. A newline character is automatically added to the end of the string. Thus, the *newline* character is printed after the string. **EOF** is returned if **puts()** fails, otherwise the number of characters written will be returned.

In the program, it reads in a string using the **gets()** function and prints the string on the screen using the **puts()** function.

String Input/Output: scanf() and printf()

- **scanf()**
  - The scanf() function reads the string up to the next **whitespace** character.
  - scanf() **returns** the **number of items** read by scanf(), otherwise **EOF** if fails.
  - Make sure that enough memory space is allocated for input string.
- **printf()**
  - The printf() function **returns** the **number of characters** transmitted, otherwise a negative value will be returned if it fails.

19

The scanf() Function

The **scanf()** function with the **%s** format specification can be used to read in a string. This function will return **EOF** if it fails, otherwise the number of items read by **scanf()** will be returned. An example of the **scanf()** function to read in a string is given as follows:

```
scanf("%s", str);
```

Unlike other data input, there is no need to have the address-of (**&**) operator to be placed before the name of the array. This is due to the fact that the array name is the address of the first element of the array.

It is important to ensure that **sufficient storage** is allocated to hold the string. If more characters are read in than the storage space allocated to hold the string, the additional characters will be overwritten to the adjacent memory locations. For example, consider the following declaration statements:

```
char *str = "string space";
scanf("%s", str);
```

A string pointer constant is created, and 13 bytes of storage are created to hold the string constant. If the string read in by the **scanf()** function is longer than 13 bytes, it will be overwritten to the adjacent memory locations.

There are two ways to read input string in **scanf()**. We may use either a **%s** or **%ns** conversion specifier, where **n** is the field width specification. In both cases, reading starts at the first non-whitespace character. When **%ns** is used, **scanf()** reads up to **n** input characters or when the first whitespace character (i.e. blank, tab or newline) is encountered. For example, **%6s** will stop after the first six characters are read. When **%s** is specified, the **scanf()** function reads the string up to the next whitespace character.

The printf() Function

The function prototype for the **printf()** function is

printf(control-string, argument-list);

It prints formatted output on the standard output device. It returns a negative value if **printf()** fails, otherwise the number of characters printed will be returned. It differs from the **puts()** function in that no newline is added at the end of the string. The **printf()** function is less convenient to use than the **puts()** function. However, the **printf()** function provides the flexibility to the user to control the format of the data to be printed.

scanf() and printf(): Example

```c
#include <stdio.h>
int main( )
{
    char name1[20], name2[20], name3[20];
    int count;
    printf("Please enter your strings.\n");
    count = scanf("%s %s %s", name1, name2, name3);
    printf("I read the %d strings: %s %s %s\n", count, name1,
          name2, name3);
    return 0;
}
```

Output

```
Please enter your strings.
Hui Siu Cheung
I read the 3 strings: Hui Siu Cheung
```

Separated by space

20

scanf() and printf(): Example

In the program, the **scanf()** function is used to read a string from the input. If the input “**Hui Siu Cheung**" is entered, then **name1** = "**Hui**", **name2** = "**Siu**" and **name3** = "**Cheung**". The **scanf()** function returns the **count** on the number of input strings.

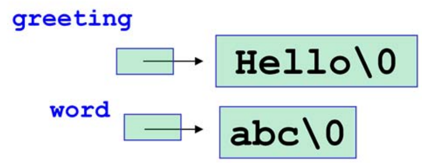
scanf() vs gets()

The **gets()** function differs from the **scanf()** function in that **gets()** reads an entire line up to the first newline character, and it also stores any characters, including whitespace, up to the first newline character. The **scanf()** function is less convenient to use than the **gets()** function for reading string input. The **gets()** function is also faster than the **scanf()** function.

String Processing – Compute String Length

```
#include <stdio.h>
int length1(char []);
int length2(char *);
int main( )
{
    char   *greeting = "hello", word[] = "abc";
    printf("The length is %d, %d\n",
        length1(greeting),
        length2(word));
    return 0;
}
```

Output
The length is 5, 3



21

String Processing: Compute String Length

In the program, it illustrates the difference between array and pointer notations for processing strings. Two functions are implemented in the program. The function **length1()** uses the array notation and the function **length2()** uses the pointer notation to compute the length of a string.

In the **main()** function, the declaration

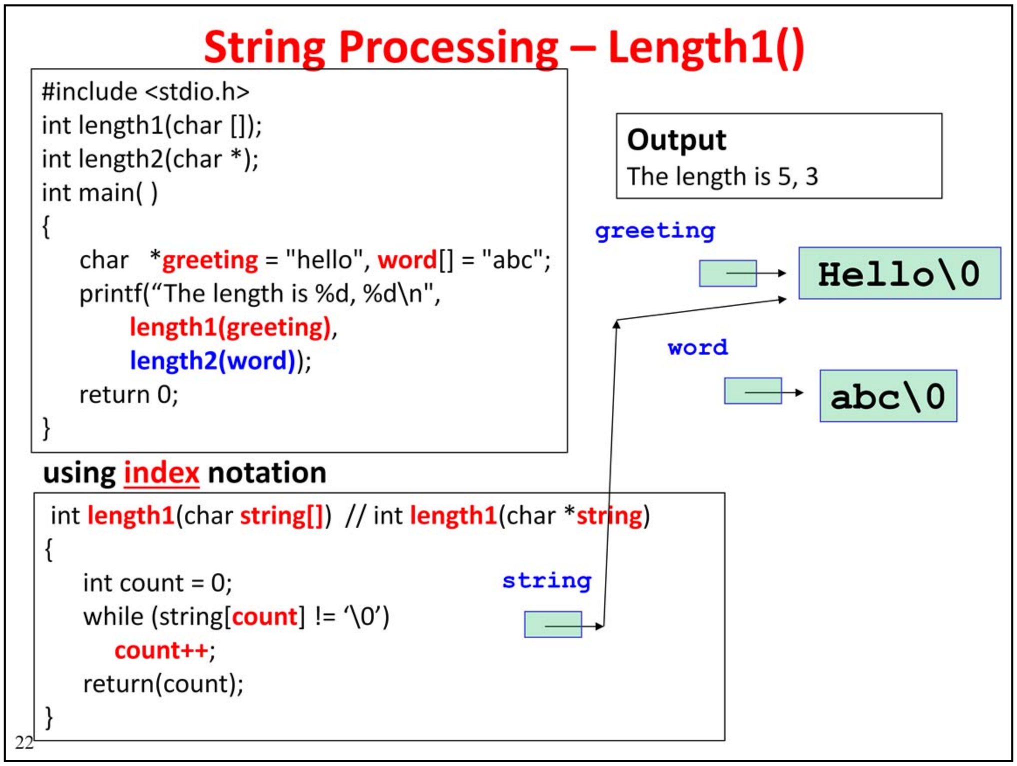
char *greeting = "hello";

creates a pointer variable called **greeting** that points to the first character of the string "**hello**".

The declaration

char word[] = "abc";

creates an array of type **char** called **word[]**. This array contains four elements including the null character.

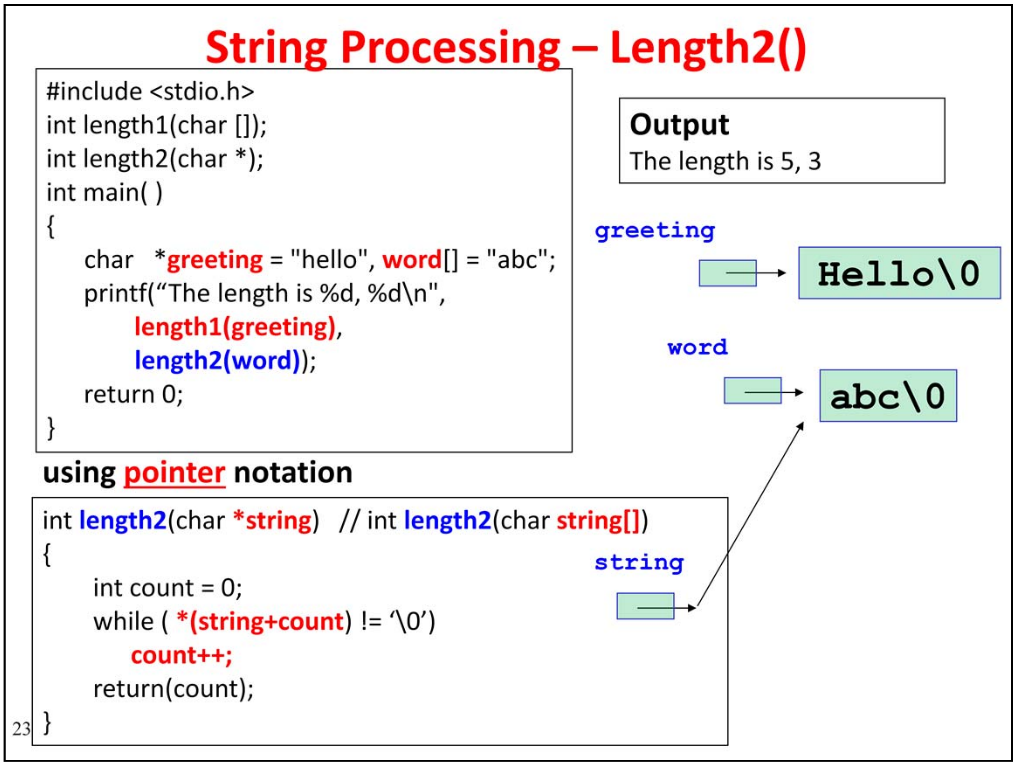


<u>**String Processing: Length1()**</u>

In the function **length1()**, it uses the index notation for the implementation. The function **length1()** uses the statement

while (string[count] != '\0')

to check for the null character ('**\0**') in the **while** loop while measuring the length of the string.



String Processing: Length2()

In the function **length2()**, it uses the pointer notation for the implementation. The function **length2()** uses the statement

while (*(string + count) != '\0')

to check for terminating null character. The expression **(*(string + count))** first gets the character stored at the address location contained in **(string + count)**, and then tests whether it is a null character. For any characters other than the null character, the condition will be true, and the statements in the body of the **while** loop will be executed to measure the length of the string.

Character Strings

- String Declaration, Initialization and Operations
- String Input and Output
- **String Functions**
- The ctype.h Character Functions
- String to Number Conversions
- Formatted String I/O
- Arrays of Character Strings

24

String Functions

Must include the following header file: **#include <string.h>**

Some standard string functions are:

strcat()	**appends one string to another**
strncat()	**appends a portion of a string to another string**
strchr()	**finds the first occurrence of a specified character in a string**
strrchr()	**finds the last occurrence of a specified characters in a string**
strcmp()	**compares two strings**
strncmp()	**compares two strings up to a specified number of characters**
strcpy()	**copies a string to an array**
strncpy()	**copies a portion of a string to an array**

25

String Functions

The standard C library provides many functions that perform string operations. To use any of these functions, we must include the header file *string.h* in the program:

#include <string.h>

String functions are very useful for writing programs that involve the manipulation of strings. Some examples of string functions include finding the length of strings, combining two strings, comparing two strings, and searching a string for a specific character. Programmers are encouraged to use these functions instead of developing their own functions. In the table, it lists some of the more commonly used string functions provided in *string.h*. In this lecture, we describe some of the most useful C string handling functions. These include **strlen()**, **strcat()**, **strcpy()**, **strcmp()**, **strchr(), strrchr()** and **strstr()**.

•The **strlen()** function computes the length of a string.

•The **strcat()** function appends one string to another.

•The **strcpy()** function copies a string to an array.

•The **strcmp()** function compares two strings.

•The **strchr()** function searches the first occurrence (leftmost) of the character in the string, and returns a pointer to that occurrence. A null pointer will be returned if it does not find the character.

•The **strrchr()** function works similarly to **strchr()** except that it searches for the last occurrence (rightmost) of the character in the string, and returns a pointer to that occurrence. Similarly, a null pointer will be returned if it does not find the character in the string.

•The **strstr()** function searches for a substring.

In the subsequent slides, we will discuss the 4 most common functions: **strlen()**, **strcat()**, **strcpy()** and **strcmp()**.

String Functions (Cont'd.)

strcspn()	computes the length of a string that does not contain specified characters
strstr()	searches for a substring
strlen()	computes the length of a string
strpbrk()	finds the first occurrence of any specified characters in a string
strtok()	breaks a string into a sequence of tokens

26

The strlen() Function

- The function prototype of **strlen** is

```
size_t  strlen(const char *s);
```

strlen computes and **returns** **the length of the string** pointed to by *s*, i.e. the number of characters that precede the terminating null character.

27

The strlen() Function

The **strlen()** function finds the length of a string. The function prototype of **strlen()** is

unsigned strlen(const char *str);

strlen() computes the length of the string pointed to by **str**. It returns the number of characters that precede the terminating null character. The null character is excluded in the calculation.

The strlen() Function: Example

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

int main()
{
    char line[81] = "This is a string";

    printf("The length of the string is %d.\n", strlen(line));
   return 0;
}
```

Output
The length of the string is 16.

28

The strlen() Function: Example

In the program, it uses the **strlen()** function to obtain the length of the string variable **line**. The program creates a character array called **line[]** to store the string. The character array **line[]** is initialized to the character string constant "**This is a string**". The length of this string is then calculated using the **strlen()** function, and printed on the display.

Note that the **sizeof** operator can also be used to determine the number of characters in a string. However, this function includes the terminating null character in its calculation.

The strcat() Function

- The function prototype of **strcat** is

```
char *strcat(char *s1, const char *s2);
```

strcat **appends** a copy of the string pointed to by s2 to the end of the string pointed to by s1.

The initial character of s2 overwrites the null character at the end of s1.

strcat **returns** the value of **s1** (i.e. string)

29

The strcat() Function

The **strcat()** function concatenates two strings to form a new string. The function prototype of **strcat()** is

char *strcat(char *str1, const char *str2);

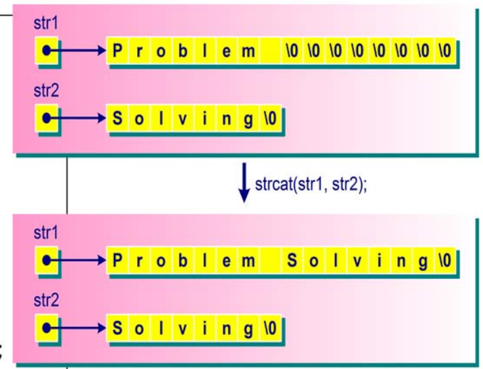
strcat() appends a copy of the string pointed to by **str2** to the end of the string pointed to by **str1**. The initial character of **str2** overwrites the null character at the end of **str1**. **strcat()** returns the address value of **str1**. **str2** is unchanged.

The strcat() Function: Example

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

int main()
{
    char str1[40] = "Problem ";
    char *str2 = "Solving";

    printf("The first string: %s\n", str1);
    printf("The second string: %s\n", str2);
    strcat(str1, str2);
    printf("The combined string: %s\n", str1);
    return 0;
}
```



Output
The first string: Problem
The second string : Solving
The combined string: Problem Solving

30

The strcat() Function: Example

In the program, it uses the **strcat()** function to concatenate two strings. The two strings are declared and initialized as follows:

char str1[40] = "Problem ";

char *str2 = "Solving";

After the function

strcat(str1, str2);

is executed, **str2** is unchanged and **str1** has been changed to store **"Problem Solving"**.

It is important to note that the storage allocated to **str1** should be big enough to hold the concatenated string as **strcat()** will not check the storage requirement before performing the concatenation operation.

The strcpy() Function

- The function prototype of **strcpy** is

```
char *strcpy(char *s1, const char *s2);
```

strcpy copies the string pointed to by s2 into the array pointed to by s1.

It **returns** the value of **s1** (i.e. string).

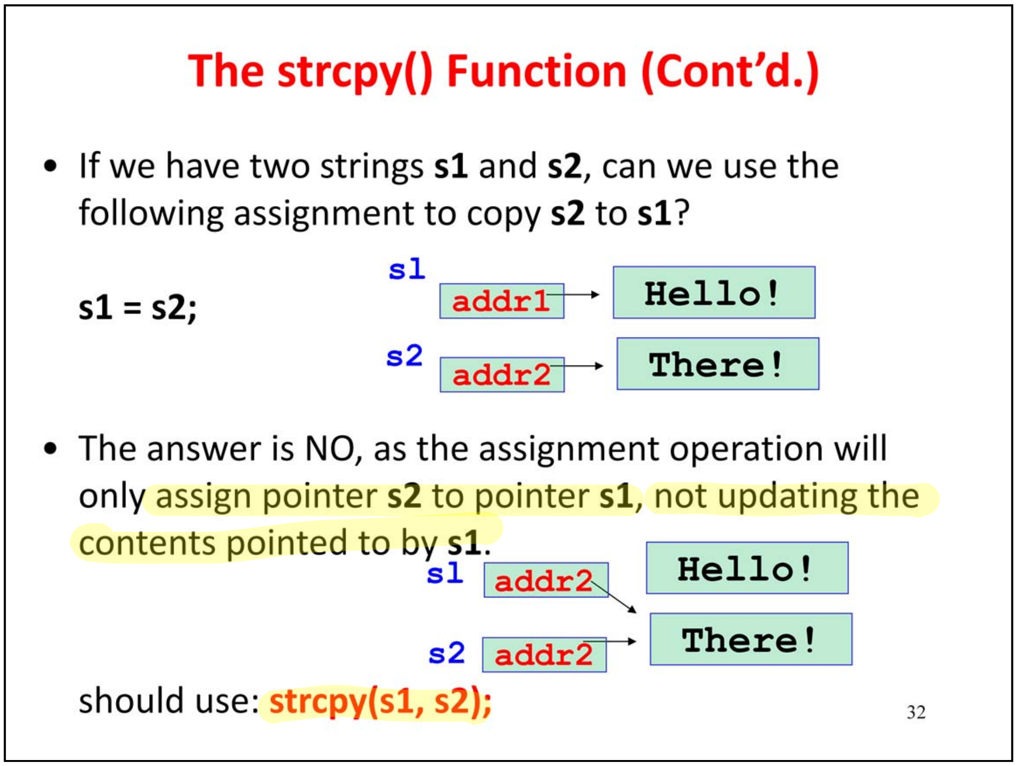
31

The strcpy() Function

The **strcpy()** function copies one string to another. The function prototype of **strcpy()** is

char *strcpy(char *str1, const char *str2);

The function copies all the characters in the string pointed to by **str2** into the array pointed to by **str1**. The copy operation includes the null character in **str2**. **str1** acts as the target string while **str2** is the source string. The order of the strings is similar to the assignment statement where the target string is on the left-hand side. The **strcpy()** function returns the value of **str1**.



<u>The strcpy() Function</u>

Note that when you want to copy a string **s2** to another string **s1**, you cannot use the following statement:

s1 = s2;

The assignment statement will only assign pointer **s2** to pointer **s1**, it does not update the contents pointed to by **s1**. Therefore, you need to use the **strcpy()** function when you want to copy the content of a string to another string.

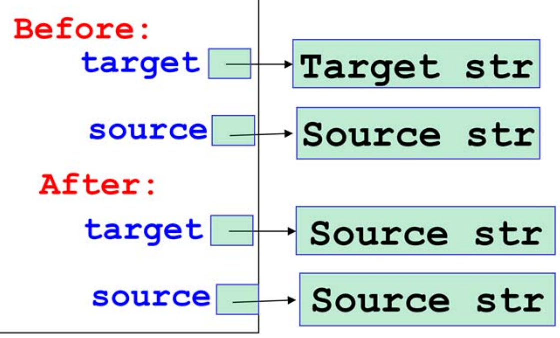
Alice

The strcpy() Function: Example

```
#include <stdio.h>
#include <string.h>
  int main()
  {
    char target[40] = "This is the target string.";
    char *source = "This is the source string.";

    puts(target);
    puts(source);
    strcpy(target, source);
    puts(target);
    puts(source);
    return 0;
  }
```

Output
This is the target string.
This is the source string.
This is the source string.
This is the source string.



33

hello hola

The strcpy() Function: Example

In the program, it use **strcpy()** function to copy a string to another string. It is important to note that the target array must have enough space to hold the updated string contents.

The declaration

char target[40];

is used instead of

char *target;

It is because the latter declaration does not have space allocated to hold the string. wut??

The strcmp() Function

- The function prototype of **strcmp** is

```
int strcmp(const char *s1, const char *s2);
```

strcmp compares the string pointed to by s1 to the string pointed to by s2.

- It **returns** an integer >, =, or < zero, accordingly if the string pointed to by s1 is >, =, or < the string pointed to by s2:
  - **0:** if the two strings are equal
  - **> 0 (i.e. the difference or 1 depending on system)**: if the first string follows the second string alphabetically, i.e. first string is larger (based on ASCII values)
  - **< 0 (i.e. the difference or -1)**: if the first string comes first alphabetically, i.e. the first string is smaller (based on ASCII values)

34

The strcmp() Function

The **strcmp()** function compares the contents of two strings. The prototype of **strcmp()** is

int strcmp(const char *str1, const char *str2);

It compares the string pointed to by **str1** to the string pointed to by **str2**. It takes the two strings and performs a letter-by-letter, alphabetic order comparison. The comparison is based on ASCII codes for the characters. It returns an integer greater than, equal to or less than zero, according to whether the string pointed to by **str1** is greater than, equal to or less than the string pointed to by **str2** respectively.

It is interesting to note that in ASCII codes, uppercase characters come before lowercase characters, and digits come before the letters. For example, '**a**' has the ASCII code value of 97, while '**A**' has 65. If the initial characters are the same, then the **strcmp()** function moves along the string until it finds the first pair of different characters, and returns the comparison result. For example, when comparing "**abcde**" with "**abcd**", the first four characters are the same in the two strings. But when it comes to the character '**e**' in the first string, it will be comparing with the terminating null character with ASCII code value of 0 in the second string. The function then returns a positive value. This way of ordering strings is called *lexicographic order.*

Comparison based on Character ASCII Value

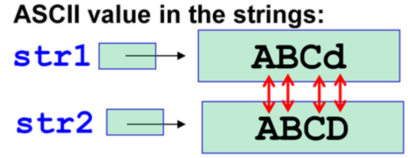
	0	1	2	3	4	5	6	7	8	9
0	NUL							BEL	BS	TAB
1	LF		FF	CR						
2								ESC		
3			SP	!	"	#	$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

The strcmp() Function: Example 1

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[81], str2[81];
    int result;

    printf("String Comparison:\n");
    printf("Enter the first string: ");
    gets(str1);
    printf("Enter the second string: ");
    gets(str2);
    result = strcmp(str1, str2);
    printf("The result of the comparison is
            %d\n\n", result);
    return 0;
}
```

Compare char by char using ASCII value in the strings:



Output

String Comparison:
Enter the first string: _A_
Enter the second string: _B_
The result of the comparison is **-1**

Enter the first string: _ABCd_
Enter the second string: _ABCD_
The result of the comparison is **1**

Enter the first string: _A0_
Enter the second string: _A1_
The result of the comparison is **-1**

Here, only 1, 0 or -1 is returned, it could also be the difference in ASCII values depending on the system.

36

The strcmp() Function: Example 1

In the program, it uses the **strcmp()** function to compare two strings. Generally, we are not interested in the actual values returned by **strcmp()**. The actual values to be returned depend on individual system implementation. When comparing "A" to "C", some systems return –1, others return –2, i.e. the difference in ASCII code values. However, in many applications, we are only interested to find out whether the two strings are equal or not.

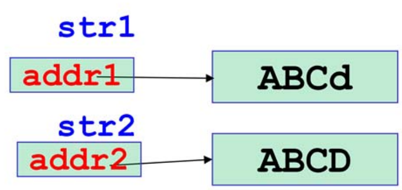
The strcmp() Function: Example 1

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[81], str2[81];
    int result;

    printf("String Comparison:\n");
    printf("Enter the first string: ");
    gets(str1);
    printf("Enter the second string: ");
    gets(str2);
    result = strcmp(str1, str2);
    printf("The result of the comparison is
            %d\n\n", result);
    return 0;
}
```

Question: In this program, could we use:
if (str1 == str2)
…….
instead?

Note: It compares the **addresses** stored in str1 (addr1) and str2 (addr2) only, not comparing the contents.



37

The strcmp() Function: Example 1

In this program, if we use the following statement:

if (str1 == str2)

to compare the two strings instead of using the **strcmp()** function, will this be ok?

The answer to this question is "not ok". It is because the contents of the two strings are addresses. Comparing the two strings in this way only compares the addresses contained in the two string variables. It does not compare the character contents stored in the strings. Therefore, we need to use the **strcmp()** function to compare the two strings.

The strcmp() Function: Example 2

```c
/* Read a few lines from standard
input & write each line to standard
output with the characters reversed.
The input terminates with the line
"END"*/
#include <stdio.h>
#include <string.h>
void reverse(char *);
int main()
{
    char line[132];
    gets(line);
    while (strcmp(line, "END") !=0) {
            reverse(line);
            printf("%s\n", line);
            gets(line);
    }
}
```

```c
void reverse(char *s)
{
    char c, *end;
    end = s + strlen(s) - 1;
    while (s < end) {
      /* 2 ends approaching centre */
      /* swapping operation */



    }
}
```

s end

How are you

s--> <--end

uoy era woH

END

38

The strcmp() Function: Example 2

In the program, it uses the **strcmp()** function to check whether to exit the loop after reading in an input string. The loop will end if the input string is "**END**". In the **main()** function of the program, it reads in a string from the standard input, calls the function **reverse()** to reverse the characters in the string, and writes the reversed string to the standard output. The loop will continue until the user enters the string "**END**".

The strcmp() Function: Example 2

```c
/* Read a few lines from standard
input & write each line to standard
output with the characters reversed.
The input terminates with the line
"END"*/
#include <stdio.h>
#include <string.h>
void reverse(char *);
int main()
{
    char line[132];
    gets(line);
    while (strcmp(line, "END") !=0) {
            reverse(line);
            printf("%s\n", line);
            gets(line);
    }
}
```

```c
void reverse(char *s)
{
    char c, *end;
    end = s + strlen(s) - 1;
    while (s < end) {
      /* 2 ends approaching centre */
      /* swapping operation */
         c = *s;
         *s++ = *end;   /*postfix op*/
               // i.e. *s = *end; s++;
         *end-- = c;
               // i.e. *end = c; end--;
    }
}
```

s → ← end

How are you

s--> <--end

uoy era woH

END

39

The strcmp() Function: Example 2

In the **reverse()** function, it accepts an input string as its parameter. The function then determines the position of the **end** pointer with the following statement:

end = s + strlen(s) - 1;

The function then uses a **while** loop to process each character in the string. In the loop, it swaps the characters from both ends of the string, and then moves the string pointer **s** and the end string pointer **end** towards the center of the string with **s++** and **end--**. The swapping operation repeats until the condition **s<end** is false. After the operation, the input string will be reversed. For example, if the input string **s** = **"**How are you**"**, then the reversed string will be **"**uoy era who**"**.

Note that the **reverse()** function illustrates a typical example on string processing.

Character Strings

- String Declaration, Initialization and Operations
- String Input and Output
- String Functions
- **The ctype.h Character Functions**
- String to Number Conversions
- Formatted String I/O
- Arrays of Character Strings

40

Name	True If Argument is
Isalnum	Alphanumeric (alphabetic or numeric)
isalpha	Alphabetic
iscntrl	A control character, e.g. Control-B
isdigit	A digit
isgraph	Any printing character other than a space
islower	A lowercase character
isprint	A printing character
ispunct	A punctuation character (any printing character other than a space or an alphanumeric character)
Isspace	A whitespace character: space, newline, formfeed, carriage return, etc.
Isupper	An uppercase character
Isxdigt	A hexadecimal-digit character

ctype.h Functions

- Return **true** **(non-zero)** if the character belongs to a particular class;
- Return **false** **(zero)** otherwise.
- Must include the following header file: **#include <ctype.h>**

41

The ctype.h Functions

C also contains the character processing library, whose functions are declared in the *ctype.h* header file. These functions are used to test the nature of a character. It returns *true* if the condition being tested is satisfied, or *false* otherwise. To use these functions, we must include the *ctype.h* header file in the programs. Some of the most commonly used functions are given in the table.

The character testing functions are very useful. For example, when an input might contain any sequence of input characters, we can use the function such as **islower()**, **isupper()**, **isdigit()**, **isalpha()**, **isalnum()** or **isspace()** to test each input character and then process the character accordingly.

ctype: Character Conversion Functions

- **toupper()** - maps lowercase character to uppercase;
  tolower() - maps uppercase character to lowercase;

```
#include <stdio.h>
#include <ctype.h>
int main()  {
    char str[80];              // allocate memory
    printf("Enter a string of text: \n");
    gets(str);     modify(str);     puts(str);
    return 0;
 }
void modify(char* str) {

    // process str, convert upper case letter  to
    // lower case and vice versa
}
```

Output
This is a test
tTIS IS A TEST

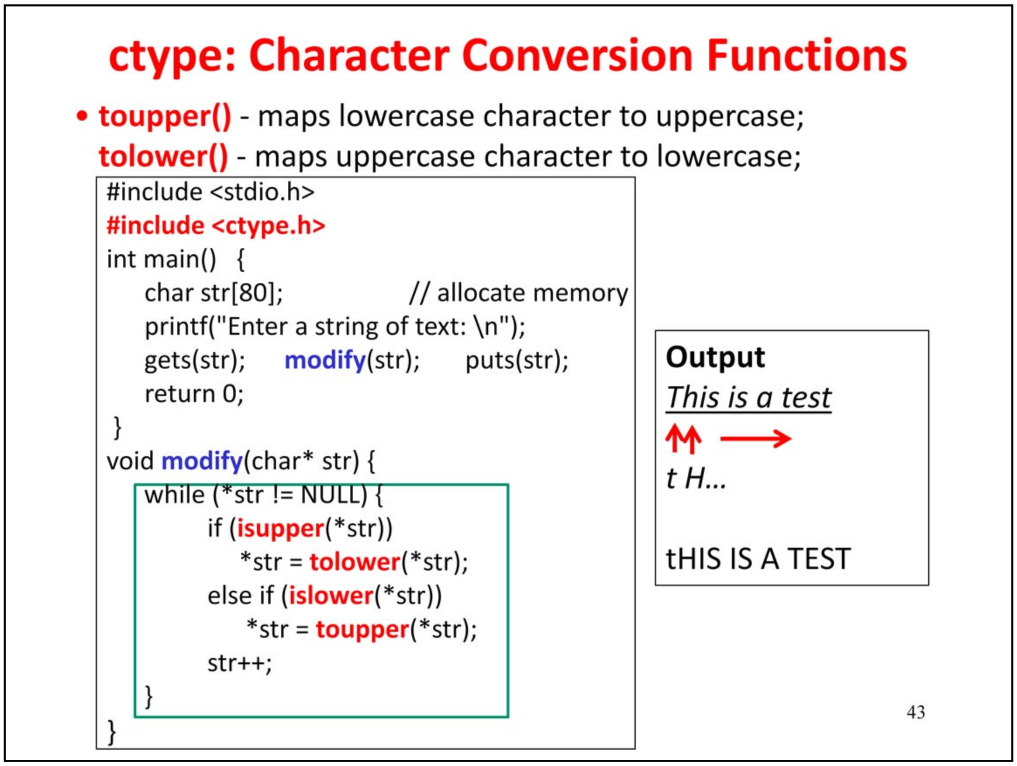
42

The ctype.h Functions: Character Conversion Functions

In addition to the functions that test characters in *ctype.h*, there are several character conversion functions for converting characters. The two most commonly used are **toupper()** and **tolower()**.

The function **toupper()** converts lowercase characters to uppercase, while the function **tolower()** converts uppercase characters to lowercase. These two functions are commonly used to test character input, and convert all of them into either lowercase or uppercase, so that the program is not sensitive to the case of the letters the user enters.

In the program, the function **modify()** aims to convert lower case letters to upper case letters and vice versa from an input string.



The ctype.h Functions: Character Conversion Functions

In the implementation of the **modify()** function, it uses the functions **isupper()**, **islower()**, **tolower()** and **toupper()** for implementing character conversion. If a character in the input string is tested to be in uppercase, it will be converted into lowercase using the function **tolower()**. Similarly, if a character in the input string is tested to be in lowercase, it will be converted to uppercase using the function **toupper()**.

Character Strings

- String Declaration, Initialization and Operations
- String Input and Output
- String Functions
- The ctype.h Character Functions
- **String to Number Conversions**
- Formatted String I/O
- Arrays of Character Strings

44

String to Number Conversions

- Must include the following header file:
  #include <stdlib.h>

atof( )

- Prototype: *double **atof** (**const** char *ptr);*
- Functionality: converts the **string** pointed to by the pointer ***ptr*** into a **double** precision floating number.
- Return value: converted value.

atoi( )

- Prototype: *int **atoi** (**const** char *ptr);*
- Functionality: *converts* the **string** pointed to by the pointer ***ptr*** *into an* ***integer****.*
- Return value: converted value.

45

String to Number Conversions

There are two ways to store a number. It can be stored as strings or in numeric form. For example, the number 123 can be stored as a string consisting of '1', '2', '3' and the terminating character '**\0**'. Sometimes, it is convenient to read in the numerical data as a string and convert it into the numeric form. To do this, C provides the functions: **atoi()** and **atof()**. To use these functions, we must include the *stdlib.h* file in the program:

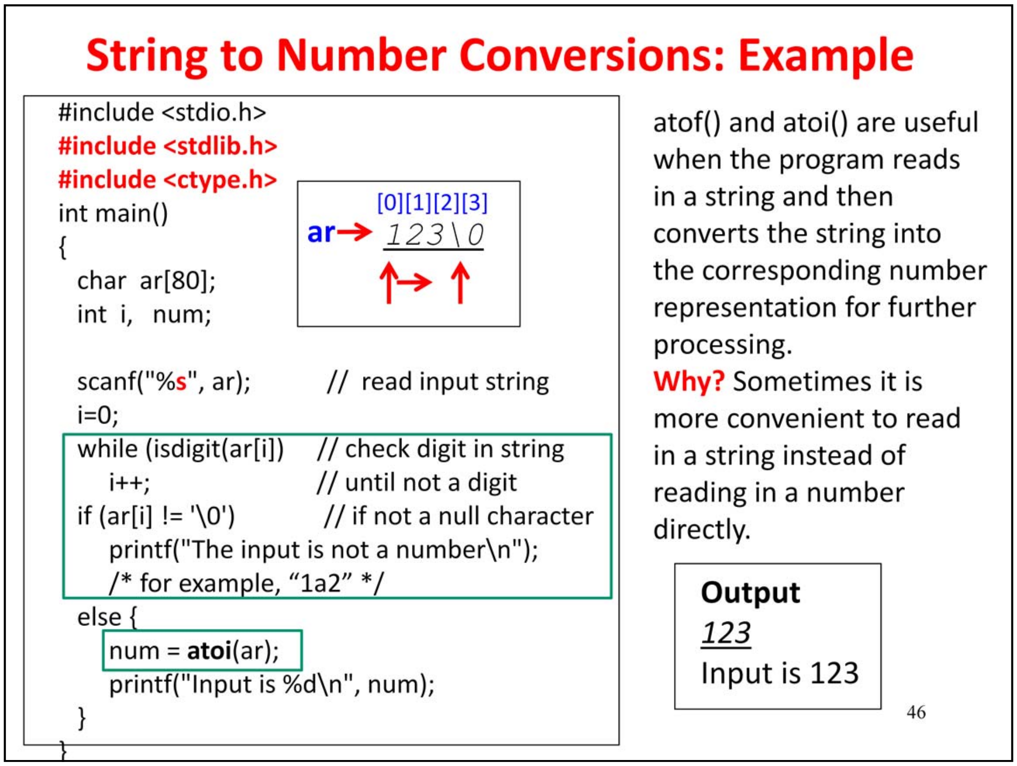
#include <stdlib.h>

The **atoi()** function converts a character string into an integer and returns the integer value. The function prototype is

int atoi(const char *ptr);

The **atoi()** function processes the digits in the string and stops when the first nondigit character is encountered. Leading blanks are ignored and leading algebraic sign (+/-) can be recognized.

The **atof()** function converts a string into a double precision floating point value.



<u>String to Number Conversions: Example</u>

In the program, it uses the **atoi()** function for string to integer number conversion. The program reads in an input string and store it in the string variable **ar**. The while loop:

while (isdigit(ar[i])

will be executed and stopped when a non-digit character is processed from the input string.

After that the statement:

if (ar[i] != '\0')

is used to check whether the next character is a null character. If yes, it means that the string contains only digit characters. If not, the string does not contain a valid integer number and an error message is printed to the screen. If the string contains valid digit characters, the function **atoi()** is called to convert the number string to the corresponding number and stores it into the variable **num**. And finally, the converted number is printed on the screen.

Character Strings

- String Declaration, Initialization and Operations
- String Input and Output
- String Functions
- The ctype.h Character Functions
- String to Number Conversions
- **Formatted String I/O**
- Arrays of Character Strings

47

Formatted String I/O

sscanf ( )

- The function *sscanf*() is similar to *scanf*(). The only difference is that *sscanf*() takes input characters from a **string** instead of from the keyboard.
- *sscanf*() can be used **to transform numbers represented in characters/strings**, i.e. "123", into numbers, i.e. 123, 123.0, of data types *int*, *float*, *double*, ..., etc.

sprintf ( )

- The function *sprintf*() is similar to printf(). The only difference is that *sprintf*() prints output to a **string**.
- *sprintf*() can be used to **transform numbers into strings**.

48

Formatted String I/O

The C standard I/O library provides two functions for performing formatted input and output to strings. These functions are **sscanf()** and **sprintf()**. The function **sscanf()** is similar to **scanf()**. The only difference is that **sscanf()** takes input characters from a string instead of from the standard input, i.e. the keyboard. It reads characters from a string and converts them into data of different types, and stores them into variables. The **sscanf()** function can be used to transform numbers represented in characters or strings (e.g. "123") into numeric numbers (e.g. 123, 123.0) of data types **int**, **float**, **double**, etc. The syntax for the function **sscanf()** is

sscanf(string_ptr, control-string, argument-list);

where **string_ptr** is a pointer to a string containing the characters to be processed. The other arguments of **sscanf()** are the same as those in **scanf()**.

The function **sprintf()** is similar to **printf()**. The only difference is that **sprintf()** prints output to a string. It formats the input data and stores them in a string. It appends a null character at the end of the string. The **sprintf()** function can be used to combine several elements into a single string. It can also be used to transform numbers into strings. The syntax for the function **sprintf()** is

sprintf(string_ptr, control-string, argument-list);

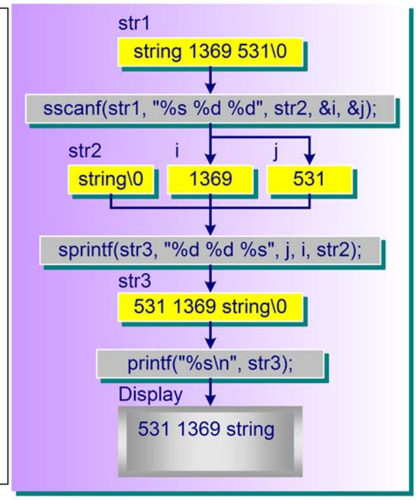
where **string_ptr** is a pointer to a string. The other arguments are the same as those in **printf()**.

Formatted String I/O - Example

```c
#include <stdio.h>
#define  MAX_CHAR 80

int main()
{
    char  str1[MAX_CHAR] = "string 1369 531";
    char  str2[MAX_CHAR], str3[MAX_CHAR];
    int     i, j;

    sscanf(str1, "%s %d %d", str2, &i, &j);
    sprintf(str3, "%d %d %s", j, i, str2);
    printf("%s\n", str3);
}
```



Output
531 1369 string

49

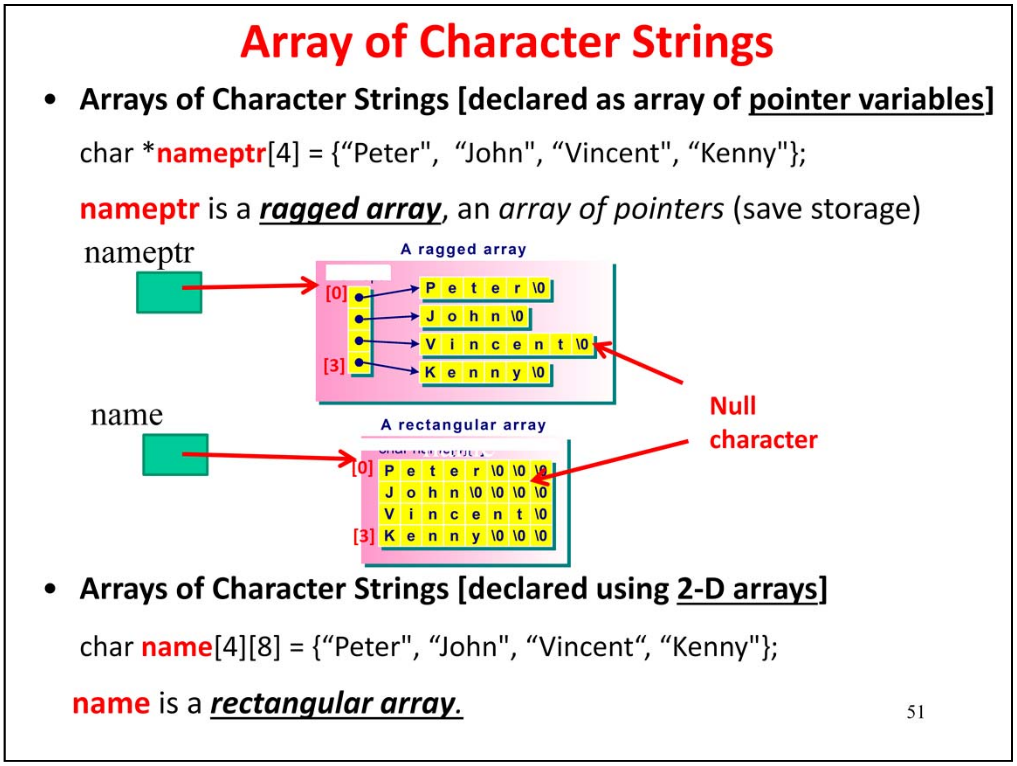
<u>Formatted String I/O - Example</u>

The example illustrates the use of the **sscanf()** and **sprintf()** functions. The **sscanf()** function is useful to convert numbers in a string to its corresponding numeric value. The **sprintf()** function is useful to combine different data items into a string. In the example, the function **sscanf()** reads from the string **str1** and stores the elements of the string into another string **str2**, and two integers, **i** and **j**. The function **sprintf()** then combines the three elements in a different order and stores them in a new string **str3**. The figure illustrates the operation of the **sscanf()** and **sprintf()** functions.

Character Strings

- String Declaration, Initialization and Operations
- String Input and Output
- String Functions
- The ctype.h Character Functions
- String to Number Conversions
- Formatted String I/O
- **Arrays of Character Strings**

50



Array of Character Strings

It is possible to define an array of character strings.

•Ragged Array

For example, we can declare

char *namePtr[4] = {"Peter", "John", "Vincent", "Kenny"};

where **namePtr** is a one-dimensional array of four pointers to type **char**. Each pointer points to the first character of the corresponding string. That is, the first pointer **namePtr[0]** points to the first character of the first string, the second pointer **namePtr[1]** points to the first character of the second string, and so on. **namePtr** is an array of pointers. This is also called a *ragged array*.

•Rectangular Array

Another way to store an array of strings is to use a two-dimensional array. For example, we can declare **name** as

char name[4][8] = {"Peter", "John", "Vincent", "Kenny"};

This is called a *rectangular array*. All the rows are of the same length. The two-dimensional array **name** needs to be defined with enough storage space to hold the longest string. Thus, some space will be wasted as not all the other strings will have the same length as the longest string. However, the ragged array declaration **namePtr** can help save storage space when compared with the rectangular array declaration.

Array of Strings: Example

```
#include <stdio.h>
int main()
{
    char *nameptr[4] = {"Peter", "John", "Vincent", "Kenny"};
    char name[4][10] = {"Mary", "Victoria", "Susan", "May"};
    int  i, j;
```

Using for loop

```
    printf("Ragged Array: \n");
    for (i=0; i<4; i++)
      printf("nameptr[%d] = %s\n", i,
          nameptr[i]);

    printf("Rectangular Array: \n");
    for (j=0; j<4; j++)
      printf("name[%d] = %s\n", j,
          name[j]);
    return 0;
 }
```

Output

```
Ragged Array:
nameptr[0] = Peter
nameptr[1] = John
nameptr[2] = Vincent
nameptr[3] = Kenny
Rectangular Array:
name[0] = Mary
name[1] = Victoria
name[2] = Susan
name[3] = Mary
```


Array of Character Strings: Example

In the program, we can process array of strings declared using ragged array or rectangular array.

After we declare an array of strings

char *namePtr[4] = {"Peter", "John", "Vincent", "Kenny"};

we can then use an index (or subscript) to access each element of the array for the character strings as follows:

```
for (i=0; i<4; i++)
        printf("namePtr[%d] = %s\n", i, namePtr[i]);
```

Similarly, we can access each element of the rectangular array declaration **name** as follows:

```
for (j=0; j<4; j++)
        printf("name[%d] = %s\n", j, name[j]);
```

Note that if we want to print a character from the string, we can use the indirection operator (i.e. '*****'). For example, the statement

printf("%c", *(namePtr[0] + 2));

prints the character '**t**' from the string "**Peter**".

Quiz: What is the output?

Quiz: What is the Output?

```
#include <stdio.h>
int main( )
{
    char *str = "abcdef";

    printf ("%c\n", *str);
    printf ("%c\n", *str++);
    printf ("%c\n", *++str);
    printf ("%c\n", *str+1);
    printf ("%c\n", *(str+1));
    return 0 ;
}
```

54

Quiz: What is the output?

Determine the output of the program.

C Operator Precedence Table

Operator	Description	Associativity
( )	Parentheses (function call)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	**Postfix increment/decrement**	
++ --	**Prefix increment/decrement**	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*****	**Dereference**	
&	**Address (of operand)**	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	

Quiz: What is the output?

To answer the quiz, you may need to understand the operator precedence table in C. Note that the precedence of dereferencing and address operators, and their relative positions when compared with the increment/decrement operators. As shown in the table, the precedence of the increment/decrement operators is higher than the dereferencing operator.

What is the Output?

```c
#include <stdio.h>
int main( )
{
    char *str = "abcdef";

    printf ("%c\n", *str);
    printf ("%c\n", *str++);
    printf ("%c\n", *++str);
    printf ("%c\n", *str+1);
    printf ("%c\n", *(str+1));
    return 0 ;
}
```

a
a
c
d
d

56

Quiz: What is the output?

The outputs are:

a

a

c

d

d

Self-Practice Exercise

57

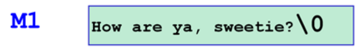
What is the output of the program?

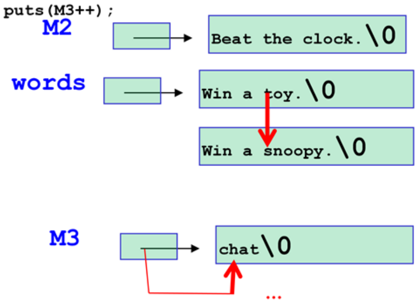
```
#include  <stdio.h>
#define  M1  "How are ya, sweetie?"
char M2[40] = "Beat the clock.";
char *M3 = "chat";
int main()
{
   char words[80];
   printf(M1);
   puts(M2);
   puts(M2+1);
   gets(words);              /* user inputs :  win a toy. */
   puts(words);
   scanf("%s", words+6);     /* user inputs :  snoopy. */
   puts(words);
   words[3] = '\0';
   puts(words);
   while (*M3)  puts(M3++);
   puts(--M3);
   puts(--M3);
   M3 = M1;
   puts(M3);
   return 0;
}
```

What is the output of the program?

Character Strings – Q4

```
#include  <stdio.h>
#define  M1  "How are ya, sweetie?"
char M2[40] = "Beat the clock.";
char *M3 = "chat";
int main()
{
   char words[80];
   printf(M1);
   puts(M2);
   puts(M2+1);
   gets(words);   /* user inputs :  win a toy. */
   puts(words);
   scanf("%s", words+6); /* user inputs :  snoopy. */
   puts(words);
   words[3] = '\0';
   puts(words);
   while (*M3)  puts(M3++);
   puts(--M3);
   puts(--M3);
   M3 = M1;
   puts(M3);
   return 0;
}
```





How are ya, sweetie?Beat the clock.
eat the clock.
win a toy.
win a toy.
snoopy.
win a snoopy.
win
chat
hat
at
t
t
at
How are ya, sweetie?

59