

6

Character Strings



Why Learning Character Strings

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

2



Character Strings

1. Here, we start by discussing string declaration, initialization and operations.

Character Strings: String Constants

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

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

Null character

- 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 University"`
- Used in function arguments, e.g. `printf()` and `puts()`: e.g. `printf("Hello, how are you?");`
- Character Constant `'X'` vs String Constant `"X"`: The character constant `'X'` consists of a single character of 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 Constants

- A character string is an array of characters terminated by a null character (`'\0'`).
- A string constant is a series of characters in double quotes, for example, `"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.
- Therefore, 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("Hello, how are you?\n");
```

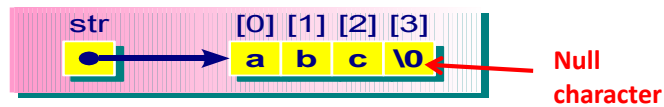
```
puts("Hello, how are you?\n");
```

- 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

(1) `char str[] = "some text";` // ok
 (2) `char str[10] = "yes";` // ok
 (3) `char str[4] = "four";` // **incorrect** -> null character missing
 (4) `char str[] = {'a','b','c','\0'};` // i.e. `char str[] = "abc";`



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

4



String Variables: Declaration using Array Notation

1. A string is essentially an array of characters. Therefore, we can also declare strings using the **array notation**.
2. To declare strings, we can use the **array notation** for declaration as shown in (1) and (2):

`char str[] = "some text";`

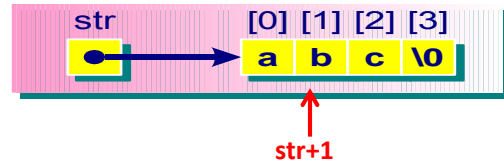
`char str[10] = "yes";`

3. However, the declaration in (3) `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.
4. We can also create strings using array initializer, for example, in (4) `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 are 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";
```



- (1) `str == &str[0]`
- (2) `*str == 'a'`
- (3) `*(str+1) == str[1] == 'b'`

5



String Variables: Declaration using Array Notation

- 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.
- For (1): **str** refers to **&str[0]** - the string **str** contains the address of the first element of the array of characters.
- For (2): ***str** refers to **'a'** - Also, we can use ***str** to retrieve the first element of the string.
- For (3): ***(str+1)** refers to **str[1]**, i.e. **'b'** - Similarly, we can use ***(str+1)** to retrieve the second element of the character string.

String Variables: Declaration using Pointer Notation

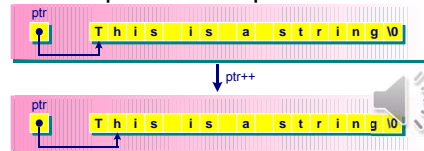
- **String variable** can also be declared using the pointer notation.
- When declaring string variable using pointer notation, we assign a string constant to a pointer that points to data type char:

```
char *str = "C Programming";
```



Null character

- 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.
- Example: `char *ptr = "This is a string";`
- For the statement: `ptr++;`
 - it means the pointer variable **ptr** will be updated and points to the next array location:



6

String Variables: Declaration using Pointer Notation

1. String variable can also be declared using the pointer notation.
2. Typically, we can declare a string variable by assigning a string constant to a **pointer variable**, for example, `char *str = "C Programming";`
3. 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 in the pointer variable;
 - terminate the string with null ('\0') character.
4. Therefore, the string variable **str** will then contain the **address** of the first element of the string constant.
5. In the example where a string constant is assigned to a pointer that points to type **char**: `char *ptr = "This is a string";`
6. For this case, the C compiler will allocate memory space to hold the string constant "This is a string", store the starting address of the string (i.e. the address of the character 'T') in the pointer variable, and terminate the string with a null character.
7. 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 Declaration 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

Q: What is the difference between the two declarations?

str1: address constant, str2: pointer variable.

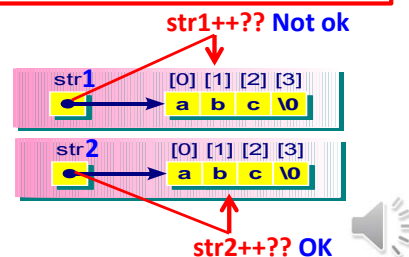
Therefore,

`++str1;` // not allowed

`++str2;` // allowed

`str1 = str2;` // not allowed

`str2 = str1;` // allowed



String Variables: Array Declaration vs Pointer Declaration

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

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

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

- 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 **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'.
- 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.
- Therefore, **str1** cannot change its value. It cannot perform any update operations. And the statements: **str1++**; and **str1 = str2**; are invalid.
- However, it is valid for **str2** to perform any update operations. Therefore: **++str2**; and **str2=str1**; are valid as **str2** is a pointer variable.

String Operations: Example

```

#include <stdio.h>
int main()
{
    char array[ ] = "pointer"; // using array notation
    char *ptr1 = "10 spaces"; // using pointer notation

    printf("ptr1 = %s\n", ptr1);
    printf("array = %s\n", array);
    array[5] = 'A';
    printf("array = %s\n", array);

    ptr1 = "OK";
    printf("ptr1 = %s\n", ptr1);

    return 0;
}

```

Diagram:

- array** points to **pointer\0**
- ptr1** points to **10 spaces\0**
- OK\0** is shown as a separate memory block.

Execution Flow:

- `printf("ptr1 = %s\n", ptr1);` → `ptr1 = 10 spaces`
- `printf("array = %s\n", array);` → `array = pointer`
- `array[5] = 'A';` → `array = pointAr`
- `ptr1 = "OK";` → `ptr1 = OK`
- `printf("ptr1 = %s\n", ptr1);` → `ptr1 = OK`

String Operations: Example

1. The declaration `char array[] = "pointer";` declares a string variable **array** using **array notation** which is initialized with 7 characters plus the null character (`'\0'`).
2. The declaration `char *ptr1 = "10 spaces";` declares a string variable **ptr1** using **pointer notation** which is initialized and pointed to the string **"10 spaces"**.
3. When **ptr1** is printed, the string **"10 spaces"** will be displayed.
4. When **array** is printed, the string **"pointer"** will be displayed.
5. The statement `array[5] = 'A';` is valid and the string is updated accordingly.
6. When **array** is printed, the string **"pointAr"** will be displayed.
7. The statement `ptr1 = "OK";` will update **ptr1** to point to a new string **"OK"**.
8. When **ptr1** is printed, the string **"OK"** will be displayed.

String Operations: Example (Cont'd.)

```

#include <stdio.h>
int main()
{
    char array[] = "pointer"; //using array
    char *ptr1 = "10 spaces"; // using pointer

    printf("ptr1 = %s\n", ptr1);
    printf("array = %s\n", array);
    array[5] = 'A';
    printf("array = %s\n", array);
    ptr1 = "OK";
    printf("ptr1 = %s\n", ptr1);
    ptr1 = array;
    printf("ptr1 = %s\n", ptr1);
    ptr1[5] = 'C';
    printf("ptr1 = %s\n", ptr1);
    ptr1 = "A new string";
    printf("ptr1 = %s\n", ptr1);
    return 0;
}

```

array → pointAr\0

ptr1 → A new string\0

ptr1 = 10 spaces
 array = pointer
 array = pointAr
 ptr1 = OK
 ptr1 = pointAr
 ptr1 = pointCr
 ptr1 = A new string

String Operations: Example

1. The statement **ptr1 = array;** changes the pointer variable **ptr1** to point to the same address contained in **array**.
2. 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**.
3. When **ptr1** is printed, the string "**pointAr**" will be displayed.
4. The statement **ptr1[5] = 'C';** is valid and the string is updated accordingly.
5. When **ptr1** is printed, the string "**pointCr**" will be displayed.
6. 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

10



Character Strings

1. Here, we discuss Character String Input and Output.

String Input/Output

- There are 4 C library functions that can be used for string input/output:
 - **fgets()** (instead of gets()): function prototype **char *fgets(char *ptr, int n, FILE *stream);**
 - **puts()**: function prototype **int puts(const char *ptr);**
 - **scanf()**: function prototype **int scanf(control-string, argument-list);**
 - **printf()**: function prototype **int printf(control-string, argument-list);**
- The two most commonly used standard library functions for reading strings are **fgets()** and **scanf()**. For printing strings, the two standard library functions are **puts()** and **printf()**.
- We use **fgets()** because **gets()** is not safe as it does not check the array bound.



11

String Input/Output

1. There are 4 C library functions that can be used for string input/output.
2. The two most commonly used standard library functions for reading strings are **fgets()** and **scanf()**.
3. For printing strings, the two standard library functions are **puts()** and **printf()**.
4. We use **fgets()** instead of **gets()** because **gets()** is unsafe as it does not check the array bound.
5. In **fgets()**, we use **stdin** as the argument for **stream** when reading from the keyboard.

String Input: fgets()

- **fgets()** returns **Null** if it fails, otherwise a pointer to the string is returned.
- Make sure **enough memory space** is allocated to hold the input string, **name**.

```
#include <stdio.h>
#include <string.h>
int main()
{
    char name[80], *p; // allocate memory
    /*read name*/
    printf("Hi, what is your name?\n");
    fgets(name, 80, stdin);
    if ( p=strchr(name, '\n') )
        *p = '\0'; // remove '\n' character in name
    /*display name*/
    printf("Nice name, %s.\n", name);
    return 0;
}
```

name

Hui Siu Cheung\0

**Question: using
pointer variable:
char *name;
Ok or not? Why?**

name → ?? Not OK!

Output:
Hi, what is your name?
Hui Siu Cheung
Nice name, Hui Siu Cheung.



String Input: fgets()

1. 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 declaration: **char name[80];**
2. 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.
3. The function **fgets()** gets a string from the standard input device. In **fgets(name, 80, stdin);** it reads characters until it reaches a newline character (**\n**). A newline character is generated when the **<Enter>** key is pressed. The function **fgets()** reads all the characters up to and including the newline character.
4. 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.
5. As **fgets()** reads the entire new line including the newline character (**'\n'**), we need to execute the function **strchr()** with the statement **if (p=strchr(name, '\n')) *p = '\0';** to replace the newline character (**'\n'**) by a null character (**'\0'**) to form a character string. The function **strchr()** finds the newline character **'\n'**, which is then replaced by the null character **'\0'**. As such, we will get the character string accordingly.

6. Then, the string **“Hui Siu Cheung”** is printed on the screen.
7. Note that we cannot use pointer notation for declaring the string as there will not have memory allocated for holding the input string data.

String Output: puts()

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

Input: 0123456789 OK

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

Output: 0123456789 OK



String Output: puts()

1. The **puts()** function 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.
2. **EOF** is returned if **puts()** fails, otherwise the number of characters written will be returned.
3. In the program, it reads in a string using the **fgets()** function and prints the string on the screen using the **puts()** function.

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

- scanf()
 - It 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()
 - It **returns** the number of characters transmitted, otherwise a negative value will be returned if it fails.
 - 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.



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

1. The **scanf()** function can be used to read in a string. It will return **EOF** if it fails, otherwise the number of items read by **scanf()** will be returned. The **scanf()** function reads the string up to the next **whitespace** character.
2. It is important to ensure that **sufficient storage** is allocated to hold the string.
3. The **printf()** function 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.
4. It differs from the **puts()** function in that **no newline** is added at the end of the string.
5. The **printf()** function is less convenient to use than the **puts()** function.
6. However, the **printf()** function provides the flexibility to the user to control the format of the data to be printed.

scanf() and printf(): Example

```
#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



scanf() and printf(): Example

1. In the program, the **scanf()** function is used to read a string from the user 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.
2. Note that the **fgets()** function differs from the **scanf()** function in that **fgets()** reads an entire line up to the first newline character, and it also stores any characters, including whitespace and the first newline character.
3. The **scanf()** function is less convenient to use than the **fgets()** function for reading string input. The **fgets()** function is also faster than the **scanf()** function.

String Processing – Using Indexes

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


Output
The length is 5

greeting → Hello\0
word → abc\0

using index notation

```
int length1(char string[]) // int length1(char *string)
{
    int count = 0;
    while (string[count] != '\0')
        count++;
    return(count);
}
```

string →



String Processing: Using Indexes

1. In this program, the function **length1()** uses the array notation to compute the length of a string.
2. 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"**.
3. 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 – Using Pointers

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

Output
The length is 3


greeting → Hello\0

word → abc\0

string → []

using pointer notation

```
int length2(char *string) // int length2(char string[])
{
    int count = 0;
    while ( *(string+count) != '\0' )
        count++;
    return(count);
}
```



String Processing: Using Pointers

1. In this program, the function **length2()** uses the pointer notation to compute the length of a string.
2. The declaration **char word[] = "abc";** creates an array of type **char** called **word[]**. This array contains four elements including the null character.
3. 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

18



Character Strings

1. Here, we discuss some commonly used string functions.

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
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

String Functions

1. 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>**.
2. 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. The table lists some of the more commonly used string functions provided in **string.h**.
3. Here, we describe some of the most useful C string handling functions. These include **strlen()**, **strcat()**, **strcpy()**, and **strcmp()**.
 - 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 strlen() Function

- The function prototype of **strlen** is

size_t **strlen**(const char *str);

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



The strlen() Function

- The function prototype of **strlen()** is: **size_t strlen(const char *str);** where **size_t** refers to unsigned integer type.
- 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.
- In the program, it uses the function **strlen()** 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 **strlen()**, 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 ***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 value of **str1** (i.e. string).

- Example**

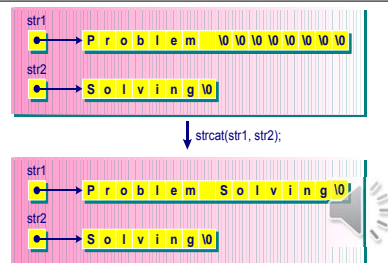
```
#include <stdio.h>
#include <string.h>
int main() {
    char str1[40] = "Problem ";
    char *str2 = "Solving";
    printf("The first string: %s\n", str1);
    printf("The second string: %s\n", str2);
    strcat(str1, str2);
    printf("The combined string: %s\n", str1);
    return 0;
}
```

Output

The first string: Problem

The second string : Solving

The combined string: Problem Solving



The strcat() Function

- The function **strcat()** 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.
- In the program, it uses **strcat()** to concatenate two strings **str1** and **str2**. The two strings are declared and initialized:

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 *str1, const char *str2);
```

strcpy copies the string pointed to by **str2** into the array pointed to by **str1**.

It **returns** the value of **str1** (i.e. string).

- Example

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

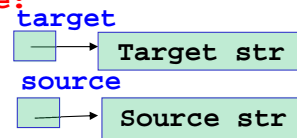
```
#include <string.h>
```

```
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;
}
```

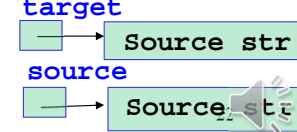
Output

```
This is the target string.
This is the source string.
This is the source string.
This is the source string.
```

Before:



After:



The strcpy() Function

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

```
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 function **strcpy()** returns the value of **str1**.
- In the program, it uses **strcpy()** 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 array declaration **char target[40];** is used instead of pointer declaration **char *target;** It is because the latter declaration does not have space allocated to hold the string.

The strcmp() Function

- The function prototype of **strcmp** is

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

strcmp compares the string pointed to by str1 to the string pointed to by str2.
- It **returns** an integer >, =, or < zero, accordingly if the string pointed to by str1 is >, =, or < the string pointed to by str2:
 - **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)



The strcmp() Function

1. The function **strcmp()** compares the contents of two strings. The prototype of **strcmp()** is shown as: `int strcmp(const char *str1, const char *str2);`
2. 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.
3. 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.

The strcmp() Function: String Comparison based on ASCII Values

	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		

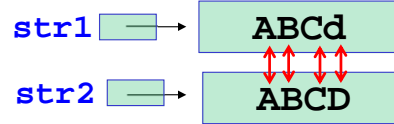
The strcmp() Function: String Comparison

1. The comparison between characters in the two strings is based on ASCII values.
2. 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.
3. 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.
4. 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.
5. This way of ordering strings is called *lexicographic order*.

The strcmp() Function: Example 1

```
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[81], str2[81], *p;
    int result;
    printf("String Comparison:\n");
    printf("Enter the first string: ");
    fgets(str1, 81, stdin);
    if ( p=strchr(str1,'\n') ) *p = '\0';
    printf("Enter the second string: ");
    fgets(str2, 81, stdin);
    if ( p=strchr(str2,'\n') ) *p = '\0';
    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: E
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: AQ
Enter the second string: AQ
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.

The strcmp() Function: Example 1

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

The strcmp() Function: Example 2

```

/* 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], *p;
    fgets(line, 132, stdin);
    if ( p=strchr(line, '\n') ) *p = '\0';
    while ( strcmp(line, "END") !=0 ) {
        reverse(line);
        printf("%s\n", line);
        fgets(line, 132, stdin);
        if ( p=strchr(line, '\n') ) *p = '\0';
    }
}

```

```

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--;
    }
}

```

The strcmp() Function: Example 2

1. In the program, it uses the function **strcmp()** to check whether to exit the loop after reading in an input string. The loop will end if the input string is "END".
2. 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".
3. In the **reverse()** function, it accepts an input string as its parameter. The function then determines the position of the **end** pointer with the statement: **end = s + strlen(s) - 1;**
4. 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.
5. 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 woH".
6. Note that the function **reverse()** 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

27



Character Strings

1. Here, we discuss the **ctype.h** character functions.

Name	True If Argument is	ctype.h Functions <ul style="list-style-type: none"> • Return true (non-zero) if the character belongs to a particular class; • Return false (zero) otherwise. • Must include the header file: <div style="border: 1px solid red; padding: 2px; display: inline-block;"> #include <ctype.h> </div> 
isalnum	Alphanumeric (alphabetic or numeric)	
isalpha	Alphabetic	
isctrl	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	
isxdigit	A hexadecimal-digit character	

The ctype.h Functions

1. C also contains the character processing library, whose functions are declared in the **ctype.h** header file.
2. These functions are used to test the nature of a character. It returns **true** if the condition being tested is satisfied, or **false** otherwise.
3. 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.
4. 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 <string.h>
#include <ctype.h>
void modify(char* str);
int main() {
    char str[80], *p;           // allocate memory
    printf("Enter a string of text: \n");
    fgets(str,80,stdin); if ( p=strchr(str,'\n') ) *p = '\0';
    modify(str); puts(str);
    return 0;
}
void modify(char* str) {
    while (*str != '\0') {
        if (isupper(*str))
            *str = tolower(*str);
        else if (islower(*str))
            *str = toupper(*str);
        str++;
    }
}
```

Output

This is a test

↑↑ →
t H...

THIS IS A TEST



ctype.h: Character Conversion Functions

1. 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 functions are **toupper()** and **tolower()**.
2. 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.
3. In the program, the function **modify()** aims to convert lower case letters to upper case letters and vice versa from an input string. It uses the functions **isupper()**, **islower()**, **tolower()** and **toupper()** for implementing character conversion. In the **while** loop, 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

30



Character Strings

1. Here, we discuss the string to number conversion functions.

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.



String to Number Conversions

1. There are two ways to store a number. It can be stored as strings or in numeric form. 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()**.
2. To use these functions, we must include the `stdlib.h` file in the program:
#include <stdlib.h>
3. The **atoi()** function converts a character string into an integer and returns the integer value. The function processes the digits in the string and stops when the first non-digit character is encountered. Leading blanks are ignored and leading sign (+/-) can be recognized.
4. The **atof()** function converts a string into a double precision floating point value.

String to Number Conversions: Example

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

int main()
{
    char ar[80];
    int i, num;

    scanf("%s", ar);    // read input string
    i=0;
    while (isdigit(ar[i]) // check digit in string
           i++;          // until not a digit
    if (ar[i] != '\0')    // if not a null character
        printf("The input is not a number\n");
        /* for example, "1a2" */
    else {
        num = atoi(ar);
        printf("Input is %d\n", num);
    }
}
```

[0][1][2][3]
ar → 1 2 3 \0
↑ → ↑

atoi() and **atoi()** are useful when the program reads in a string and then converts the string into the corresponding number representation for further processing.

Why? Sometimes it is more convenient to read in a string instead of reading in a number directly.

Output

123
Input is 123



String to Number Conversions: Example

1. **atoi()** and **atoi()** are useful when the program reads in a string and then converts the string into the corresponding number representation for further processing, as sometimes it is more convenient to read in a string instead of reading in a number directly.
2. 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**.
3. The while loop: **while (isdigit(ar[i])** will be executed and stopped when a non-digit character is processed from the input string.
4. 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 will be printed to the screen.
5. 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**.
6. 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

33



Character Strings

1. Here, we discuss the formatted string I/O functions.

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.
- Function prototype:

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

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**.
- Function prototype:

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



Formatted String I/O

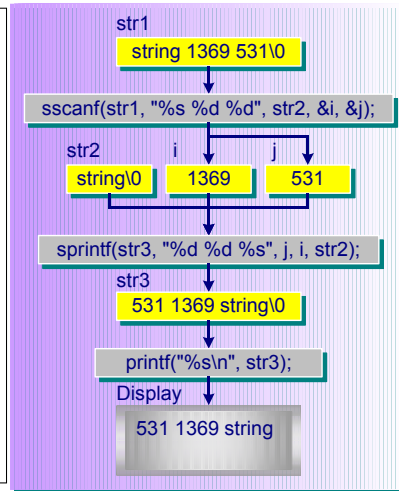
1. The C standard I/O library provides two functions for performing formatted input and output to strings. These functions are **`sscanf()`** and **`sprintf()`**.
2. The function **`sscanf()`** is similar to **`scanf()`**. The only difference is that **`sscanf()`** takes input characters from a string 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.
3. 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()`**.
4. 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.
5. 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

```
#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



Formatted String I/O – Example

1. The program illustrates the use of the **sscanf()** and **sprintf()** functions.
2. The **sscanf()** function is useful to convert numbers in a string to its corresponding numeric value. It is useful to combine different data items into a string.
3. In the program, 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**.

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**

36



Character Strings

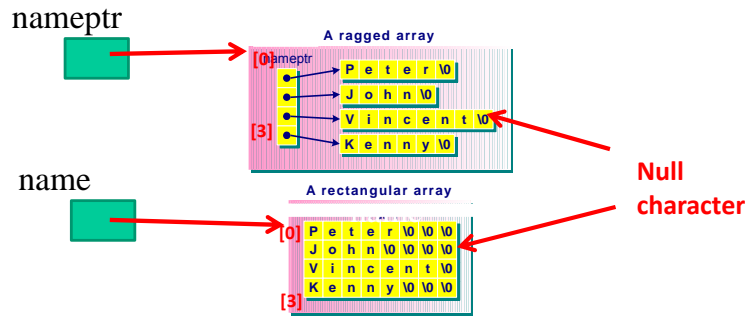
1. Here, we discuss array of character strings.

Array of Character Strings

- Arrays of Character Strings [declared as array of pointer variables]

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

nameptr is a ragged array, an array of pointers (save storage)



- Arrays of Character Strings [declared using 2-D arrays]

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

name is a rectangular array.



Array of Character Strings

1. It is possible to define an array of character strings.
2. For example, we can declare an array of four strings: **char *namePtr[4] = {"Peter", "John", "Vincent", "Kenny"};** where **namePtr** is an one-dimensional array of four pointers to type **char**. This is also called a **ragged array**.
3. 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.
4. Another way to store an array of strings is to use a two-dimensional array. For example, we can declare the two-dimensional array **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.
5. Therefore, the ragged array declaration **namePtr** can help save storage space when compared with the rectangular array declaration.

Array of Character 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

1. In the program, we can process array of strings declared using ragged array or rectangular array.
2. After we declare an array of strings **char *namePtr[4] = {"Peter", "John", "Vincent", "Kenny"};** we can then use an index (or subscript) to access and print each element of the array for the character strings with a **for** loop:

```
for (i=0; i<4; i++)
```

```
    printf("namePtr[%d] = %s\n", i, namePtr[i]);
```

3. Similarly, we can access and print each element of the rectangular array declaration **name** with a **for** loop:

```
for (j=0; j<4; j++)
```

```
    printf("name[%d] = %s\n", j, name[j]);
```

4. 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".

Thank you !!!

