

Week 5
Character Strings
(Summary on Key Points)

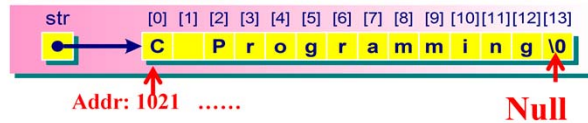
Character Strings

- Character Strings and Arrays
- Character Strings and Pointers
- String Functions and Character Functions
- Application (1) – strlen()
- Application (2) – strreverse()
- Application (3) – strncpy()
- Application (4) – strcmp()
- Application (5) – processing array of strings

2

Character Strings and Arrays

- A **string** is an **array of characters** terminated by a **NULL** ('\0') character,
e.g. `char str[] = "C Programming";`



- **str?**
=> pointer - containing the address (or pointer to) of the **1st element of the string** (i.e. 1021)
- **&str?**
=> memory address of the string variable (e.g. 1234)
- ***str?**
=> dereferencing the array to get the value (i.e. str[0], the value of 'C')
- **str[0], str[1], str[2], etc.**
=> the char value stored at the resp. array index locations, i.e. 'C', ' ', etc.
- **&str[0], &str[1], &str[2], etc.**
=> the memory addresses of the memory locations of the respective array index locations

Character Strings and Pointers: Declaration

- There are two ways to declare a string:

`char str1[] = "abc";` // using array declaration

and

`char *str2 = "abc";` // using pointer declaration

- str1: address (pointer) constant**

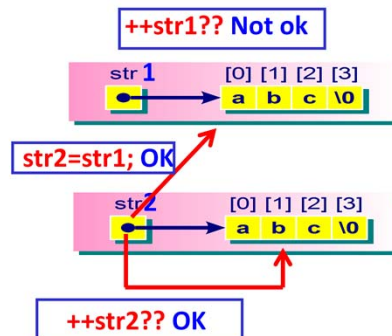
`++str1;` // not ok

`str1 = str2;` // not ok

- str2: pointer variable**

`++str2;` // ok

`str2 = str1;` // ok



4

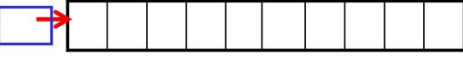
String Variables: Declaration

- The **str1** declaration creates an array of type **char**. The array has been allocated with memory to hold 4 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 'a'.
- 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. Therefore, the following 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 Functions + Character Functions

- **String functions** #include <string.h>: strlen(), strcat(), strcmp(), strcpy()..
- **Character functions** #include <ctype.h> : isdigit(), isupper(), islower()..
- **String to number functions** #include <stdlib.h>: atof(), atoi() ..
- **Formatted string I/O functions** #include <stdio.h>: sscanf(), sprintf() ..
- **Note when creating a character string:**

```
#include <stdio.h>
int main()
{
    char name[80]; // to allocate memory
    printf("Hi, what is your name?\n");
    gets(name);
    printf("Nice name, %s.\n", name);
    return 0;
}
```

name 

Question:
using pointer variable:

char *name1;

Ok or not? Why?

**=> Not OK, no memory
is allocated for storing
the name**

name1 

String Functions and Character Functions

1. String functions #include <string.h>: gets(), puts(), scanf(), printf(), strlen(), strcat(), strcmp(), strcpy(), etc.
2. Character functions #include <ctype.h> : isdigit(), isupper(), islower(), etc.
3. String to number functions #include <stdlib.h>: atof(), atoi(), etc.
4. Formatted string I/O functions #include <stdio.h>: sscanf(), sprintf(), etc.

Example

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 following 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 **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.

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. Note that we cannot use pointer notation for creating the string as there will not have memory allocated for holding the input string data.

Application (1) – String Length

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

Call by reference

using **index** notation

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

string

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

Output

The length is 5, 3

greeting

→

Hello\0

word

→

abc\0

6

Application (1) – String Length

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

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

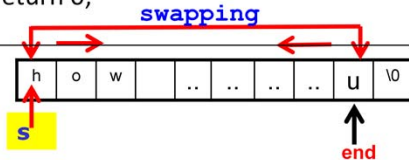
while loop will be executed to measure the length of the string.

Application (2): Reverse String

```
/* 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 *s);
int main(){
    char line[132];
    gets(line);
    while (strcmp(line, "END") != 0) {
        reverseString(line);
        printf("%s\n", line);
        gets(line);
    }
    return 0;
}
```

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



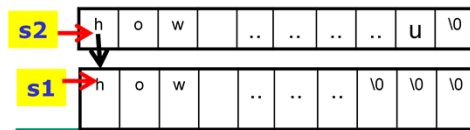
```
How are you
s-->          <--end
uoy era woH
END
```

7

Application (2) – Reverse String

1. 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".
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 following 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 who".
6. Note that the **reverse()** function illustrates a typical example on string processing.

Application (3): String Copy



```
char *stringncpy(char *s1, char *s2, int n)
{
    int k, h;

    for (k = 0; k < n; k++) {
        if (s2[k] != '\0')
            s1[k] = s2[k];
        else
            break;
    }
    s1[k] = '\0';
    // to append '\0' if n>s2 length
    for (h = k; h < n; h++)
        s1[h] = '\0';
    return s1;
}
```

Using index

```
char *stringncpy2(char *s1, char *s2,
int n)
{
    int k, h;

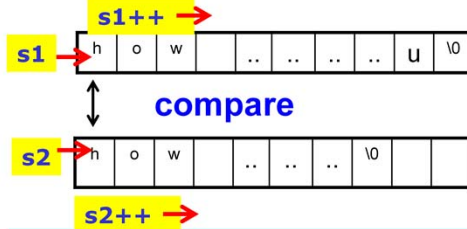
    for (k = 0; k < n; k++) {
        if (*(s2+k) != '\0')
            *(s1+k) = *(s2+k);
        else
            break;
    }
    *(s1+k) = '\0';
    // append '\0'
    for (h = k; h < n; h++)
        s1[h] = '\0';
    return s1;
}
```

Using Pointer

Application (3) – String Copy

1. In the function, it copies the string content from **s2** to **s1** according to the specified size **n**.
2. The function uses a **for** loop to traverse the string when copying the contents from one to another.

Application (4): String Compare



```

/* return 0 if the two strings (based on
ASCII values) are the same;
return 1 or -1 if one string is larger/smaller
than another string in alphabetical order*/
#include <stdio.h>
int strcmp(char * s1, char * s2);
int main()
{
    char source[80], target[80];
    gets(source);
    gets(target);
    printf("strcmp=%d", strcmp(source, target));

    return 0;
}

```

```

int strcmp(char *s1, char *s2)
{
    while (1) {
        if (*s1 == '\0' && *s2 == '\0')
            return 0;
        else if (*s1 == '\0')
            return -1;
        else if (*s2 == '\0')
            return 1;
        else if (*s1 < *s2)
            return -1;
        else if (*s1 > *s2)
            return 1;

        s1++;
        s2++;
    }
}

```

Comparison based on ASCII value

Application (4) – String Compare

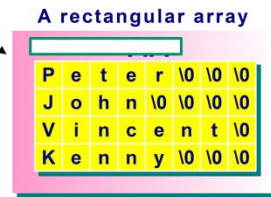
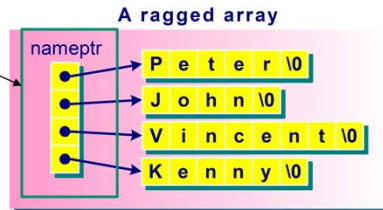
1. In the function, it compares two strings and returns the comparison result to the calling function.
2. This is done based on the comparison of the ASCII values of each character, character by character, in the two character strings **s1** and **s2**.

Application (5): Array of Strings

```
#include <stdio.h>
int main() {
    char name[4][10]; // NB: using array declaration;
    int i,j;           // do not use pointer declaration: char *nameptr[4]

    printf("Enter 4 names: ");
    for (i=0; i<4; i++) {
        scanf("%s", name[i]);
    }

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



```
Enter 4 names: Peter John Vincent Kenny
Rectangular Array:
name[0] = Peter
name[1] = John
name[2] = Vincent
name[3] = Kenny
```

Application (5): Array of Character Strings

1. Note that if we read input string data from the user, we will need to use the array declaration in order to store the input string data. This will enable the allocation of memory to store the input string data.
2. If we use the pointer declaration, no memory will be allocated to store the input string data. The program will be incorrect.