Strings (2)

Program Design (II)

2022 Spring

Fu-Yin Cherng
Dept. CSIE, National Chung Cheng University

Outline

- Reading and Writing Strings
- Accessing the Characters in a String
- Using the C String Library

Reading and Writing Strings

- Writing a string is easy using either `printf` or `puts`.
- Will explain `puts` function later

```
printf("hello world");
puts("hello world");
```

Reading and Writing Strings

- Reading a string is a bit harder, because the input may be longer than the string variable into which it's being stored.
- To read a string in a single step, we can use either `scanf` or `gets`.
  - However, `gets` is not recommended, which we will explain later
- As an alternative, we can read strings one character at a time.

Reading and Writing Strings

- The `%s` conversion specification allows `printf` to write a string:
- `printf` writes the characters in a string one by one until it encounters a null character (`\0`).

```
char str[] = "Are we having fun yet?";

printf("%s\n", str);
```

```
Are we having fun yet?
```

Reading and Writing Strings

- The `%`*m*`s` conversion will display a string in a field of size *m*.
- If the string has fewer than *m* characters, it will be right-justified within the field.

```
char str[] = "ABCD";

char str2[] = "ABC";

printf("%4s\n", str);
printf("%4s\n", str2);
```

```
ABCD
 ABC
```

Reading and Writing Strings

- To force left justification instead, we can put a minus sign in front of *m*.

```
char str[] = "ABCD";

char str2[] = "ABC";

printf("%-4s\n", str);
printf("%-4s\n", str2);
```

```
ABCD
ABC
```

Reading and Writing Strings

- The *m* and *p* values can be used in combination.
- A conversion specification of the form `%m.ps` causes the first *p* characters of a string to be displayed in a field of size *m*.

```
char str[] = "ABCD";

char str2[] = "ABC";

printf("%4s\n", str);
printf("%4.3s\n", str);
printf("%4.2s\n", str2);
```

```
ABCD
 ABC
  AB
```

Writing Strings Using `printf` and `puts`

- `printf` isn't the only function that can write strings.
- The C library also provides `puts`
- After writing a string, `puts` always writes an additional new-line character (`\n`).

```
puts("ABCD");
puts("ABCD");

printf("abc");
printf("abc");
```

```
ABCD
ABCD
abcabc
```

Reading Strings Using `scanf` and `gets`

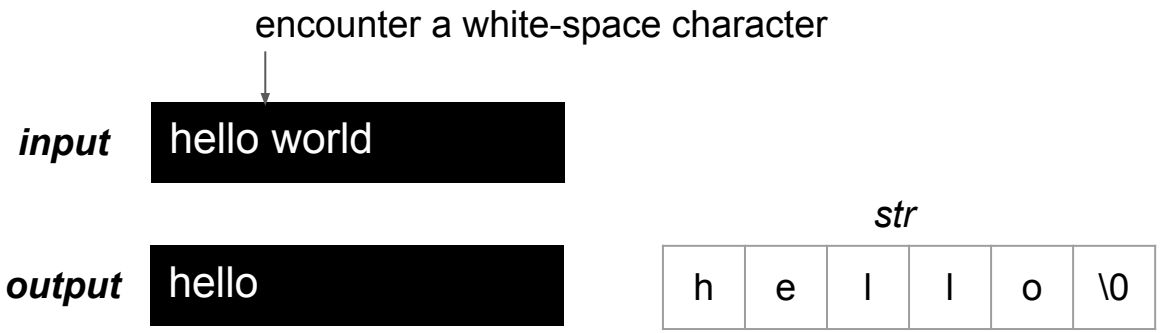
- The `%s` conversion specification allows `scanf` to read a string into a character array:
- Why we don't need to add `&` in front of `str` in the call of `scanf`?

```
char str[5];
scanf("%s", str);
```

Reading Strings Using `scanf` and `gets`

- When `scanf` is called, it skips white space, then reads characters and stores them in `str` until it encounters a white-space character.
- `scanf` always stores a null character at the end of the string.

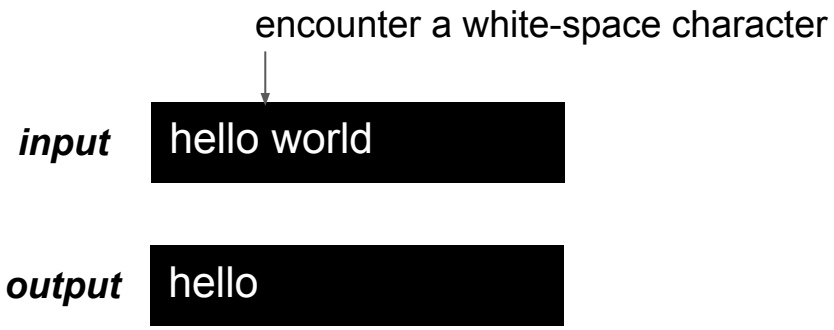
```
char str[20];
scanf("%s", str);
puts(str);
```



Reading Strings Using `scanf` and `gets`

- `scanf` won't usually read a full line of input.
- A new-line/space/tab character will cause `scanf` to stop reading
- To read an entire line of input, we can use `gets`.

```
char str[20];
scanf("%s", str);
puts(str);
```



Reading Strings Using `scanf` and `gets`

- Properties of `gets`:
  - Doesn't skip white space before starting to read input.
  - Reads until it finds a new-line character (`\n`).
    - the end of one line
  - Discards the new-line character instead of storing it
    - the null character (`\0`) takes its place

Reading Strings Using `scanf` and `gets`

- Consider the following program fragment
- we can store the entire "`hello world`" string literal in `str`

```
char str[20];
gets(str);
puts(str);
```

input hello world

output hello world

Reading Strings Using `scanf` and `gets`

- However!
- As they read characters into an array, `scanf` and `gets` have no way to detect when it's full.
- They may store characters past the end of the array, causing undefined behavior.

```
char str[5];
gets(str);
puts(str);
```

input

```
hello world
```

```
*** stack smashing detected ***: terminated
```

Reading Strings Using `scanf` and `gets`

- `scanf` can be made safer by using the conversion specification `%`*n*`s` instead of `%s`.
- *n* is an integer indicating the maximum number of characters to be stored.
  - need to reserve one character for the null character at the end of the string

```
char str[5];
scanf("%4s", str);
puts(str);
```

input hello world

output hell

Reading Strings Using `scanf` and `gets`

- `gets` is inherently unsafe because we cannot control the number of character it read
- `fgets` is a much better alternative.
  - https://stackoverflow.com/questions/1694036/why-is-the-gets-function-so-dangerous-that-it-should-not-be-used
  - https://www.geeksforgeeks.org/gets-is-risky-to-use/

```
char str[5];
fgets(str, 5, stdin);
printf("%s", str);
```

input hello world

output hell

Reading Strings Using `scanf` and `gets`

- `gets` is inherently unsafe because we cannot control the number of character it read
- `fgets` is a much better alternative.
  - https://stackoverflow.com/questions/1694036/why-is-the-gets-function-so-dangerous-that-it-should-not-be-used
  - https://www.geeksforgeeks.org/gets-is-risky-to-use/
- However, we need to add `\0` manually if using `fgets`

```
…

//compute the length of string in str (introduce later)
int len = strlen(str);

// Remove the '\n' character and replace it with '\0'
str[len - 1] = '\0';
```

Reading Strings Character by Character

- Because there are so many restrictions of `scanf`, `gets`, `fgets`
- Programmers often write their own input functions.
- Issues to consider:

Reading Strings Character by Character

- Because there are so many restrictions of `scanf`, `gets`, `fgets`
- Programmers often write their own input functions.
- Issues to consider:
  - Should the function skip white space before beginning to store the string?

Let’s Take a Break!

Reading Strings Character by Character

- Because there are so many restrictions of `scanf`, `gets`, `fgets`
- Programmers often write their own input functions.
- Issues to consider:
  - Should the function skip white space before beginning to store the string?
  - What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?

Reading Strings Character by Character

- Because there are so many restrictions of `scanf`, `gets`, `fgets`
- Programmers often write their own input functions.
- Issues to consider:
  - Should the function skip white space before beginning to store the string?
  - What character causes the function to stop reading: a new-line character, any white-space character, or some other character? Is this character stored in the string or discarded?
  - What should the function do if the input string is too long to store: discard the extra characters or leave them for the next input operation?

Reading Strings Character by Character

- Suppose we need a function that
  - (1) doesn't skip white-space characters,
  - (2) stops reading at the first new-line character (which isn't stored in the string)
  - (3) discards extra characters.
- A prototype for the function

```
int read_line(char str[], int n);
```

Reading Strings Character by Character

- If the input line contains more than `n` characters, `read_line` will discard the additional characters.
- `read_line` will return the number of characters it stores in `str`.

```
int read_line(char str[], int n);
```

Reading Strings Character by Character

- `read_line` consists primarily of a loop that calls `getchar` to read a character and then stores the character in `str`, provided that there's room left:

```
int read_line(char str[], int n)
{
     int ch, i = 0;

     while ((ch = getchar()) != '\n')
       if (i < n)
         str[i++] = ch;
     str[i] = '\0';   /* terminates string */
     return i;        /* number of characters stored */
}
```

Reading Strings Character by Character

- Before returning, `read_line` puts a null character at the end of the string.

```
int read_line(char str[], int n)
{
     int ch, i = 0;

     while ((ch = getchar()) != '\n')
       if (i < n)
         str[i++] = ch;
     str[i] = '\0';    /* terminates string */
     return i;         /* number of characters stored */
}
```

Reading Strings Character by Character

- `scanf` and `gets` automatically put a null character at the end of an input string.
- If we're writing our own input function, we must take on that responsibility!

```
int read_line(char str[], int n)
{
     int ch, i = 0;

     while ((ch = getchar()) != '\n')
       if (i < n)
         str[i++] = ch;
     str[i] = '\0';    /* terminates string */
     return i;         /* number of characters stored */
}
```

Accessing the Characters in a String

- Since strings are stored as arrays, we can use subscripting to access the characters in a string.
- To process every character in a string `s`, we can set up a loop that increments a counter `i` and selects characters via the expression `s[i]`.

```
int count_spaces(const char s[]){
     int count = 0, i;

     for (i = 0; s[i] != '\0'; i++)
       if (s[i] == ' ')
         count++;
     return count;
}
```

Accessing the Characters in a String

- A version that uses pointer arithmetic instead of array subscripting
- How to rewrite this program?

```
int count_spaces(const char *s){
     int count = 0, i;

     for (; _(1)_ != '\0'; _(2)_)

        if (_(3)_ == ' ')
          count++;
     return count;
}
```

slido

How to rewrite this program? (Please remember to enter your student ID or name)

ⓘ Start presenting to display the poll results on this slide.

Accessing the Characters in a String

- So, now we have two version of accessing the character in a string
- Which one is better? array operation or pointer operation?
- You may have the questions like
  - Is it better to use array operations or pointer operations to access the characters in a string?

Ans: We can use either or both. Traditionally, C programmers lean toward using pointer operations.

Accessing the Characters in a String

- Should a string parameter be declared as an array or as a pointer?
- Ans: no difference between the two for arguments
  - "ABC" can be use as an argument for these two versions

```
int count_spaces(const char s[]){
    …
     return count;
}
```

```
int count_spaces(const char *s){
    …
     return count;
}
```

Using the C String Library

- Some programming languages provide operators that can copy strings, compare strings, concatenate strings, select substrings, and the like.
- For example, in python

```
s1 = 'String'
s2 = 'String'
s3 = 'string'

if s1 == s2:
    print('s1 and s2 are equal.')
```

Using the C String Library

- C's operators, in contrast, are essentially useless for working with strings.
- Strings are treated as arrays in C, so they're restricted in the same ways as arrays.
- In particular, they can't be copied or compared using operators.

Using the C String Library

- For example, direct attempts to copy or compare strings will fail.
- Copying a string into a character array using the = operator is not possible:
- *Initializing* a character array using = is legal
  - In this context, = is not the assignment operator.

```
char str1[10], str2[10];
…
str1 = "abc";  /*** WRONG ***/
str2 = str1;   /*** WRONG ***/

char str3[10] = "abc"; /*** CORRECT ***/
```

Using the C String Library

- Attempting to compare strings using a relational or equality operator is legal but won't produce the desired result
- This statement compares `str1` and `str2` as *pointers*.
- Since `str1` and `str2` have different addresses, the expression `str1 == str2` must have the value 0 (not equal).

```
if (str1 == str2) …   /*** WRONG ***/
```

Using the C String Library

- The C library provides a rich set of functions for performing operations on strings.
- Programs that need string operations should contain the following line to include the header file

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

Using the C String Library

- In subsequent examples, assume that `str1` and `str2` are character arrays used as strings.
- We will introduce the functions for
  - String Copy
  - String Length
  - String Concatenation (並列) (introduce in next week)
- There are more other useful functions for strings. Please check them online!

The `strcpy` (String Copy) Function

- `strcpy` copies the string `s2` into the string `s1`.
  - To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`."
- `strcpy` returns `s1` (a pointer to the destination string).

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

strcpy(str2, "abcd");  /* str2 now contains "abcd" */

strcpy(str1, str2);  /* str1 now contains "abcd" */
```

The `strcpy` (String Copy) Function

- `strcpy` copies the string `s2` into the string `s1`.
  - To be precise, we should say "`strcpy` copies the string pointed to by `s2` into the array pointed to by `s1`."
- `strcpy` returns `s1` (a pointer to the destination string).

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

strcpy(str2, "abcd");  /* str2 now contains "abcd" */

strcpy(str1, str2);  /* str1 now contains "abcd" */
```

The `strcpy` (String Copy) Function

- In the call `strcpy(str1, str2)`, `strcpy` has no way to check that the `str2` string will fit in the array pointed to by `str1`.
- If it doesn't, undefined behavior occurs.

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

strcpy(str2, "abcd");  /* str2 now contains "abcd" */

strcpy(str1, str2);  /* str1 now contains "abcd" */
```

The `strcpy` (String Copy) Function

- Hence, Calling the `strncpy` function is a safer, albeit slower, way to copy a string.
- `strncpy` has a third argument that limits the number of characters that will be copied.
  - `sizeof(str1)`: how many characters in array `str1`

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

strncpy(str1, str2, sizeof(str1));
```

The `strcpy` (String Copy) Function

- However, `strncpy` will leave `str1` without a terminating null character if the length of `str2` is greater than or equal to the size of the `str1` array.
- A safer way to use `strncpy`:
- The second statement guarantees that `str1` is always null-terminated.

```
//  char *strcpy(char *s1, const char *s2);
strncpy(str1, str2, sizeof(str1) - 1);
str1[sizeof(str1)-1] = '\0';
```

The `strlen` (String Length) Function

- `size_t` is a `typedef` name that represents one of C's unsigned integer types.
- `strlen` returns the length of a string `s`, not including the null character.

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

int len;

len = strlen("abc");  /* len is now 3 */
len = strlen("");     /* len is now 0 */
strcpy(str1, "abc");
len = strlen(str1);   /* len is now 3 */
```

Summary

- Reading and Writing Strings
  - Writing Strings Using `printf` and `puts`
  - Reading Strings Using `scanf` and `gets`
  - Reading Strings Character by Character
- Accessing the Characters in a String
- Using the C String Library
  - `strcpy`
  - `strlen`

Next lesson will finish the rest topics about string

- String Concatenation
- Arrays of Strings