Strings (1)

Program Design (II)

2022 Spring

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

Outline

- Introduction Strings
- How String Literals Are Stored
- String Variables
- Character Arrays versus Character Pointers

Introduction

- Introduce string *constants* (*literals* in the C standard) and string *variables*.

1 0.2 ‘a’

integer / float / character constant

“hello world”

string constant/literal

Introduction

- **Strings** are **arrays** of **characters** in which a special character—the **null character** (`\0`) —marks the end.
- The C library provides a collection of **functions** for working with **strings**.

h	e	l	l	o		w	o	r	l	d	**\0**

“hello world”

string constant/literal

Introduction

- A ***string constant/literal*** is a sequence of characters enclosed within double quotes
  - singal quotes is character `'a'`
  - double quoates is string “a”
- String literals may contain **escape sequences** (e.g., `\n`).
  - cursor to advance to the next line

“hello world”

“hello\nworld”

```
hello
world
```

Introduction

- We actuall saw lots of string literal in calls of `printf()` and `scanf()`

```
int a;
printf("hello world");
scanf("%d", &a);
```

Continuing a String Literal

- If a string literal is **too long** to fit on a single line,
- it’s hard to write and read in the file of code

```
printf("When you come to a fork in the road, take it. --Yogi Ber
```

Continuing a String Literal

- The **backslash character** (\) can be used to continue a string literal from one line to the next
- In general, the \ character can be used to **join** two or more lines of a program into a single line.

```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");
```

Continuing a String Literal

- **However**, if we use \ , the string must continue at the **beginning** of the next line
- damaging the programs’ **indented structure**

```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");
```

Indented (縮排) structure

Which one is the good Indented structure?

```
int i = 0;
while(1){
    if(i > 0){
        printf("Hello World");
    }
    i++;
}
```

```
int i = 0;
while(1){
    if(i > 0){
        printf("When you come to \
a fork in the road, take it. \
--Yogi Berra");
    }
    i++;
}
```

Indented (縮排) structure

Good Indented structure

```
int i = 0;
while(1){
    if(i > 0){
        printf("Hello World");
    }
    i++;
}
```

Bad Indented structure

```
int i = 0;
while(1){
    if(i > 0){
        printf("When you come to \
a fork in the road, take it. \
--Yogi Berra");
    }
    i++;
}
```

Continuing a String Literal

- To maintain **good indented** structure, there’s a **better** way to deal with long string literals.
- When multiple **string literals** are **adjacent** (相鄰), the compiler will **join** them into a single string.
- allows us to split a string literal over two or more lines:

```
printf("When you come to a fork in the road, take it.  \
--Yogi Berra");

printf("When you come to a fork in the road, take it.  "
       "--Yogi Berra");
```

How String Literals Are Stored

- When a C compiler encounters a **string literal** of length ***n*** in a program, it sets aside ***n* + 1 bytes** of memory for the string.
- This memory will contain the **characters** in the **string**, **plus one extra** character—the ***null character*** `\0`—to mark the **end of the string.**

“hello world”

1 2 3 4 5 6 7 8 9 10 11

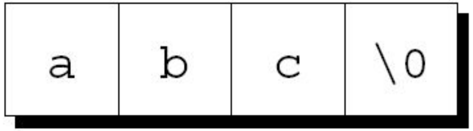
a string literal with 11 characters (bytes)

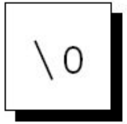
h	e	l	l	o		w	o	r	l	d	**\0**

need 12 bytes to store it
11 character + 1 extra character (null character)

How String Literals Are Stored

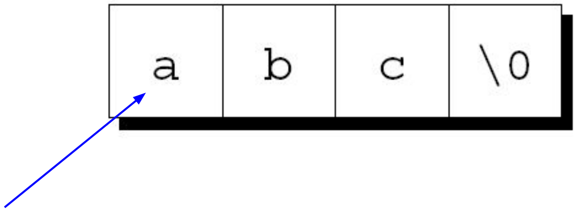
- The string literal `"abc"` is stored as an array of four characters:
- The **empty** string `""` is stored as a single null character:





How String Literals Are Stored

- Since a **string literal** is stored as an **array**, the compiler treats it as **a pointer of type** `char *`.
- Both `printf` and `scanf` expect **a value of type** `char *` as their first argument.
- The following call of `printf` passes the address of `"abc"`
    - a pointer to where the letter `a` is stored in memory (Review Ch 12. Pointer and Array)



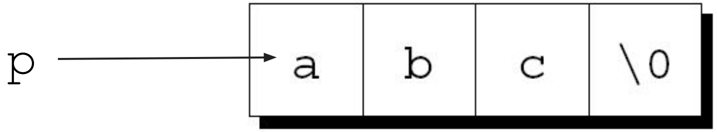
a	b	c	\0

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

type: `char *`

Operations on String Literals

- We can use a string literal wherever C allows a `char *` pointer:
- This assignment makes `p` **point** to the **first character** of the string.



```
char *p;

p = "abc";
```

Operations on String Literals

- However, attempting to **modify** a **string literal** causes **undefined** behavior
- The program below wants to change string into `"dbc"`
  - `*p:` the object that pointer `p` point to (indirection operater)

```
char *p;
p = “abc”;
*p = ‘d’;   /*** WRONG ***/
```

String Literals versus Character Constants

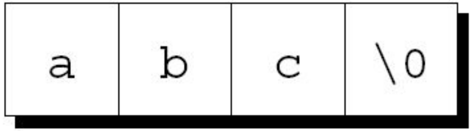
- A **string literal** containing a **single** character isn’t the same as a **character constant.**
  - `"a"` is represented by a ***pointer***.
  - `'a'` is represented by an ***integer***
    - C uses integer value (**ASCII** code) to represent character

```
printf("\n");
printf('\n');   /*** WRONG ***/
```

Let’s take a break!

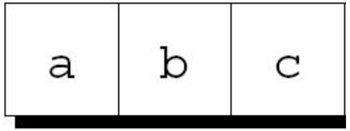
String Variables

- Some programming languages provide a special **string type** (e.g., c++)
- But C uses **one-dimensional array of characters** to store a string.
- And a string must be terminated by a null character `\0`.



a string literal

```
char str[4] = “abc”;
```



1D array of characters

```
char str2[3] = {‘a’, ‘b’, ‘c’};
```

String Variables

- Because of the **null** character `\0`, if a string variable needs to hold 80 characters, it must be **declared** to have **length 81**
- Adding 1 to the desired length allows room for the null character at the end of the string.

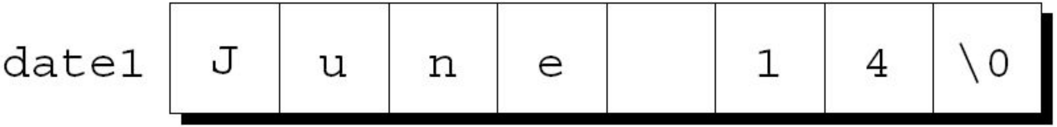
```
#define STR_LEN 80
…
char str[STR_LEN+1];
```

String Variables

- Be sure to leave room for the null character when declaring a string variable.
  - Failing to do so may cause **unpredictable** results when the program is executed.
  - because a string variable highly depend on the null character
- The actual **length** of a string depends on the **position** of the terminating **null character.**
  - without null character, you cannot determine the length of a string

Initializing a String Variable

- A string variable can be **initialized** at the same time it’s declared
- The compiler will **automatically** add a null character so that `date1` can be used as a string



date1	J	u	n	e		1	4	\0

```
char date1[8] = "June 14";
```

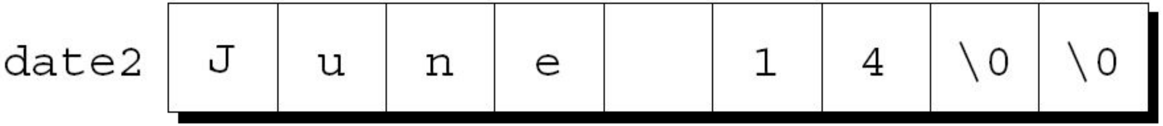
Initializing a String Variable

- `"June 14"` is not a string literal in this context
    - when being used to initialize a string variable
- Instead, C views it as an **abbreviation** for an array initializer.

```
char date1[8] = "June 14";
//char date1[8] = {'J', 'u', 'n', 'e', ' ', '1', '4', '\0'};
```

Initializing a String Variable

- If the initializer is too short to fill the string variable, the compiler adds extra null characters



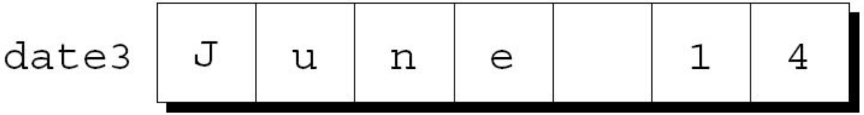
date2	J	u	n	e		1	4	\0	\0

```
char date2[9] = "June 14";
```

Initializing a String Variable

- An initializer for a string variable can’t be longer than the variable.
- It can be the same length, but…
- There’s no room for the null character, so the compiler makes no attempt to store one
- May cause **unpredictable** results, which should be avoided!

```
char date3[7] = "June 14";
```



slido

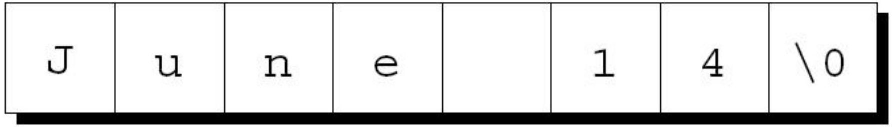
If you want to store "123456", which string variable you should declare?

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

Initializing a String Variable

- The declaration of a string variable may **omit** its **length**, in which case the compiler computes it. For example, …
- The compiler sets aside **8 characters** for `date4`, enough to store the characters in `"June 14"` **plus** a **null** character.
- useful if the initializer (string) is **long**

```
char date4[] = "June 14";
```



J	u	n	e		1	4	\0

Character Arrays versus Character Pointers

- Let’s compare the two declarations below.

```
char date[] = "June 14";
```

`date` is __?___

```
char *date = "June 14";
```

`date` is __?___

Character Arrays versus Character Pointers

- Thanks to the close relationship between arrays and pointers, either version can be used as a string.
- However, there are **significant differences** between the two versions of `date`.

```
char date[] = "June 14";
```

```
char *date = "June 14";
```

Character Arrays versus Character Pointers

- In the array version, the characters stored in `date` can be modified.
- In the pointer version, `date` points to a string literal that shouldn’t be modified.

```
char date[] = "June 14";
```

```
char *date = "June 14";
```

Character Arrays versus Character Pointers

- In the array version, `date` is an array name.
- In the pointer version, `date` is a **variable** that can **point to other strings**.
  - you can make `date` point to another string by `date = "abc";`

```
char date[] = "June 14";
```

```
char *date = "June 14";
```

Character Arrays versus Character Pointers

- The declaration `char *p;` does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.
- In other words, we need to **initialize** the pointer `p.` For example,...

```
char *p;
p = "June 14";
```

```
char *p;
char str[STR_LEN + 1];
p = str;
```

Character Arrays versus Character Pointers

- Using an uninitialized pointer variable as a string is a serious error.
- An attempt at building the string `"abc"`
- Since `p` hasn't been initialized, this causes **undefined** behavior.
  - before pointing to a string, `p` just a pointer variable

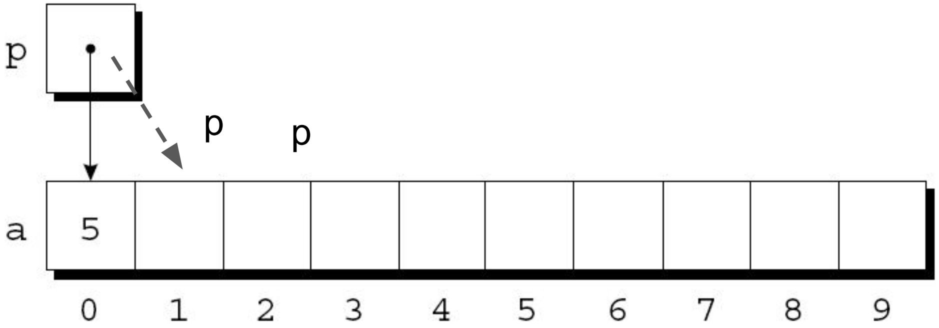
```
char *p;

p[0] = 'a';    /*** WRONG ***/
p[1] = 'b';    /*** WRONG ***/
p[2] = 'c';    /*** WRONG ***/
p[3] = '\0';   /*** WRONG ***/
```

Review: Pointer Arithmetic

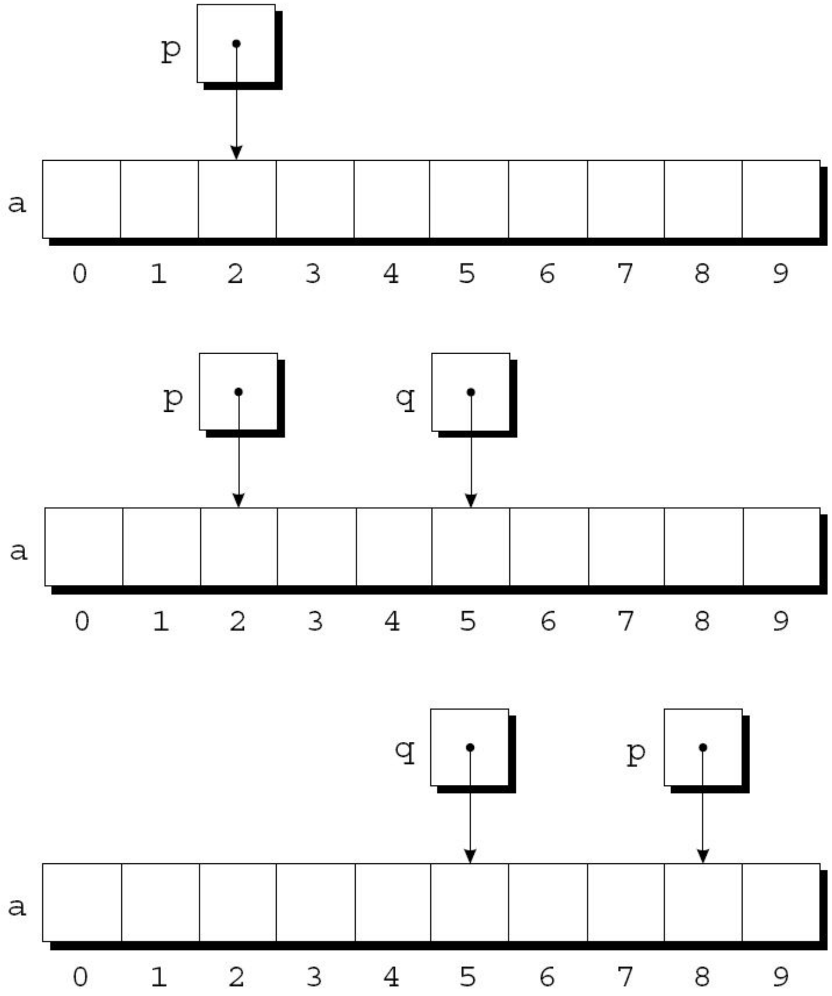
- If p points to an element of an array a, the **other elements** of a can be **accessed** by performing ***pointer arithmetic*** (or ***address arithmetic***) on p.

```
int a[10], *p;
p = &a[0];
*p = 5;
*(p + 1) = 6;
//equal to: p[1] = 6;
```



Review: Pointer Arithmetic

```
int a[10], *p, *q, i;
p = &a[2];
q = p + 3;
p += 6;
```



slido

If we want to print c by using pointer p, what is the correct printf() statement?

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

Summary

- Introduction Strings
- How String Literals Are Stored
- String Variables
- Character Arrays versus Character Pointers