

# Computer Programming: Skills & Concepts (CP)

## Strings

Ajitha Rajan

Tuesday 31 October 2017

CP-14 – slide 1 – Tuesday 31 October 2017

## Today's lecture

- ▶ Strings
- ▶ String I/O.
- ▶ String Comparison.

CP-14 – slide 3 – Tuesday 31 October 2017

## Last lecture

- ▶ Input handling
- ▶ char

CP-14 – slide 2 – Tuesday 31 October 2017

## Strings

A *string* is any 1-dimensional character array that is terminated by a null character.

- ▶ Null is `'\0'`.
- ▶ For local use, we declare strings by `char *s = "thestring"` or `char s[11]`
- ▶ `char *s = "thestring"` declares a pointer variable that points to the first character of the (constant) string;
- ▶ whereas `char s[11]` declares an array of 11 characters.
- ▶ Strings are declared in *function arguments* either as `char *s` or `char s[]`.  
eg, `void foo(char *s)` or `void foo(char s[])`  
(meaning ... a pointer to a char)
- ▶ In declaring a string, array length must be 1 greater than the longest string it will hold, to allow for the null.  
eg, `char [11]` can hold a 10-character string.

CP-14 – slide 4 – Tuesday 31 October 2017

## char \* and char []

char \*a makes space for a single pointer variable – it makes no space for the string.

char b[] makes space for the string (but makes no space for a pointer).

If you want a string to read into or modify, use char [].

```
char b[] = "I can be written into";
```

```
char c[256]; // a nice big string to use
```

If you want a *constant* string (e.g. for messages), you can use char \*.

```
char *a = "I can't be written into";
```

If you want a variable to refer to strings that already exist, use char \*.

```
char *a;
```

See end of lecture for gory details.

CP-14 – slide 5 – Tuesday 31 October 2017

## The string library – types

```
char *strcpy(char *p1, const char *p2);
```

*Returns the pointer p1*

```
char *strcat(char *p1, const char *p2)
```

*likewise*

```
size_t strlen(const char *p1)
```

*size\_t is a system-dependent type. On DICE PCs it is an unsigned long int, i.e. an 8-byte integer.*

**WARNING:** When using strcat or strcpy, it is **your** responsibility to make sure p1 has enough space. E.g:

```
char a[5];
```

```
strcpy(a, "This string is too long");
```

will segfault, or worse, overwrite some other data.

CP-14 – slide 7 – Tuesday 31 October 2017

## The string library

- ▶ Need to include it at the start:
  - ▶ #include <string.h>
- ▶ To copy a string s2 into s1:
  - ▶ strcpy(s1,s2);    strcpy(s1,"Hello\n");
- ▶ To add s2 onto the end of s1:
  - ▶ strcat(s1,s2)
- ▶ Returns the length of s1:
  - ▶ strlen(s1)
- ▶ Many others ...

CP-14 – slide 6 – Tuesday 31 October 2017

## String I/O

(don't need <string.h> for these)

- ▶ To printf a string: printf("%s", s1);
- ▶ To read in a string:
  - ▶ scanf("%s", s1);    /\* ?why no & on s1? \*/

Write/Read from a *string* (not I/O stream):

- ▶ To print a float a into a string s1:
  - ▶ sprintf(s1,"hello, num=%f", a);
  - ▶ sprintf returns an integer, being the number of chars written;
  - ▶ make sure s1 has space.
- ▶ Similarly, we can read ints/floats etc; from a string via sscanf:
  - ▶ int sscanf(s1, "%d Montgomery St", &door);
  - ▶ Value returned is the number of variables assigned to.

CP-14 – slide 8 – Tuesday 31 October 2017

## What about <, <=, == etc on strings?

```
int main(void) {
    char sone[] = "hiya";
    char stwo[] = "cp";
    char sthr[] = "coders";
    if (sone <= stwo) {
        printf("\\"hiya\\" is less than or equal to \\"cp\\".\n");
    } else {
        printf("\\"cp\\" is less than \\"hiya\\".\n");
    }
    if (stwo <= sthr) {
        printf("\\"cp\\" is less than or equal to \\"coders\\".\n");
    } else {
        printf("\\"coders\\" is less than \\"cp\\".\n");
    }
    return EXIT_SUCCESS;
}
```

CP-14 – slide 9 – Tuesday 31 October 2017

## strcmp

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

returns 0 if s1 and s2 are equal,  
a negative int if string s1 is *lexicographically* less than s2  
a positive int if string s1 is *lexicographically* greater than s2

```
...
if (strcmp(sone, stwo) <= 0) {
    printf("\\"%s\\" is less than or equal to \\"%s\\".\n", sone, stwo);
} else {
    printf("\\"%s\\" is greater than \\"%s\\".\n", sone, stwo);
}
```

CP-14 – slide 11 – Tuesday 31 October 2017

## <, <=, == don't work for strings

(sone <= stwo)

- ▶ sone and stwo are *pointers* to char variables (ie, are addresses in memory).
- ▶ comparison is true is and only if address in sone is less than stwo.

Output is *unpredictable*: compiler is free to allocate memory addresses for variables

- ... in order of declaration in the program, or maybe
- ... combination of declaration order and string length, or maybe
- ... in reverse order of declaration in program, or even
- ... in lexicographic order of initialization string (if given).

CP-14 – slide 10 – Tuesday 31 October 2017

## Comparing arrays of other types

A string is a char array. What about comparing arrays of ints or floats?

```
int memcmp (const void *a1, const void *a2, size_t size);
```

- ▶ memcmp compares the size bytes of memory beginning at a1 against the size bytes of memory beginning at a2.
- ▶ Value returned has the same sign as the difference between the *first differing pair of bytes*.
- ▶ For this reason, only useful for testing *equality*, not relative order.

What is this void \* type? void is a type that nothing can be! But void \* is used as a generic pointer type: a void \* can be cast to any other pointer type.

CP-14 – slide 12 – Tuesday 31 October 2017

## strncpy and friends

The requirement to ensure that `s1` has enough space in `strcpy(s1,s2)` etc. is tedious – have to check length of `s2`. Frequent cause of ‘buffer overflows’ and security exposures.

For safety, all professionally written C code uses:

```
char *strncpy(char *dest, const char *src, size_t n);
```

which copies at most `n` characters of `src`. Example:

```
/* 50 character strings (excl. null) */
#define LEN 50
char s[LEN+1]; /* add one for the null */

strncpy(s,maybe_long_string,LEN);
s[LEN] = '\0'; /* make sure there's a null at the end */
```

Similarly for `strncat`, `snprintf` and so on.

CP-14 – slide 13 – Tuesday 31 October 2017

## char \* and char []

```
char *a = "foo1";
```

a 

char *
--------

 → 

'f'	'o'	'o'	'1'	'\0'
-----	-----	-----	-----	------

  
*memory* *in the program code*

```
char b[] = "foo2";
```

b → 

'f'	'o'	'o'	'2'	'\0'
-----	-----	-----	-----	------

  
*coded* *allocated memory*

In fact, `char b[] = "foo2";` is effectively a convenient abbreviation for

```
char b[sizeof("foo2")];
strcpy(b,"foo2");
```

and `b` is an abbreviation for `&b[0]`, the address of the first of the allocated character cells.

CP-14 – slide 15 – Tuesday 31 October 2017

## char \* and char []

What's the difference between

```
char *a = "foo1";
char b[] = "foo2";
```

`a` is a variable, holding a pointer to the first character of "foo1".

You can assign to it: `a = "bar";`

`b` is a pointer to the first character of "foo2".

You can't assign to it. `b = "bar";` is a *compile-time* error.

Can you modify the contents of the string?

`strcpy(b, "bar");` is ok, because `b` is an array of characters.

`strcpy(a, "bar");` fails *at run-time*, because `a` is a pointer to (the first character of) the *literal* string "foo1", and (reasonably enough) you can't change a literal string!

(But `a = b; strcpy(a, "bar");` is fine.)

CP-14 – slide 14 – Tuesday 31 October 2017

## Assigned Reading (Kelley and Pohl)

For Strings: §6.10, §6.11, Appendix A.14

CP-14 – slide 16 – Tuesday 31 October 2017