Computer Programming: Skills & Concepts (CP1)
Strings

8th November 2010
Last lecture

Sorting with merge sort and bubble sort.
Today’s lecture

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

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

- Null is ‘\0’.
- Strings are declared in function arguments either as `char *s` or `char s[]`.
  eg, `void foo(char *s)` or `void foo(char s[])`
- In declaring a string, array length must be 1 greater than the longest string it will hold, to allow for the null.
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 ...
The string library – types

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

Actually returns the pointer p1 which at return time holds the value of *p2.

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

size_t strlen(const char *p1)
the return type will be unsigned int or similar.
The string library – types

char *strcpy(char *p1, const char *p2);
Actually returns the pointer p1 which at return time holds the value of *p2.

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

size_t strlen(const char *p1)
the return type will be unsigned int or similar.

**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.
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? */
▶ 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 Bellevue Road", &door);
  ▶ Value returned is the number of variables assigned to.
What about <, <=, == etc on strings?

```c
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;
}
```
<, <=, == don’t work for strings

\[(s\text{one} <= \text{s\text{two}})\]

- \text{s\text{one} and \text{s\text{two} are pointers to char variables (ie, are addresses in memory).}}
- comparison is true is and only if address in \text{s\text{one}} is less than \text{s\text{two}}.

Output is \textit{unpredictable}: compiler may 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).
Better (non)-example for $\leq$

```c
char sone[12], stwo[12];
printf("Input 1 please: ");
scanf("%s", sone);
printf("\nInput 2 please: ");
scanf("%s", stwo);
if (sone $\leq$ stwo)
    printf("%s is less than %s.\n", sone, stwo);
else
    printf("%s is less than %s.\n", stwo, sone);
```

No initialization bias on memory-allocation.
Can swap roles of input 1 and 2 to see result of comparison is non-lexicographic.
**strcmp**

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

returns 0 if \( s_1 \) and \( s_2 \) are equal,
a negative int if string \( s_1 \) is *lexicographically* less than \( s_2 \)
a positive int if string \( s_1 \) is *lexicographically* greater than \( s_2 \)

```c
... if (strcmp(sone, stwo) <= 0)
    printf("'hiya' is less than or equal to 'cp'.\n");
else
    printf("'cp' is greater than 'hiya'.\n");
```
Comparing arrays of other types

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

```c
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.
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:

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

which copies at most n characters of src. Example:

```c
const int LEN = 50; /* 50 character strings (excl. null) */
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.
Assigned Reading (Kelley and Pohl)

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