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 . . .

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?

```
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;
}
```
\(<, \leq, ==\) don’t work for strings

\(\text{sone} \leq \text{stwo}\)

\(\text{sone and stwo are pointers to char variables (ie, are addresses in memory).}\)

\(\text{comparison is true is and only if address in sone is less than stwo.}\)

Output is \textit{unpredictable:} compiler may allocate memory addresses for variables

\(\ldots\) in order of declaration in the program, or maybe

\(\ldots\) combination of declaration order and string length, or maybe

\(\ldots\) in reverse order of declaration in program, or even

\(\ldots\) in lexicographic order of initialization string (if given).

\(\text{strcmp}\)

\[\text{int strcmp(const char *s1, const char *s2);}\]

\(\text{returns}\ 0\ \text{if}\ s1\ \text{and}\ s2\ \text{are equal,}\)

\(\text{a negative int if string s1 is lower than s2}\)

\(\text{a positive int if string s1 is higher than s2}\)

\[\ldots\]

\(\text{if (strcmp(sone, stwo) <= 0)}\)

\(\text{printf("'hiya' is less than or equal to 'cp'.\n");}\)

\(\text{else}\)

\(\text{printf("'cp' is greater than 'hiya'.\n");}\)

\(\text{Comparing arrays of other types}\)

\(\text{A string is a char array. What about comparing arrays of ints or floats?}\)

\[\text{int memcmp (const void *a1, const void *a2, size_t size);}\]

\(\text{\(
\text{\textit{memcmp}\ compares the size bytes of memory beginning at a1 against the size bytes of memory beginning at a2.}\)

\(\text{\textit{Value returned has the same sign as the difference between the first differing pair of bytes.}\)

\(\text{\textit{For this reason, only useful for testing equality, not relative order.}}\)\)
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