5. Strings

- No built-in *String* type in C
- Strings are implemented as null-terminated ('\0') arrays of *char*
  - NB: null **pointer** (4 or 8 bytes) is not the null **character** (1 byte)

```
#include <stdio.h>
int main(void)
{
    char name1[] = {'F', 'r', 'e', 'd'};     // NOT a string!
    char name2[] = {'F', 'r', 'e', 'd', 0};  // IS a string!
    char name3[] = "Fred";                   // Same as name2
    char* p = name2;           // Like name2 but p can vary
    char* q = "Fred";          // Like p but *q is read-only!
    printf("name2: %s\nname3: %s\n*p: %s\n*q: %s\n", name2, name3, p, q);
    printf("sizes: %zu, %zu, %zu, %zu, %zu\n", sizeof name1,
            sizeof name2, sizeof name3, sizeof p, sizeof q);
}
```

NB: *sizeof* is NOT a string length!!

Output:

```
name2: Fred
name3: Fred
*p: Fred
*q: Fred
sizes: 4, 5, 5, 8, 8
```

strings (cont'd)

- String are just arrays, so are modifiable (unlike Java or Python strings)
  - Be *extremely careful* not to run off end of array
    - "buffer overrun"
- But:

```
char s1[] = "My string";
char* s2 = "Another string";
s1[0] = 'x';  // OK
s2[0] = 'x';  // Segmentation fault!
```

DEMO with valgrind

How long is a piece of string?

Answer (?):

- However long it is if you measure it.
- As long as you want. You are the master of your own destiny.
- Never quite long enough.
- Exactly as long as it needs to be.

Specifically:

```
size_t strlen(const char* s)
{
    size_t len = 0;
    while (s[len] != 0) {
        len++;
    }
    return len;
}
```

This function, and many other semi-useful ones, is available via **`#include <string.h>`**

Library functions <string.h>

Don't use the crossed-out ones!

```
char* strcat(char* dest, const char* src);
char* strchr(const char* s, int c);
int strcmp(const char* s1, const char* s2);
int strcoll(const char* s1, const char* s2);
char* strcpy(char* dest, const char* src);
size_t strcspn(const char* s, const char* reject);
char* strdup(const char* s);
char* strfry(char* string);
size_t strlen(const char* s);
char* strncat(char* dest, const char* src, size_t n);
int strncmp(const char* s1, const char* s2, size_t n);
char* strncpy(char* dest, const char* src, size_t n);
```

This one would be really useful (once we've done dynamic memory) but isn't C99/C11 ☹

Blue ones particularly useful.
Note how use of *const* prevents mix-up of *src* and *dest*

Library functions <string.h> (cont'd)

```
char* strpbrk(const char* s, const char* accept);
char* strrchr(const char* s, int c);
char* strsep(char** stringp, const char* delim);
size_t strspn(const char* s, const char* accept);
char* strstr(const char* haystack, const char* needle);
char* strtok(char* s, const char* delim);
size_t strxfrm(char* dest, const char* src, size_t n);
```

Why not use strcpy, strcat *etc?*

- Consider:

```
char buff[6];
strcpy(buff, "Hello world!");
```

Crashes the computer!
Or, worse, corrupts other variables on the stack.

- No length check so entire 12-bytes-plus-null string gets copied into 6-char array.
  - That'll work. Yeah, right.
- `strncpy(buff, "Hello world", 6)` *doesn't* overflow buffer
  - But nor does it terminate the string. AARGGHHHH!!
  - Still, it's MUCH safer.
- BUT: `strncpy(buff, s, strlen(s))` is worse than useless
  - Doesn't check buffer size AND doesn't terminate the string!

Note on strlen

- It's important to remember that library functions are just ordinary pre-compiled C functions, e.g., *strlen* might be:

```
size_t strlen(const char* s)
{
    const char *p = s;
    while (*p != '\0') {
        p += 1;
    }
    return p - s;
}
```

or

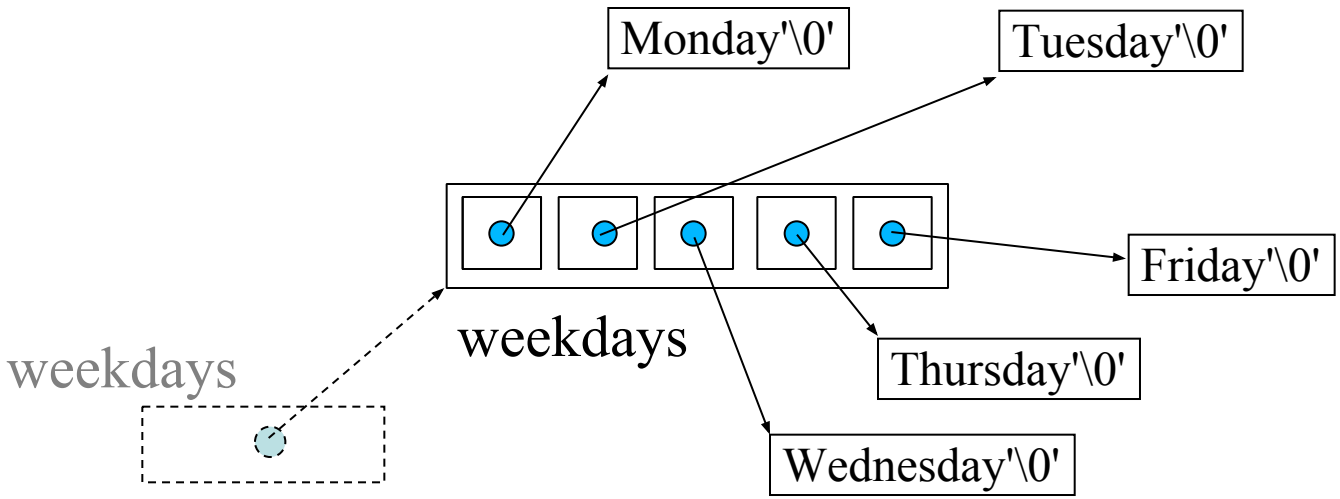
```
size_t strlen(const char* s)
{
    size_t len = 0;
    while (s[len] != '\0') {
        len += 1;
    }
    return len;
}
```

  - Note that these traverse the entire string, just to find its length!
- Valuable exercise:
  - Write your own versions of the following library functions:
    - *strchr*, *strncat*, *strncmp*, *strstr*

Arrays of strings

- Actually arrays of pointers to *char*

```
char* weekdays[] = {"Monday", "Tuesday", "Wednesday",
                                "Thursday", "Friday"};
```



Command-line arguments

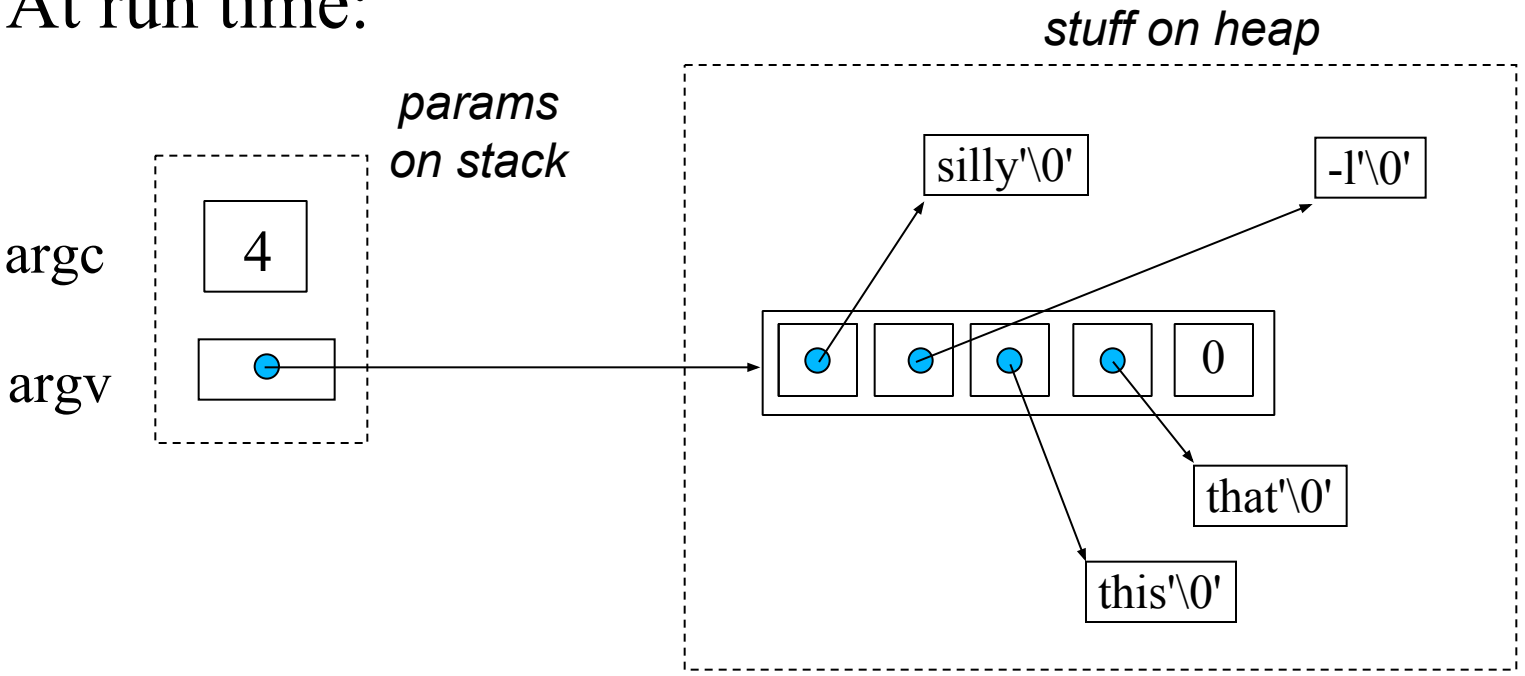
bash command: `silly -l this that`

where *silly* is:

```
int main(int argc, char* argv[]) {...}
```

Or we can use the type *char** argv*

At run time:



Example: program args.c

```
#include <stdio.h>
#include <stdlib.h>

// Print out all the command line arguments

int main(int argc, char* argv[])
{
    printf("The command line arguments are:\n");
    for (int i = 0; i < argc; i++) {
        printf("%d: %s\n", i, argv[i]);
    }
    return EXIT_SUCCESS;
}
```