C strings

(Reek, Ch. 9)

Review of strings

- Sequence of zero or more characters, terminated by `NUL` (literally, the integer value `0`)
- `NUL` terminates a string, but isn't part of it
  - important for `strlen()` – length doesn't include the `NUL`
- Strings are accessed through pointers/array names
- `string.h` contains prototypes of many useful functions

String literals

- Evaluating `"dog"` results in memory allocated for three characters `'d '`, `' o '`, `' g '`, plus terminating `NUL`

```
char *m = "dog";
```

- Note: If `m` is an array name, subtle difference:

```
char m[10] = "dog";
```

10 bytes are allocated for this array

This is not a string literal;
It's an array initializer in disguise!
Equivalent to
`{'d','o','g','\0'}`

String manipulation functions

- Read some "source" string(s), possibly write to some "destination" location

```
char *strcpy(char *dst, char const *src);
char *strcat (char *dst, char const *src);
```

- Programmer's responsibility to ensure that:
  - destination region large enough to hold result
  - source, destination regions don't overlap
    - "undefined" behavior in this case –

      according to C spec, anything could happen!

```
char m[10] = "dog";
strcpy(m+1, m);
```

Assuming that the implementation of `strcpy` starts copying left-to-right without checking for the presence of a terminating `NUL` first, what will happen?

strlen() and size_t

```
size_t strlen(char const *string);
/* returns length of string */
```

- `size_t` is an unsigned integer type, used to define sizes of strings and (other) memory blocks
  - Reasonable to think of "size" as unsigned"...
  - But beware! Expressions involving `strlen()` may be unsigned (perhaps unexpectedly)

```
if (strlen(x) - strlen(y) >= 0) ...
```

always true!

- avoid by casting:

```
((int) (strlen(x) - strlen(y)) >= 0)
```

  - Problem: what if `x` or `y` is a very large string?
- a better alternative: `(strlen(x) >= strlen(y))`

`strcmp()` “string comparison”

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

- returns a value less than zero if `s1` precedes `s2` in lexicographical order;
- returns zero if `s1` and `s2` are equal;
- returns a value greater than zero if `s1` follows `s2`.

- Source of a common mistake:
  - seems reasonable to assume that `strcmp` returns “true” (nonzero) if `s1` and `s2` are equal; “false” (zero) otherwise
  - In fact, *exactly the opposite* is the case!

Restricted vs. unrestricted string functions

- Restricted versions: require an extra integer argument that bounds the operation

```
char *strncpy(char *dst, char const *src, size_t len);
char *strncat(char *dst, char const *src, size_t len);
int strncmp(char const *s1, char const *s2, size_t len);
```

  - "safer" in that they avoid problems with missing `NUL` terminators
  - safety concern with `strncpy`:

    If bound isn't large enough, terminating `NUL` won't be written

    Safe alternative:

```
strncpy(buffer, name, BSIZE);
buffer[BSIZE-1] = '\0';
```

String searching

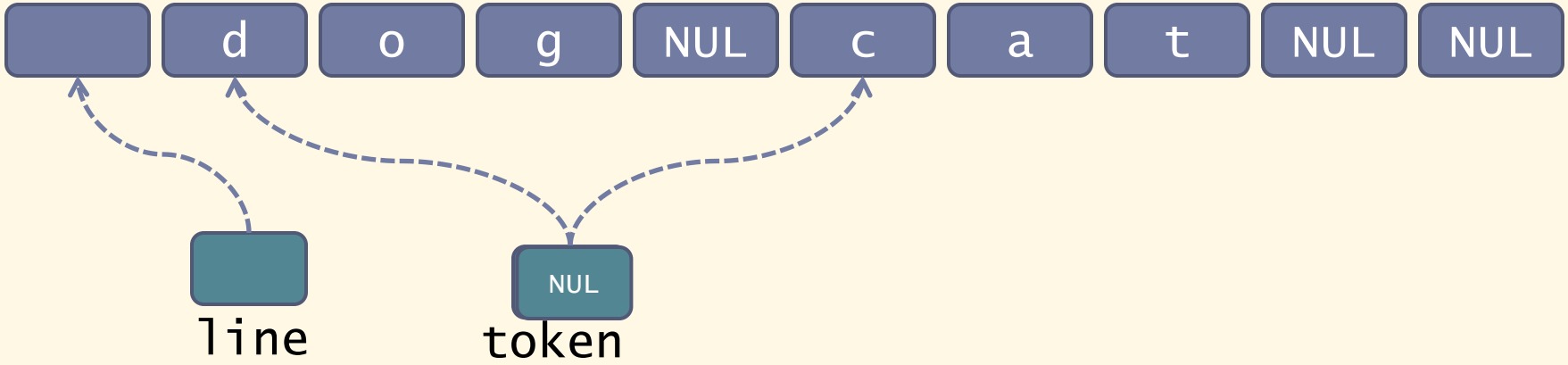
```
char *strpbrk(char const *str, char const *group);
/* return a pointer to the first character in str
   that matches *any* character in group;
   return NULL if there is no match */

size_t *strspn(char const *str, char const *group);
/* return number of characters at beginning of str
   that match *any* character in group */
```

strtok "string tokenizer"

```
char *strtok(char *s, char const *delim);
/* delim contains all possible "tokens":
   characters that separate "tokens".
   if delim non-NULL:
      return ptr to beginning of first token in s,
      and terminate token with NUL.
   if delim is NULL:
      use remainder of untokenized string from the
      last call to strtok */
```

strtok in action

```
for ( token = strtok(line, whitespace);
      token != NULL;
      token = strtok(NULL, whitespace))
  printf("Next token is %s\n", token);
```



An implementation of `strtok`

```
char* strtok(char *s, const char *delim) {
    static char *old = NULL;
    char *token;
    if (! s) { s = old; if (! s) return NULL; }
    if (s) {
      s += strspn(s, delim);
      if (*s == 0) { old = NULL; return NULL; }
    }
    token = s;
    s = strpbrk(s, delim);
    if (s == NULL) old = NULL;
    else { *s = 0; old = s + 1; }
    return token;
}
```

`old` contains the remains of an earlier `s` value (note use of `static`)

`NULL` has been passed in for `s`, so consult `old`

`strspn` returns number of delimiters at beginning of s – skip past these characters

strpbrk gives the position of the next delimiter. `s` is updated to this position, but `token` still points to the token to return.

Memory operations

- Like string operations, work on sequences of bytes
  - but do not terminate when `NUL` encountered

```
void *memcpy(void *dst, void const *src, size_t length);
void *memcmp(void const *a, void const *b, size_t length);
```

- Note: `memmove` works like `memcpy`, but allows overlapping source, destination regions
- Remember, these operations work on *bytes*
  - If you want to copy `N` items of type `T`, get the length right: `memcpy(to, from, N * sizeof(T))`