#preprocessor

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

--Brian W. Kernighan

David Notkin ● Autumn 2009 ● CSE303 Lecture 17

Type `char`

- `char` : A primitive type representing single characters
  - literal `char` values have apostrophes: `'a'` or `'4'` or `'\n'` or `'\''`
- you can compare char values with relational operators
  - `'a' < 'b'` and `'X' == 'X'` and `'Q' != 'q'`
- What does this example do?

```
for (char c = 'a'; c <= 'z'; c++) {
    printf("%c”,c);
}
```

`char` and `int`

- `char`s are stored as integers internally (ASCII encoding)

```
'A' 65        'B'  66        ' ' 32        '\0' 0
'a' 97        'b'  98        '*' 42        '\n' 10
```

```
char letter = 'S';
printf("%d", letter);    // 83
```

- mixing `char` and `int` causes automatic conversion to `int`
  - `'a' + 2` is `99,` `'A' + 'A'` is `130`
  - to convert an `int` into the equivalent `char`, type-cast it -- `(char) ('a' + 2)` is `'c'`

Strings

- in C, strings are just arrays of characters (or pointers to char)
- the following code works in C – why?

```
char greet[7] = {'H', 'i', ' ', 'y', 'o', 'u'};
printf(greet);              // output:  Hi you
```

- the following versions also work and are equivalent:

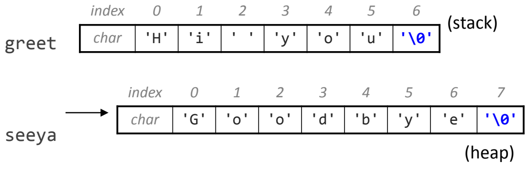
```
char greet[7] = "Hi you";
char greet[] = "Hi you";
```

- Why does the array have 7 elements?

Null-terminated strings

- in C, strings are **null-terminated** (end with a 0 byte, aka `'\0'`)
- string literals are put into the "code" memory segment
  - technically `"hello"` is a value of type `const char*`

```
char greet[7] = {'H', 'i', ' ', 'y', 'o', 'u'};
char* seeya = "Goodbye";
```



greet (stack)

index	0	1	2	3	4	5	6
char	'H'	'i'	' '	'y'	'o'	'u'	'\0'

seeya (heap)

index	0	1	2	3	4	5	6	7
char	'G'	'o'	'o'	'd'	'b'	'y'	'e'	'\0'

String input/output

```
char greet[7] = {'H', 'i', ' ', 'y', 'o', 'u'};
printf("Oh %8s!", greet);    // output: Oh   hi you!

char buffer[80] = {'\0'};    // input
scanf("%s", buffer);
```

- `scanf` reads one word at a time into an array (note the lack of `&`)
  - if user types more than 80 chars, will go past end of buffer (!)
- other console input functions:
  - `gets(char*)` reads an entire line of input into the given array
  - `getchar()` reads and returns one character of input

Looping over chars

- don't need `charAt` as in Java; just use `[]` to access characters

```
int i;
int s_count = 0;
char str[] = "Mississippi";
for (i = 0; i < 11; i++) {
    printf("%c\n", str[i]);
    if (str[i] == 's') {
        s_count++;
    }
}
printf("%d occurrences of letter s\n", s_count);
```

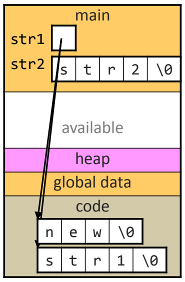
String literals

- when you create a string literal with "***text***", really it is just a `const char*` (unchangeable pointer) to a string in the code area

```
// pointer to const string literal
char* str1 = "str1";    // ok
str1[0] = 'X';          // not ok

// stack-allocated string buffer
char str2[] = "str2";   // ok
str2[0] = 'X';          // ok


// but pointer can be reassigned
str1 = "new";           // ok
str2 = "new";           // not ok
```



Pointer arithmetic

- +/- `n` from a pointer shifts the address by `n` times the size of the type being pointed to
  - Ex: Adding `1` to a `char*` shifts it ahead by `1` byte
  - Ex: Adding `1` to an `int*` shifts it ahead by `4` bytes

```
char[] s1 = "HAL";
char* s2 = s1 + 1;          // points to 'A'


int a1[3] = {10, 20, 30, 40, 50};
int* a2 = a1 + 2;           // points to 30
a2++;                       // points to 40
for (s2 = s1; *s2; s2++) {
    *s2++;                  // what does this do?
}                           // really weird!
```

How about this one?

```
char* s1 = "HAL";
char* s2;

for (s2 = s1; *s2; s2++) {
  printf("%c\n",(char)((*s2)+1));
};
```

Strings as user input

```
char buffer[80] = {0};
scanf("%s", buffer);
```

- reads one word (not line) from console, stores into buffer
- problem : might go over the end of the buffer
  - fix: specify a maximum length in format string placeholder
  - `scanf("%79s", buffer);     // why 79?`

- if you want a whole line, use `gets` instead
- if you want just one character, use `getchar` (reads `\n` explicitly)

String library functions

- `#include <string.h>`

function	description
`int strlen(`*`s`*`)`	returns length of string *s* until `\0`
`strcpy(`*`dst, src`*`)`	copies string characters from *src* into *dst*
`char* strdup(`*`s`*`)`	allocates and returns a copy of *s*
`strcat(`*`s1, s2`*`)`	concatenates *s2* onto the end of *s1 (puts \0)*

`int strcmp(`*`s1, s2`*`)`	returns < 0 if s1 comes before s2 in ABC order;
	returns > 0 if s1 comes after s2 in ABC order;
	returns 0 if s1 and s2 are the same

`int strchr(`*`s, c`*`)`	returns index of first occurrence of *c* in *s*
`int strstr(`*`s1, s2`*`)`	returns index of first occurrence of *s2* in *s1*
`char* strtok(`*`s, delim`*`)`	breaks apart ***s*** into tokens by delimiter ***delim***
`strncpy, strncat, strncmp`	length-limited versions of above functions

Comparing strings

- relational operators (`==, !=, <, >, <=, >=`) do not work on strings

```
char* str1 = "hello";
char* str2 = "hello";
if (str1 == str2) {           // no
```

- instead, use `strcmp` library function (`0` result means equal)

```
char* str1 = "hello";
char* str2 = "hello";
if (!strcmp(str1, str2)) {
    // then the strings are equal
    ...
}
```

More library functions

function	description
`int atoi(s)`	converts string (ASCII) to integer
`double atof(s)`	converts string to floating-point
`sprintf(s, format, params)`	writes formatted text into *s*
`sscanf(s, format, params)`	reads formatted tokens from *s*

- `#include <ctype.h>` (functions for `char`s)

function	description
`int isalnum(c), isalpha, isblank, isdigit, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper`	tests info about a single character

  - `isalpha('A')` returns a nonzero result (`true`)

Copying a string

- copying a string into a stack buffer:

```
char* str1 = "Please copy me";
char str2[80];  // must be >= strlen(str1) + 1
strcpy(str2, str1);
```

- copying a string into a heap buffer:

```
char* str1 = "Please copy me";
char* str2 = strdup(str1);
```

- do it yourself (hideous, yet beautiful):

```
char* str1 = "Please copy me";
char str2[80];
while (*s2++ = *s1++);  // why does this work?
```

Midterm A

- Suppose you have a shell script named **abc** and you execute

  $ ./abc > /dev/null

  Since standard output is redirected to /dev/null there is no output sent to the console. Does this always, never, or sometimes have the same effect as simply not executing the script? Briefly explain.

Midterm B

Consider the following commands and output in the shell:

$ grep grep grep
grep: grep: No such file or directory
$ grep
Usage: grep [OPTION]... PATTERN [FILE]...
Try `grep --help' for more information.

If you instead enter

$ grep grep

what happens? Be precise.

Midterm C

•Consider the following command

```
grep -E "(/\*([^*]|(\*+[^*/]))*\*+/)|(//.*)" *.c
```

•It is intended to search C programs for lines that include comments. The part of the regular expression before the underlined part matches *****, the part immediately after matches one or more ***** followed by a **/**, and the last part matches comments starting with **//**. Concisely explain what the underlined part of the regular expression matches.

Midterm D

1) Write a shell script **double** that accepts a single argument. The script must execute the command named by the argument and pass this command the original argument. For example, if you execute

$ ./double man

it will execute the **man** command with **man** as an argument…

2) What will this do?

$ ./double echo

3) What will this do?

$ ./double ./double

Midterm E

```
#include <stdio.h>
int main (int argc,char *argv[]) {
  int init,i,j,k;
  int data[10][10][10];
  init = atoi(argv[1]);
  init = scanf("%d",&init);
  for (i=0;i<=10;i++) {
    for (k=0;k<=10;k++) {
      for (j=0;j<=10;j++) {
        data[i][j][k] = init*i*j*k;
        printf("%d %d %d %d %d\n",
                  init,i,j,k,data[i][j][k]);
      };
    };
  };
}
```

Midterm F

- A Unix process can have more virtual memory than there is physical memory on the machine it runs on.
- We think of [the output from digits.c] as data. Is it imaginable to consider this as a program in a programming language called (for example) CSE303-weird?

Questions?