Strings
Announcements

• TA office hours for A1:
  Tues: 3-4:30 - BA 4261
  Wed: 3-4:30 - BA 5256
  Thurs: 2-3:30 - BA 4261
Today

- Strings
- String manipulation
- String functions
- Exercises
Strings

• Strings are not a built-in data type.
• C provides almost no special means of defining or working with strings.
• A string is an array of characters terminated with a “null character” (\0)
String Variables

- arrays are used to store strings
- strings are terminated by the null character (\0) (That's how we know a string's length.)
- Initializing strings:
  - char course[8] = "csc209h";
  - char course[8] = {'c','s','c','2','0','9','h','\0'};
  - course is an array of characters
String Variables

- **Initializing strings as character arrays:**
  - `char course[8] = "csc209h"; /* stored where? Stack! */`
  - `char course[8] = {'c','s','c','2','0','9','h','\0'};`
  - `course` is an array of characters

- **Using pointers:**
  - `char* course2 = course; /* stack or heap? Stack! */`
  - `char* course3 = malloc(8*sizeof(char));
    memcpy(course3, "CSC209h\0", 8); /* Heap! */`

- **String literals:**
  - `char *s = "csc209h";`
  - `s` is a pointer to a string literal
  - Completely different altogether!
String literals

```c
char *name = "csc209h";
```
String literals

```c
char *name = "csc209h";
printf("This is a string literal: %s\n", name);
```

- String literals are stored as character arrays, but you can't change them.

```c
name[1] = 'c'; /* Error */
```

- The compiler reserves space for the number of characters in the string plus one to store the null character.
String literals

- Avoid them unless no changes are expected!
- They are not const by default, but it’s good practice to mark it as such:
  - `const char *course = “csc209h”;`
  - Warns you if you pass to functions which may modify it:
    ```c
    void function1 (char* string) {
        string[0] = ‘A’;
    }
    ```
  - Fine for functions that take const strings:
    ```c
    void function2 (const char* string) {
        printf(“%s\n”, string);
    }
    ```
Character Arrays vs Character Pointers

• Using an uninitialized pointer variable as a string is a serious error.
• An attempt at building the string "abc":

```c
char *p, q[10];

p[0] = 'a';     //** WRONG **/
p[1] = 'b';     //** WRONG **/
p[2] = 'c';     //** WRONG **/
p[3] = '\0';   //** WRONG **/
```

• Since `p` hasn’t been initialized, this causes undefined behavior.
• Remember arrays are statically allocated

```c
q[0] = 'a';     //** CORRECT **/
```
Warning!

• Big difference between a string's length and size!
  – **length** is the number of non-null characters currently present in the string
  – **size** is the amount of memory allocated for storing the string
• e.g., `char s[10] = "abc";`
  – length of `s = 3`, size of `s = 10`
  – ensure length + 1 ≤ size!
String functions

• The string library provides a bunch of string functions which you should use (most of the time).

$\text{man string}$

• \text{int strlen(char \ast str)}
  – returns the length of the string. Remember that the storage needed for a string is one plus its length
Strlen and Sizeof operator

- Strlen always gives you the string length, excluding the ‘\0’

- Be careful when using sizeof:

```c
p = q;
printf(“%lu %lu”, sizeof(p), sizeof(q));
printf(“%lu %lu”, strlen(p), strlen(q));
/* What do you expect this to print? Try it! */
```

```c
p = malloc(10*sizeof(char));
printf(“%lu %lu”, sizeof(p), sizeof(q));
/* What about now? Try it on your system! */
```
Copying a string

char *strncpy(char *dest, char *src, int size)
  – copy up to size bytes of the string pointed to by src in to dest. Returns a pointer to dest.
  – Do not use strcpy (buffer overflow problem!)

char str1[3] = "a";
char str2[5] = "abcd";
/*common error*/
strncpy(str1, str2, strlen(str2)); /*wrong*/

What would be the correct way to do this?
Duplicating a string

• `strcpy` / `strncpy` assumes the destination is already malloc-ed

• To duplicate: malloc the destination + `strncpy`!

• `strdup` allocates memory automatically

  ```c
  char * str2 = strdup(str1);
  ```

  Equivalent to malloc-ing `str2` and a `strncpy`. 
Concatenating strings

char *strncat(char *s1, const char *s2, size_t n);

- appends the contents of string s2 to the end of s1, and returns s1.
- only appends up to n bytes to s1

• Watch out! It is easy to forget how much space is left.
  - char str1[6] = "abc";
  - strncat(str1, "def", 6); /*wrong*/
Comparing strings

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

• compares s1 and s2, returning a value less than, equal to, or greater than 0 depending on whether s1 is less than, equal to, or greater than s2
• if( strcmp(str1, str2) <= 0) 
  /* is str1 <= str2? */
• Alphabetically, according to ASCII codes
  – Capital letters are “before” small letters

Careful: if (str1 == str2) - compares the pointers (the addresses), NOT the string contents!!
Other useful functions

char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
int strncmp(const char *s1, const char *s2, size_t n)
char *strstr(const char *haystack, const char *needle)
char *strtok(char *str, const char *delim)
size_t strspn(char *s, const char *accept)
size_t strcspn(char *s, const char *reject)
char *strpbrk(const char *s, const char *accept)

Const == string will not be altered
$ man strchr

NAME
strchr, strrchr - locate character in string

SYNOPSIS
#include <string.h>

char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);

DESCRIPTION
The strchr() function returns a pointer to the first occurrence of the character c in the string s.

The strrchr() function returns a pointer to the last occurrence of the character c in the string s.

RETURN VALUE
The strchr() and strrchr() functions return a pointer to the matched character or NULL if the character is not found.
Strings example

• Problem: Given a name in the format “Last, First”, return a string in the format “First Last”

```c
char *reverse_name(char *src) {
    char *dest;
    char *sptr = strchr(src, ',');
    ...
    return dest;
}
```

• We'll first do an example with pointers.
Using string functions

char *reverse_name(char *src) {
    int src_len = strlen(src), dest_len = 0;
    char *dest;
    char *sptr = strchr(src, ',');

    /* allocate space for return string */
    if ((dest = malloc(src_len+1)) == NULL) {
        return NULL;
    }

    /* Move past the comma and the spaces between the comma and the first name */
    sptr++;
    while(*sptr == ' ')
        sptr++;
    sptr++;
}
/* Copy the first name to dest */
strncat(dest, sptr, strlen(src) + 1);

/* Add a space to the destination string */
dest_len = strlen(dest);
dest[dest_len] = ' '; 
dest[dest_len + 1] = '\0';

/* Copy the last name from src to dest */
strncat(dest, src, src_len - dest_len - 1);
dest[src_len-1] = '\0';
return dest;
int main()
{
    char *sptr, name[MAX];

    while((sptr = fgets(name, MAX, stdin)) != NULL) {
        /* strip the newline */
        sptr = strchr(name, '\n');
        *sptr = '\0';
        printf("%s\n", reverse_name(name));
    }
    return 0;
}
Recall commandline args?

- Command line arguments:

```c
int main(int argc, char **argv) {
    if (argc != 3) {
        printf(“Usage: %s <name> <mark>
”, argv[0]);
        exit(1);
    }
    char* name = argv[1];
    int mark = atoi(argv[2]);
    printf(“%s %d
”, name, mark);
    return 0;
}
```
Pointers to pointers (revisited)

```
$ ./PROG  JOE  97
```

```
char** argv
```

```
Pointer to pointer =>
Array of char* =>
Array of strings =>
Array of arrays of characters
```

```
*argv == argv[0]
*(argv+1) == argv[1]
*(argv+2) == argv[2]
```

```
argv[1][0]
argv[1][1]
argv[2][0]
argv[2][1]
argv[2][2]
```

```
argv[0][0]
argv[0][1]
argv[0][2]
argv[2][0]
argv[2][1]
argv[2][2]
```

```
argv == An array of words!
```
int *p = malloc(4*sizeof(int));  
int **pp = &p;

Draw the memory map of the above, like in the previous slide!
# Pointer practice

```c
int *p = malloc(4*sizeof(int));
int *q = malloc(3*sizeof(int));
q[0]=9; q[1]=1; q[2]=13;

int **pp = malloc(2*sizeof(int*));
pp[0] = p; pp[1] = q;
```

Similarly, draw the memory map!

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01b0</td>
<td></td>
</tr>
<tr>
<td>0x01b4</td>
<td></td>
</tr>
<tr>
<td>0xdca0</td>
<td>0</td>
</tr>
<tr>
<td>0xdca4</td>
<td>15</td>
</tr>
<tr>
<td>0xdca8</td>
<td>23</td>
</tr>
<tr>
<td>0xdcac</td>
<td>-1</td>
</tr>
<tr>
<td>0xaae4</td>
<td>9</td>
</tr>
<tr>
<td>0xaae8</td>
<td>1</td>
</tr>
<tr>
<td>0xaaec</td>
<td>13</td>
</tr>
</tbody>
</table>

```

Each dimension of `pp` can be different! (Recall argv, each word can have a distinct size).