Intro to C

- Strings
- C String functions
- Arrayness of char arrays
- File I/O
- I/O functions

What is a "string"?

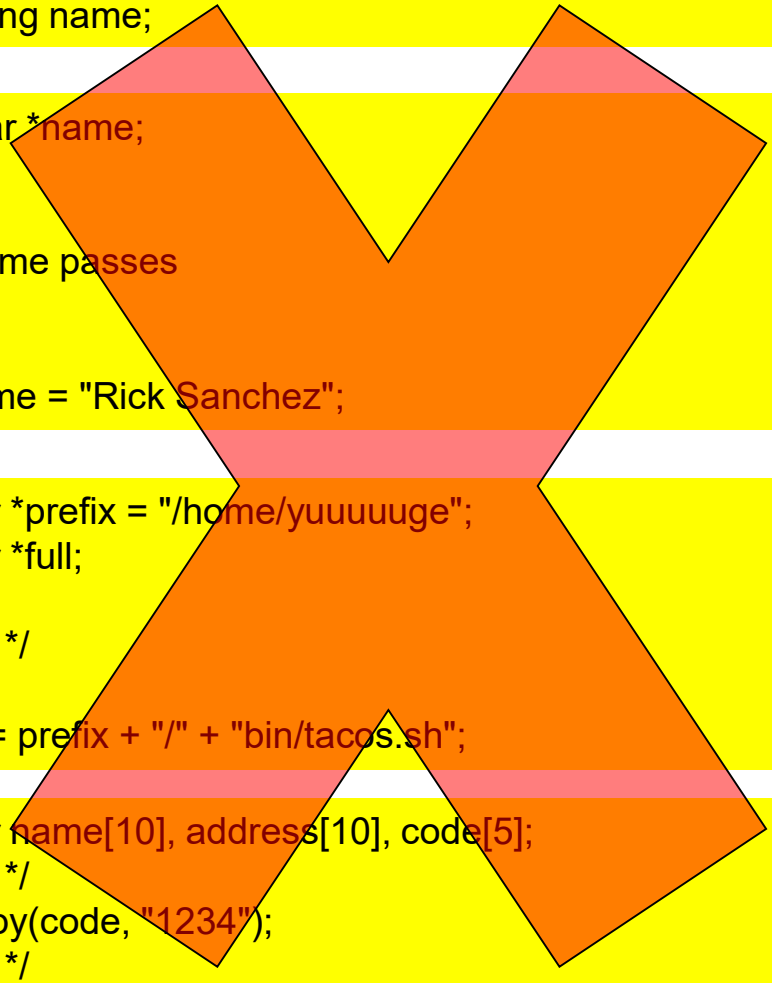
- "Strings" as a datatype known in Java **do not exist** in C
- Memory for strings is **not automatically allocated** on assignment.
- Concatenation via the "+" operator **is not possible**.
- The boundaries between strings **are not enforced** by the C runtime.

```
String name;
```

```
char *name;

/*
 * time passes
 */

name = "Rick Sanchez";
```

```
char *prefix = "/home/yuuuuuge";
char *full;

/* ... */

full = prefix + "/" + "bin/tacos.sh";
```

```
char name[10], address[10], code[5];
/* ... */
strcpy(code, "1234");
/* ... */
strcpy(address, "abcdefghijklmnopq");
/* ... */
printf("%s\n", code);
```



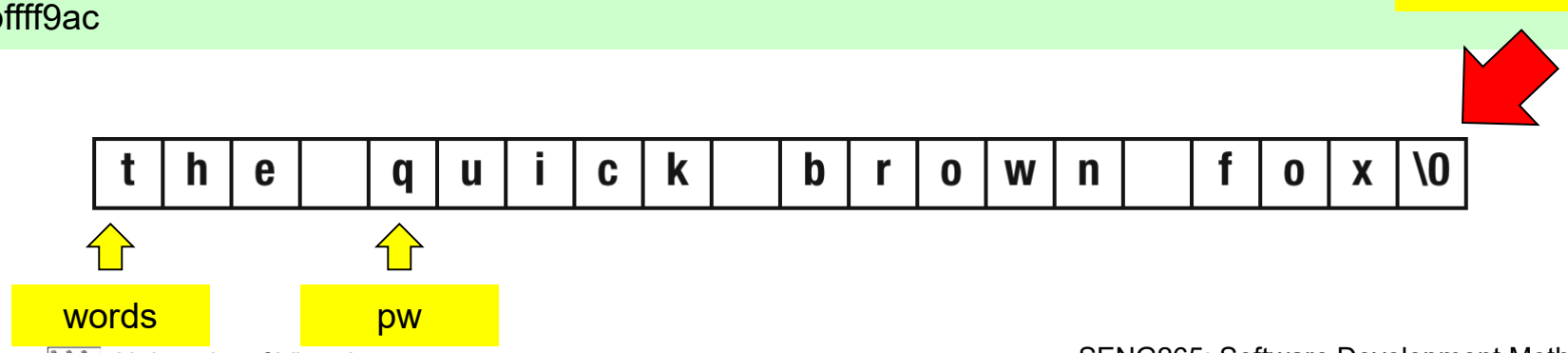
Strings are character arrays

- A C string is stored in a **character array**
- The **start** of a string is an **address to a char**
  - **The start of the string need not be identical with the start of an array!**
- The end of a string is indicated with a **null character** ('\0')
- The size of a string **need not necessarily be the same size** as the character array in which it is stored.
- C strings are often manipulated using special functions
  - **strncpy()**
  - **strcmp()**
  - **strncat()**
  - **strtok()**
- C strings are sometimes accessed **char** by **char**
- C strings are difficult to use at first
  - But you always have access to their underlying representation
- Mourn, and move on.

Example

```
char words[20];
char *pw;

/* ... */
strncpy(words, "the quick brown fox", 20);
pw = &words[0];  /* That's the same as writing "pw = words;". */
pw += 4;

printf ("%s\n%s\n", words, pw);
printf ("%x\n%x\n", words, pw);
```

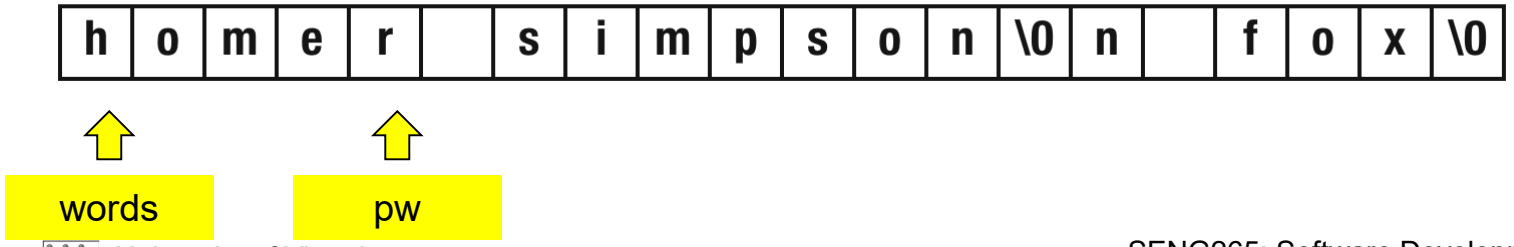
```
the quick brown fox
quick brown fox
bffff9a8
bffff9ac
```



Example

```
/* ... continued from previous slide ... */

strncpy(words, "homer simpson", 20);

printf ("%s\n%s\n", words, pw);
printf ("%x\n%x\n", words, pw);
```

```
homer simpson
r simpson
bffff9a8
bffff9ac
```



Always be aware of array-ness!

- Always be aware that C strings are, underneath, really just C char arrays
- To store a string in your program:
  - **You must have enough room in some character array** for all the string's characters **plus** one extra character for the null
  - Therefore correct program behavior often boils down to declaring (and later in the course, allocating) char arrays which have correct sizes for your purposes
- Must be scrupulous about specifying "maximum" sizes
  - Note the third parameter of "strncpy"
- Also use "strncat" to append a string to an already existing string

Example

```
char words[20];
char first[10];
char second[10];

strncpy(first, "aaaaa", 10);
strncpy(second, "bbbbb", 10);

strncpy(words, first, 20);
strncat(words, " ", 2);
strncat(words, second, 10);

printf("%s\n", words);
```

```
aaaaa bbbbb
```

Example with serious problems

```
#include <stdio.h>

int main() {
    char s[50] = "abcdefghijklmnopqrstuvwxyz";
    char *t = "zyxwvutsrqponmlkjihgfedcba";

    printf("message s is: '%s'\n", s);
    s[0] = ' ';
    s[1] = ' ';
    printf("modified message s is: '%s'\n", s);

    printf("message t is: '%s'\n", t);  /* next two lines will fail */
    t[0] = ' ';
    t[1] = ' ';
    printf("modified message s is: '%s'\n", t);
}
```

```
$ ./staticstring
message s is: 'abcdefghijklmnopqrstuvwxyz'
modified message s is: '  cdefghijklmnopqrstuvwxyz'
message t is: 'zyxwvutsrqponmlkjihgfedcba'
Bus error: 10
```

Strings

- In C, we can manipulate pointers in many ways
- This can help us when working with strings

char *cp = buffer same as **cp = &buffer[0]**

cp + n same as **&buffer[n]**

***(cp + n)** same as **buffer[n]**

cp++ same as **cp = cp + 1**

***cp++** same as ***cp, cp++**

C string functions

string.h: C string functions

- **strncpy(char *dest, const char *src, int length)**:
  - copies the contents of string **src** to the array pointed to by **dest**. **src** and **dest** should not overlap.
- **strncmp(const char *s1, const char *s2, int length):**
  - compares the two strings **s1** and **s2**, returning a negative, zero, or positive integer if **s1** is lexicographically <, ==, > **s2**.
- **strlen(const char *s):**
  - compute the length of string **s** (not counting the terminal null character (**'\0'**)).

Do not use strcpy()!

- **strcpy()** takes only two parameters:
  - destination char array
  - source char array
- If the string in the source array is longer than the size of the destination array:
  - then **strcpy()** will write over the end of destination array…
  - … and this is what happens in a buffer overflow attack
- What kind of bad things can happen?
  - Overwrite data in the activation frame
  - Cause function to return to a different location
  - read: https://en.wikipedia.org/wiki/Buffer_overflow

Extracting words from an array

- Common problem to be solved:
  - An input line consists of individual words
  - Words are separated by "whitespace" (space character, tabs, etc.)
  - Want to get a list of the individual words
- This is called "tokenization"
  - From the word "token" used by compiler writers
  - Once streams of tokens are extracted from text, compiler operates on tokens and not the text
- We ourselves can used tokenize functionality available in the C runtime library.

tokenize.c: global elements

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

/*
 * Compile-time constants
 */
#define MAX_WORD_LEN 20
#define MAX_WORDS    100
#define MAX_LINE_LEN 100
#define MAX_LINES    10

/*
 * Global variables
 */
int num_words = 0;
int num_lines = 0;
char lines[MAX_LINES][MAX_LINE_LEN];
char words[MAX_WORDS][MAX_WORD_LEN];

void dump_words (void);
void tokenize_line (char *);
```

The program will store lines of text.

It will also store words.

Size of global arrays is determined by the run-time constants.

The constants are not stored with the array!

Function prototypes…

tokenize.c: easy stuff

```
void dump_words ()
{
        int i = 0;

        for (i=0; i<num_words; i++) {
                printf("%5d : %s\n", i, words[i]);
        }

        return;
}
```

tokenize.c: easy stuff

```
int main(int argc, char *argv[])
{
        int i;

        if (argc == 1) {
                exit(0);
        }

        for (i=0; i < argc-1; i++) {
                strncpy(lines[i], argv[i+1], MAX_LINE_LEN);
                tokenize_line (lines[i]);
        }

        dump_words();

        printf("first line: \"%s\"\n", lines[0]);

        exit(0);
}
```

tokenize.c: hard stuff

```
void tokenize_line (char *input_line)
{
        char *t;

        t = strtok (input_line, " ");
        while (t && num_words < MAX_WORDS) {
                strncpy (words[num_words], t, MAX_WORD_LEN);
                num_words++;
                t = strtok (NULL, " ");
        }

                /* Question: What would now be the output from
                 * this statement:
                 *
                 * printf("%s\n", input_line);
                 *
                 */
        return;
}
```

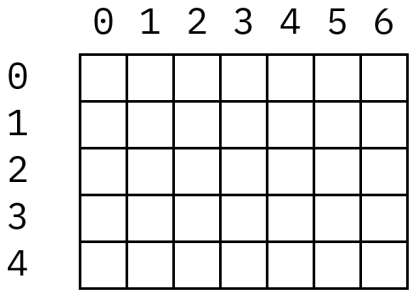
Note difference in the two calls to "strtok"

Second one uses "NULL" as the first paramteer.

Why do we use a "while" to structure the loop? Or could it be converted into a "for" loop?

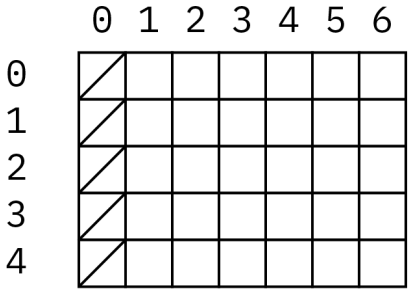
Array-ness of char arrays...

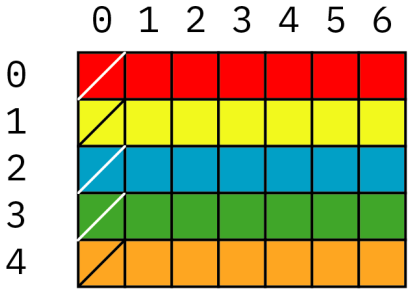
```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_WORDS 5
#define MAX_WORD_LEN 7

char w[MAX_WORDS][MAX_WORD_LEN];

...
```



Array-ness of char arrays...

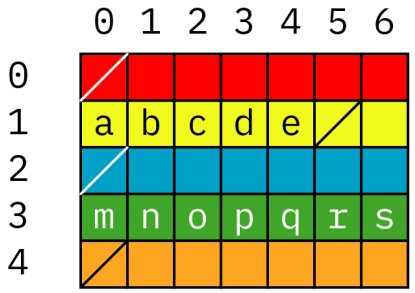
```
...

int main(int argc, char *argv[]) {
    int i;

    for (i = 0; i < MAX_WORDS; i++) {
        w[i][0] = '\0';  /* same as strncpy(w[i], "", 1);  */
                         /* ... but avoid w[i] = "";       */
    }

   ...
```





Array-ness of char arrays...

```
...

strncpy(w[1], "abcde", MAX_WORD_LEN);
strncpy(w[3], "mnopqrstuvwxyz", MAX_WORD_LEN);

...
```



Two-dimensional arrays must by laid down onto main memory (which is just a massive 1D array of bytes).

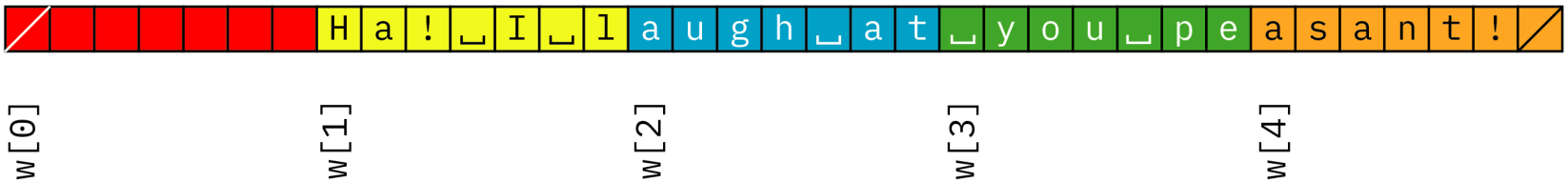
C uses “row-major” order

Array-ness of char arrays...

```
...

strcpy(w[1], "Ha! I laugh at you peasant!");

...
```

Before strcpy() ....

After strcpy() ....



/home/zastre/seng265/in-class/arrayness-of-words.c

File input and output

- C, like most languages, provides facilities for reading and writing files
- files are accessed as **streams** using **FILE** objects
- the **fopen()** function is used to open a file; it returns a pointer to info about the file being opened

```
FILE *data = fopen("input.txt", "r");
```

- streams **FILE *stdin**, **FILE *stdout**, and **FILE *stderr** are automatically opened by the O/S when a program starts
- But we need to unpack all this a little bit more...

File I/O

- open modes (text): "r" for reading, "w" for writing, and "a" for appending
- open modes (binary): "rb" for reading, "wb" for writing, and "ab" for appending
  - We'll look at **fopen()** in a moment...
- the **fclose()** function is used to close a file and flush any associated buffers
- use **fgetc()** to read a single character from an open file (file was opened in "r" mode)
- similarly, **fputc()** will output a single character to the open file (file was opened in "w" mode)

I/O functions

- **FILE *fopen(char *filename, char *mode)**
  - open file corresponding to filename
  - mode can be "r" or "rw" or "rw+" (depending on flavour of Unix)
  - if an error occurs when opening file, function returns 0 (NULL)
- **char *fgets(char *buf, int n, FILE *stream)**
  - read at most **n-1** characters from **stream** and copy to location **buf**; input terminates when newline encountered or n-1 characters input. Appends a null character to end of buffer.
  - returns **NULL** if error or end-of-file encountered
  - set **stream** to **stdin** to accept input from standard input
- **int scanf(char *format, […])**
  - read formatted data from standard input
  - returns **EOF** when end-of-file encountered, otherwise it returns the number of fields successfully converted
  - format specifiers encoded in **format** (variable # of arguments)

I/O functions

- **standard output (stdout)**
- **int printf(char *format, […])**
  - print formatted output to standard output
  - returns the number of characters printed
  - the format specifiers are encoded in the string **format**
  - takes a variable number of arguments
- Examples:
  - printf("My name is %s\n", name); /* char array */
  - printf("My name is %s and my age is %d\n", name, age);
    /* name is a char array, age is an int */
  - printf("The temperature today is %f\n", temp_celsius);
    /* temp_celsius is a float */
  - printf("%d/%d/%d", year, month, day);
    /* year, month and day are ints; there is no newline */

I/O functions

- **int fprintf(FILE *stream, char *format, […])**
  - like printf, but output goes to (already opened) stream
- **int fputc(int c, FILE *stream)**
  - outputs a single character (indicated by ASCII code in c) to (already opened) stream
  - note that the character is stored in an integer
  - idea here is the character is a number from 0 to 255
  - (if you pass a char as the first parameter, the function will still work)
- **int fclose(FILE *stream)**
  - closes the stream (i.e., flushes all OS buffers such that output to file is completed)
  - dissociates the actual file from the stream variable
  - returns 0 if file closed successfully.

File I/O

```
/* charbychar.c
 * Echo the contents of file specified as the first argument,
 * char by char. */

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

int main(int argc, char *argv[]) {
    int ch, num_char;

    if (argc < 2) {
        fprintf(stderr, "You must provide a filename\n");
        exit(1);
    }

    FILE *data_fp = fopen(argv[1], "r");

    if (data_fp == NULL) {
        fprintf(stderr, "unable to open %s\n", argv[1]);
        exit(1);
    }

    num_char = 0;
    while ((ch = fgetc(data_fp)) != EOF) {
        num_char++;
        printf("%c", ch);
    }
    fclose(data_fp);

    fprintf(stdout, "Number of characters: %d\n", num_char);
    return 0;
}
```

```c
/* linebyline.c
 * Echo the contents of file specified as the first argument, line by line. */

#include <stdio.h>
#include <stdlib.h>
#define BUFLEN 100

int main(int argc, char *argv[]) {
    char buffer[BUFLEN];
    int num_lines;

    if (argc < 2) {
        fprintf(stderr, "You must provide a filename\n");
        exit(1);
    }

    FILE *data_fp = fopen(argv[1], "r");

    if (data_fp == NULL) {
        fprintf(stderr, "unable to open %s\n", argv[1]);
        exit(1);
    }

    num_lines = 0;
    while (fgets(buffer, sizeof(char) * BUFLEN, data_fp)) {
        num_lines++;
        printf("%d: %s", num_lines, buffer);
    }
    fclose(data_fp);
    return 0;
}
```