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 = "Homer Simpson";
```

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

/* ... */

full = prefix + "/" + "bin/foo.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.

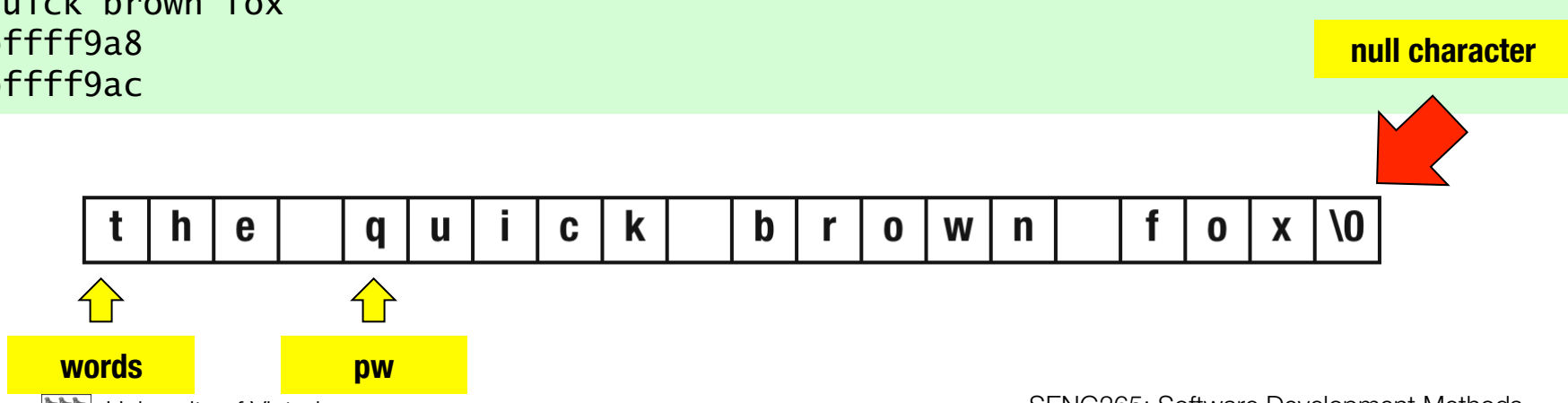
Example

```
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
```



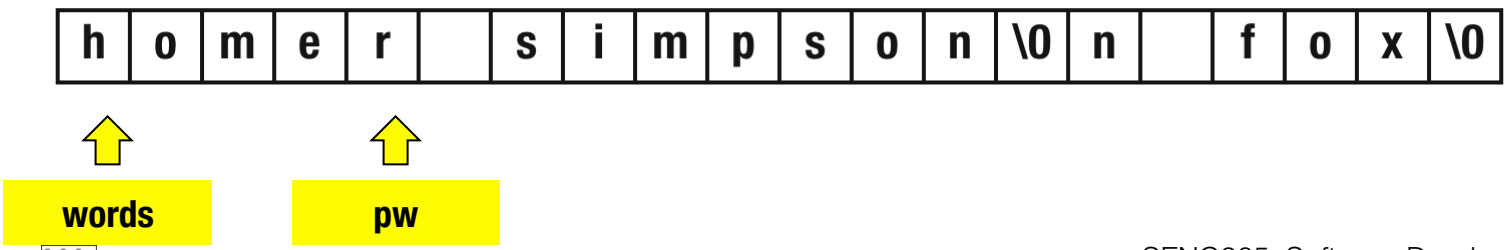
Example

```
/* ... 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, "first", 10);
strncpy(second, "second", 10);

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

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

```
first second
```

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 + 1)`

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'`)).

C string programming idioms

- "programming idiom"
  - "means of expressing a recurring construct in one or more programming languages"
  - use of idioms indicates language fluency
  - also assumes some comfort with the language
- idioms also imply terseness
  - expressions using idioms tend to be the "ideal" size
  - "terseness" can even have an impact as machine-code level
- non-string example: infinite loop

A C-language idiom: Infinite loop

```
/*
 * Not the ideal technique
 */

while (1) {
    some_function();
    if (someflag == 0) {
        break;
    }
    some_other_function();
}
```

Loop must always perform a check at the start of the loop.

```
/*
 * Recommended approach ("idiomatic").
 */

for (;;) {
    some_function();
    if (someflag == 0) {
        break;
    }
    some_other_function();
}
```

There is no check at the start of the loop -- no extra instructions!

Example: Computing string length

- Note!
  - Normally we use built-in library functions wherever possible.
  - There is a built-in string-length function ("strlen").
  - These libraries functions are very efficient and very fast (and bug free)
- Algorithm:
  - Function accepts pointer to a character array as a parameter
  - Some loop examines characters in the array
  - Loop terminates when the null character is encountered
  - Number of character examined becomes the string length

First example

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

#define MAX_STRING_LEN 100

int stringlength_1(char a_string[])
{
        int len = 0;

        while (a_string[len] != '\0') {
                len = len + 1;
        }
        return len;
}

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

        if (argc == 1) { exit(0); }
        strncpy(a_string, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_1(a_string));

         exit(0);
}
```

C knows nothing about the bounds of arrays!

"char a_string[]" is the same as "char *a_string"

Each character is explicity compared against the null character. Note the single quotes!

Name of character array is passed as the parameter to stringlength_1.

"a_string" is the same as "&a_string[0]"

First example: not idiomatic

- C strings are usually manipulated via indexed loops
  - "for" statements
- "For" statements are suitable to use with loops:
  - where termination depends the size of some array
  - where termination depends upon the size of some linear structure
  - where loop tests are at loop-top and loop-variable update operations occur at the loop-end
- "While" statements are most suitable with loops:
  - where termination depends on the change of some state
  - where termination depends on some property of a complex data structure
  - where actual loop operations can possibly lengthen or shorten number of loop iterations (e.g., "worklist" algorithms)

Second example

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

#define MAX_STRING_LEN 100

int stringlength_2(char a_string[])
{
        int len;

        for (len = 0; a_string[len] != '\0'; len = len + 1) { }

        return len;
}

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

        if (argc == 1) { exit(0); }
        strncpy(a_string, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_2(a_string));

         exit(0);
}
```

Each character is explicity compared against the null character, but this is done within the "for" header.

"For" loop itself is empty.

Second example: not idiomatic

- C strings are most often accessed via char pointers
- Accessing individual characters by array index is rare
  - Principle is that strings are usually processed in one direction or another
  - That direction proceeds char by char
- More idiomatic usage also depends upon pointer arithmetic

Third example

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

#define MAX_STRING_LEN 100

int stringlength_3(char a_string[])
{
        char *c;
        int len = 0;

        for (c = a_string; *c != '\0'; c = c + 1) {
                len = len + 1;
        }
        return len;
}

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

        if (argc == 1) { exit(0); }
        strncpy(a_string, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_3(a_string));

        exit(0);
}
```

Note that a character pointer is used (i.e., dereferenced in the control expression, and incremented in the post-loop expression).

The body of the "for" loop is not empty here as variable "len" is increment in it.

(Note: We could add "len" to our "for"-loop header and keep the body empty. What would that look like?

Fourth example

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

#define MAX_STRING_LEN 100

int stringlength_4(char a_string[])
{
        char *c;
        int len;

        for (len = 0, c = a_string; *c; c++, len++) { }

        return len;
}

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

        if (argc == 1) { exit(0); }
        strncpy(a_string, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_4(a_string));

        exit(0);
}
```

Note the "for"-loop termination condition!

We depend here on the meaning of "true" and "false" in C.

Note use of commas in the "for"-loop header.

Last examples: more idiomatic

- Char pointers were dereferenced
  - Value of dereference directly used to control loop.
- Char pointers were incremented
  - The most idiomatic code (not shown) combines dereferencing with incrementing
  - Example: `*c++`
  - Only works because "*" has a higher precedence than "++"
  - Meaning of example: "read the value stored in variable 'c', read the memory address corresponding to that value, return the value in that address as the expression value, and then increment the address stored in variable 'c'."

And tighter still…

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

#define MAX_STRING_LEN 100

int stringlength_5(char a_string[])
{
        char *c;

        for (c = a_string; *c; c++);

        return c - a_string;
}

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

        if (argc == 1) { exit(0); }
        strncpy(a_string, argv[1], MAX_STRING_LEN);
        printf("%d\n", stringlength_4(a_string));

         exit(0);
}
```

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?