C-STYLE STRINGS

Review

```
float foo[] = {1,2.32,3,4,5.0};
int throwAmt = 5;
int quux = 10;

void pityFool( int* foo )
{
    int throwAmt = 999;
    bool donePitying = false;
    while( !donePitying )
    {
        int pityAmt = 100;
        int* foo = &quux;
        *foo += throwAmt;
    }
}

int main()
{
    int* quux = new int[50];
    pityFool( quux );
}
```

Let's take a look-see at this code and figure it out.

(This is not terribly well-written code, by the way... make yours better than this!)

More Review

- How do you allocate a dynamic array?
- How do you clean it up?
- How do you clean up a single, dynamically allocated object?
- What chunk of memory do dynamically allocated objects come from?
- how ‘bout statically allocated objects?

Hey! Wouldn't it be nice if...

you could do stuff like this?

```
string word = "pickles";
word += " are tasty!";
cout << word << endl;
```

You ***sure can***! Just... not today.

Today we're learning about C-style strings,
which are quite a bit harder to use
and more annoying! Hooray!

C Strings

- It's important to know these - you'll come across them a lot, even when using C++
- A string in C is nothing special - just an array of `char`'s; each `char` holds a single character
- Messing with strings involves lots of nifty pointer arithmetic and manipulation

About Chars

- A character in C++ is a number (an 8-byte integer, to be exact)
- The numbers are coded using a standard mapping called ASCII: (American Standard Code for Information Interchange)
  - ‘a’ = 97, ‘b’ = 98, ‘A’ = 65, etc.
- You can find a table of these in about a gazillion places on the web

- You can assign single ASCII character values to a char using single quotes:

```
char letter = ‘A’;
cout << letter;
```

- Or you can assign a char an integer value (since it is an integer type):

```
char letter = 65;
cout << letter;
```

- You can also do arithmetic on characters:

```
char letter = ‘a’ + 2;
cout << letter;
```

Arrays of Chars

- Since a string is a sequence of characters, we can represent it as an array of characters:

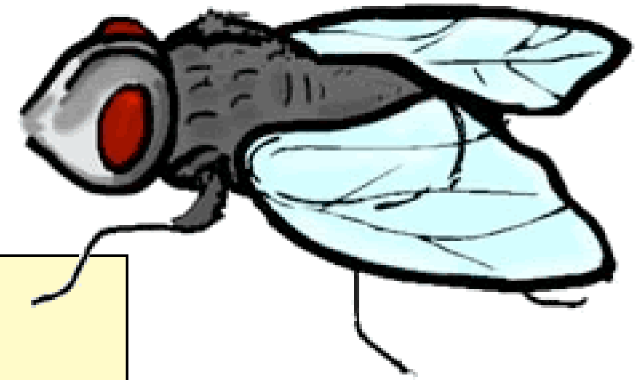
```
char turk[12];
```

- This array can hold up to 12 characters, as you'd expect
- This brings up the old array problem, though: how can you tell how big an array is?

TERMINATORS

- Other than storing a separate counter variable, there's no easy way to tell how many characters are in a string.
- The C solution to this is to have the last 'character' be a special character called a ***null terminator***, which has the value 0 - after this the string is considered "ended", even if there is more following.
- There needs to be space to store the null terminator too, so each character array needs to have at least one more slot than you have characters.

C Strings



```
char turk[12];
```

- turk has room for only 11 actual characters, and one null terminator:

c	h	r	i	s	t	o	p	h	e	r	\0

length: 11

- Even though 11 characters will fit, you don't *need* 11 characters. Less is fine:

c	h	r	i	s	\0						

length: 5

Declaring Strings (Character Arrays)

- Because a C-style string is just a character array, we can declare it like any other array:

```
char kelso[7];
```

- If you want to pre-initialize it with numbers, that's OK too: a `char` is an integer, after all!

```
char kelso[7] = {1,2,3,4,5,6,7};
```

- More useful, though, to be able to fill it with characters...

```
char kelso[7] = {'d','o','c','t','o','r','\0'};
```

Declaring Strings (Character Arrays)

- A shortcut in C/C++ is to use double-quotes in the initialization, instead of having to specify each character individually:

```
char kelso[7] = “doctor”;
```

- Note that we aren’t specifying the null terminator here: any string literal in C/C++ has the null terminator automatically appended.
- (A string literal means: any time you see stuff in double quotes in your source code file)

More Null Terminator Stuff

- The value of the null terminator is zero. Note that we specify it using a backslash-zero: '\0'
- You can embed this inside a string, too:

```
char CSBuilding[] = "MacLean\0 Hall";
cout << CSBuilding << endl;
```

- Even though there's more characters following "MacLean", once a function encounters the null terminator it will stop printing

A Quick Detour:
Fun with Escape Sequences!

- Notice that we had to use '\0', instead of just '0'? Why is that?
- The backslash (\) tells the compiler that this is the start of an **escape sequence**: it means that the character following the backslash has a special meaning
- So '\0' means "null terminator", whereas '0' just means 'zero'
  - Not the *integer* zero, mind you: it means the character zero, which is actually the integer 48!

A Quick Detour:
Fun with Escape Sequences!

Some common escape sequences:

\0	⟶	null terminator
\n	⟶	newline (like endl)
\’	⟶	single quote
\”	⟶	double quote
\\	⟶	an actual backslash

What does this mean?

It means that sometimes what you see isn’t what you get, and that you have to be careful with backslashes!

A Quick Detour:

Fun with Escape Sequences!

Here’s an actual chunk of (C) code that someone might write.
What’s wrong with this?

```
FILE* file = fopen(“C:\nichols\test.txt”,”r”)
```

We want these particular backslashes to be interpreted as *actual* backslashes, not escape sequences, so do it like this:

```
FILE* file = fopen(“C:\\nichols\\test.txt”,”r”)
```

On the other hand, escape sequences (newlines in particular) are often very handy, so feel free to use them:

```
cout << “I am very tired.\nI will go to sleep now.\n”;
```

Declarations:
Review / Check Yer Understanding

Which of these are valid and/or proper?

```
char bob[] = 1;
char bob[] = {1};
char bob[] = {'1', '\0'};
char bob[] = {'1', 0};
char bob[] = "hello";
char bob[] = {'h','e','l','l','o'};
char bob[30] = {'h','e','l','l','o','\0'};
char bob[3] = {'h','e','l','l','o','\0'};
```

Remember, we can also create strings dynamically:

```
char* bob = new char[50];
```

Note about Declarations

Stuff like this is nice and handy, but you **only** get to assign a string (or a group of numbers/ characters) to an array when you're declaring it.

```
char janitor[20] = "fearitude";
```

This doesn't work: (why not?)

```
char janitor[20];
janitor = "fearitude";
```

String Functions

- We've been using `<iostream>` for weeks now, but there are other libraries: a handy one for string functions is `<cstring>` or `<string.h>`
- Remember: this will include a header file, made up of function prototypes, but not the functions themselves: those get linked in later
- `<cstring>` gets you access to the old-school string functions in the C Standard Library
- ... it's important to know how these work, and what they're doing behind the scenes!

example: `strcpy`

This is a function that copies one string into another.

Here’s the prototype:

```
char *strcpy( char *dest, const char *src );
```

Here’s a sample usage:

```
char buffer[100];
strcpy( buffer, “Hi, I’m a string!” );
```

Anything bother you about this?

A Quick Detour:

Fun with Computer Security!

- When you put something into a string or array or *any* sort of data buffer, **C/C++ does not check to make sure that the data "fits"**.
- ***You*** are responsible for doing that.
- If you're not careful, `strcpy` and friends can be dangerous to use, because it will happily write past the end of the string, clobbering whatever happens to be in that memory.
- This isn't just bad programming; it can also be used to compromise your machine.

A Quick Detour:

Fun with Computer Security!

- So the moral of the story:
  - When you're coding your own functions, make **sure** that you include code to prevent any overwriting of the buffer. (How would you do this?)
  - Use "safe" C functions (strncpy, etc) when you can instead of the "dangerous" ones (like strcpy, wgets, etc)

Anyway... string functions.

Here's the prototype of `strlen`, a function that calculates the length of a string:

```
int strlen( const char *s );
```

- strlen works by counting each character in a string until it hits a null terminator (which is not included in the count). It's a pretty simple function.
- Let's try writing our own version of strlen!

Another Handy Library...

- ... is <cctype>, or <ctype.h>
- This is another group of functions from the C Standard Library that deal with classifying and modifying characters

some examples:

```
cout << isalpha('a') << endl;    // 1
cout << isalpha('8') << endl;    // 0
cout << isdigit('9') << endl;    // 1
cout << ispunct('#') << endl;    // 1
cout << isalnum('?') << endl;    // 0
cout << toupper('e') << endl;    // E
```

More Programming!

to tie a lot of this stuff together...

- Let's write a function kinda like strcpy, in that it copies a source buffer to a destination buffer, which we will create dynamically.
- It will include a maximum number of characters to copy (does this prevent overflow?)
- It will *only* copy characters that are either whitespace characters or alphanumeric.
- It will use lots of pointers! Hooray!

```
char* gcopy( char *dest, int maxCharsToCopy );
```