STRINGS

ADAM SWEENEY
CS 211

INTRODUCTION

- C++ has two ways of handling strings
- The older method, inherited from C, is why we've had to wait until now

AGENDA

- C-strings
- std::string

C-STRINGS

WHAT IS EVEN A C-STRING?

- C-strings are a special type of array
- Specifically, a null-terminated array of characters
  - `'\0'` is the null character
  - It is always the last character of a C-string
- A C-string is **always** an array of characters
- An array of characters is **not always** a C-string
- This distinction is extremely important
  - C-strings are treated differently than a plain array of characters
  - It is extremely easy to lose the null character

DECLARING A C-STRING

- Multiple methods
  - `char word[10] = “Hello”;`
  - `char word[] = {‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};`
  - `char word[] = “Hello”;`
  - `const char *word = “Hello”;`
- The last declaration is our first look at a pointer
  - `const char *` is also the type of all string literals
- The declarations are also initializations
  - This has to do with C-strings being a bit special

THIS IS NOT HOW TO INITIALIZE C-STRINGS

- `char word[10];`
  `word = “Hello”;`
- The above code will not compile; remember that C-strings are arrays
- Most common operations cannot be simply done with C-strings like they can with integers or doubles
- This is where `<cstring>` comes into play

SOME FUNCTIONS FROM `<CSTRING>`

Function	Description
`strlen(srcString)`	Returns integer equal to length of C-string. Null character is **not** counted
`strncat(dest, src, count)`	Joins `src` string to end of `dest`, up to count characters. **`dest` must be large enough to hold all the characters and the null character**
`strncpy(dest, src, count)`	Writes `src` into `dest`, up to count characters. **`dest` must be large enough to hold `src` and the null character**
`strncmp(lhs, rhs, count)`	Compares `lhs` against `rhs`, up to count characters. Returns a negative integer if `lhs` is "less" than `rhs` and a positive number if `lhs` is "greater" than `rhs`. Returns 0 if the two C-strings are equivalent. Comparison is lexicographic

UTILITY OF C-STRINGS

- Use `std::cin` to read into a C-string directly
  - Size is incredibly important
  - `std::cin.get()` or `std::cin.getline()` are safer; they allow you to specify a size
- Directly `std::cout` a C-string
  - No loop needed, automatically outputs only the relevant data
- Convert C-string to number and vice-versa
  - From `<cstdlib>`, functions like `atoi()` and `strtol()`

STD::STRING

`STD::STRING` IS A CLASS

- This course is not too concerned with Object-Oriented Programming
  - At the same time classes cannot be avoided
  - We have been using classes/objects since Hello World
- We do need to learn about them a bit in order to use them

CLASSES, IN BRIEF

- Classes collect data and functions around a common goal
- An instance of a class is an object
  - In other words, we declare objects of a class type
- Once we have an object, we can call class functions on it

```
std::string foo = "Hello";
unsigned int fooLength = foo.length();
```

- Class functions are called by using the **dot operator** on an object
- Class functions only work on the object they are called from

SMOOTH(ER) SAILING

- Many operations that required a <cstring> function work more naturally
- Concatenation occurs using addition
- Equivalence checks can be done with ==, and behaves the way we expect
- Simple assignment works as we would expect, as well
- No need to worry about the null character

BACKWARDS COMPATIBILITY

- We can still treat a std::string like an array of characters if we choose
  - This also incurs all the same risks
- We can even call the class member function c_str() to temporarily treat our std::string object like a C-string for use with older code