Computer Science I

Strings

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

1. Introduction
2. Manipulating Strings
3. String Processing
4. Data Processing
5. Exercises

Part I: Introduction

Strings

- A *string* is a sequence of *characters*
- ASCII, but more generally may be Unicode
- ASCII, CJK characters, emojis, etc.
- As of June 2017, Unicode 10.0: 136,755 characters
- Support for 1,112,064 characters
- We'll stick with ASCII

Strings

- Languages represent strings differently
- In C: a string is an array of `char` elements
- Huge caveat: all strings in C must end (terminate) with a null character, `\0`
- Failure to ensure that all strings are *null-terminated* will result in undefined behavior; seg faults, bus errors, etc.
- ASCII value of 0, but it is *NOT*:
  - `'\0'` ≠ `'0'`
  - `'\0'` ≠ `'\n'`
  - `'\0'` ≠ `NULL`
  - it is the *null terminating character*

Strings in C

- So far: single characters, `char`
- Character literals denoted with single quotes: `'A'`
- String literals denoted with double quotes: `"Hello World"`
- String variables:
  - Static strings: `char s[100];`
  - Static string and initialization: `char s[] = "Hello";`
  - Dynamic strings: `char *s;`
  - Constant Strings: `char *s = "Hello";` Don't do this

Static Strings

- Static string: `char s[100];`
- Creates a string that can hold *up to 99* characters
- Room is needed for the null terminating character
- It may hold *shorter* strings, but it may not exceed 99 characters
- At declaration, *contents are undefined*
- May or may not contain the null terminating character

Static Strings With Initialization

- Static string + initialization:
  `char s[] = "Hello";`
- Creates a character array of size 6
- Automatically includes the null-terminating character for you
- Contents can be changed, strings in C are *mutable*
- Static strings are allocated on the stack

Dynamic Strings

- Dynamic string:
  `char *s = (char *) malloc(sizeof(char) * 100);`
- Creates a character array of size 100
- May only hold a string that can hold *up to 99* characters
- Room is needed for the null terminating character
- *Contents are undefined*; may or may not contain the null terminating character
- Allocated on the heap
- Most of your strings will be of this type

Constant Strings

- Constant string declaration:
  `char *s = "Hello";`
- Dynamic, allocated on the heap, but
- Creates a *read-only, immutable* string
- Actually uses `const char *s = "Hello";`
- But the compiler generally doesn't catch it!
- Avoid unless you *really want* a dynamically allocated, immutable string for some reason

Manipulating Strings by Character

- Strings are simply character arrays
- Each individual element can be *indexed* and
- a value can be assigned to it.
- Demonstration

Manipulating Strings by Character

```
char message[] = "hello World!"; //size is 13

message[0] = 'H';
//bad:
message[0] = "H";
printf("message = %s\n", message);

//cut the string short:
message[5] = '\0';
printf("message = %s\n", message);

//restore it: the rest of the contents were unchanged
message[5] = ' ';
printf("message = %s\n", message);

message[11] = '?';
printf("message = %s\n", message);

//bad:
message[12] = '!';
printf("message = %s\n", message);

//really bad:
message = "Goodbye World!";
```

Part II: String Manipulation

Copying Strings

- You cannot *assign* a string after it has been declared

```c
char s1[] = "hello";
char *s2 = (char *) malloc(sizeof(char) * 6);
```

- Compiler error:
  `s1 = "Hello";`
- Memory leak:
  `s2 = "World";`
- A string is a character array which is a *memory address*

Copying Strings

- You must *copy* the contents of one string into another
- String library: `string.h` provides a copy function and many others
- `char * strcpy(char * dest, const char * src);`
- Copies the contents of the *source* string into the *destination* string
- Assumes `src` is properly null-terminated
- It is *your* responsibility to ensure that `dest` is large enough to hold the contents
- Demonstration

Copying Strings

```c
char *name = (char *) malloc(sizeof(char) * 10);

strcpy(name, "Chris");
strcpy(name, "Bourke");
//invalid:
strcpy(name, "Chris Bourke");
```

String Length

- Essential to know how many characters are stored in a string
- `size_t strlen(const char *s);`
- Result does *not* include the null terminating character!
- Demonstration

String Length

```c
char message[] = "Hello World!";
int n = strlen(message);
printf("n = %d\n", n);
message[5] = '\0';
n = strlen(message);
printf("n = %d\n", n);

char * stringCopy(const char *s) {
  char *copy = (char *) malloc(sizeof(char) * (strlen(s) + 1));
  strcpy(copy, s);
  return copy;
}
```

String Concatenation

- Copying overwrites a string's contents
- Alternative: append or *concatenation* the contents of one string onto the end of another
- `char *strcat(char *dest, const char *src);`
- Assumes both are null-terminated
- Assumes `dest` is large enough to hold both
- Demonstration

String Concatenation

```c
char *firstName = (char *) malloc((5+1) * sizeof(char));
char *lastName = (char *) malloc((6+1) * sizeof(char));
char *str = (char *) malloc(101) * sizeof(char));

strcpy(firstName, "Chris");
strcpy(lastName, "Bourke");

strcpy(str, lastName);
strcat(str, ", ");
strcat(str, firstName);
//str contains "Bourke, Chris"
```

Byte-Limited Versions

- String library provides *byte-limited* versions of both copy and concatenation functions
- `char *strncat(char *dest, const char *src, size_t n);`
- `char *strncpy(char *dest, const char *src, size_t n);`
- Only copies *at most* first `n` bytes/characters
- Stops early if it sees `\0`
- Includes `\0` *only if* it is within the first `n` bytes!
- Demonstration

Byte-Limited Versions

```c
char fullName[] = "Christopher";
char *nickName[] = (char *) malloc(6 * sizeof(char));

strncpy(nickName, fullName, 5);
//don't forget:
firstName[5] = '\0';
```

Part III: String Processing

Iterating Over Strings

- Common to iterate over a string character-by-character
- Straightforward solution: for-loop using `strlen()`
- Generally better ways
- Demonstration

Iterating Over Strings

```c
//straightforward code:
for(int i=0; i<strlen(s); i++) {
  printf("%c\n", s[i]);
}

//strlen works like:
int i = 0;
while(s[i] != '\0') {
  i++;
}
//the value of i equals strlen(s)

//optimized:
int n = strlen(s);
for(int i=0; i<n; i++) {
  printf("%c\n", s[i]);
}

//even better:
for(int i=0; s[i] != '\0'; i++) {
  printf("%c\n", s[i]);
}
```

ctype library

- May want to process or manipulate individual characters
- The `ctype.h` library provides many useful character functions
- Functions use ASCII `int` values
- Automatically type casted

ctype library

- `int isdigit(int c)` – returns true if `c` is a digit character, `0` thru `9`
- `int islower(int c)` – returns true if `c` is a lowercase letter character, `a` thru `z`
- `int isupper(int c)` – returns true if `c` is an uppercase letter character, `A` thru `Z`
- `int isspace(int c)` – returns true if `c` is a whitespace character: space, tab, newline, etc.
- `int tolower(int c)`, `int toupper(int c)` – return the ASCII text value of the lowercase/uppercase version of `c`
- Demonstration: write a code snippet to count the number of spaces and the total number of whitespace characters.

ctype library

```c
char str[] = "Hello how \n\t are you today? \n\n";
int numSpaces = 0;
int numWhiteSpaces = 0;

for(int i=0; s[i] != '\0'; i++) {
  if(isspace(s[i])) {
    numWhiteSpaces++;
  }
  if(s[i] == ' ') {
    numSpaces++;
  }
}
printf("number of spaces: %d\n", numSpaces);
printf("total whitespace: %d\n", numWhiteSpaces);
```

String Comparisons

- Often need to compare *entire strings* for equality
- You *cannot* use the equality operator!
- `s1 == s2` compares memory addresses!
- Need to use:
  `int strcmp(const char *str1, const char *str2)`
- Returns:
  - < 0 if `str1` comes before `str2`
  - 0 if contents are equal
  - > 0 if `str1` comes after `str2`
- Order is determined by ASCII text values
- Demonstration

String Comparisons

```c
int result;

result = strcmp("apple", "apple"); //0
result = strcmp("apple", "apples"); //negative
result = strcmp("apples", "apple"); //positive
result = strcmp("Apple", "apple"); //negative
result = strcmp("apples", "oranges"); //negative

result = strcmp("100", "99"); //negative!

result = strncmp("apple", "apples", 5); //zero

result = strcasecmp("ApPlE", "apple"); //zero
```

Substrings

- It is possible to reference a *substring* of a string
- Reference a part of the string starting at a particular index
- Demonstration

Substrings

```c
char name[] = "Margaret Hamilton";
char *lastName = &name[9];
printf("Greetings, Ms. %s\n");
```

Formatting Strings

- Already familiar with `%s` placeholder to print a string to the standard output
- `atoi` and `atof` convert strings to numbers
- Possible to convert numbers to strings
- "Print" to a string instead of the standard output
- `sprintf()` : print to a string
- Demonstration

Formatting Strings

```c
char s[100]; //buffer that is "big enough"
char state[] = "Nebraska";
int numCounties = 93;
double population = 1.92;
sprintf(s, "%s has %d counties and a population of %.2f million.\n", state,nu
```

Arrays of Strings

- Arrays of strings are simply 2-D arrays of `char` elements
- Each "row" is a string that must be null-terminated
- Each row/string need not be the same size
- Easy extension of 2-D arrays: `char **arrayOfStrings`
- Demonstration

Arrays of Strings

```c
char **names = (char **) malloc(sizeof(char*) * 5);
names[0] = stringCopy("Margaret Hamilton");
names[1] = stringCopy("Grace Hopper");
names[2] = stringCopy("Alan Turing");
names[3] = stringCopy("Ada Lovelace");
names[4] = stringCopy("Dennis Ritchie");
for(int i=0; i<5; i++) {
  printf("Famous Computer Scientist: %s\n", names[i]);
}
```

Part IV: Data Processing

String Tokenization

- Strings may contain formatted data: CSV, TSV
- *Tokenization* is the process of splitting a string along some *delimiter* and processing each *token* separately
- Example: `"Hedy,Lamarr,UNL,Avery Hall,Lincoln,NE"`
- Tokens: `"Hedy"` `"Lamarr"` `"UNL"` `"Avery"` `"Hall"` `"Lincoln"` `"NE"`
- Generally ignore the delimiter

String Tokenization

- `char * strtok(char *str, const char *delim);`
- Tokenizes `str` along instances of `delim`
- Usage:
  - First time you call it: pass the string to be tokenized
  - Subsequent calls: pass `NULL` to continue with the same string
  - Returns a pointer to the next token
  - It *modifies your string!*
- Demonstration

String Tokenization

```
char str[] = "Hedy,Lamarr,UNL,Avery Hall,Lincoln,NE";
char *token = NULL;
token = strtok(str, ",");
while(token != NULL) {
  printf("token = %s\n", token);
  token = strtok(NULL, ",");
}
```

Part V: Exercises

- Write a string function to change a string's characters to uppercase letters
- Write a string function that returns a new copy of a string with all characters converted to uppercase
- Write a string function to "double space" a paragraph
- Write a "split"-style function: it takes a string and a delimiter and returns an array of strings of the tokens. Ensure no memory leaks!