Computer Science & Engineering 155E
Computer Science I: Systems Engineering Focus

Lecture – Strings

Christopher M. Bourke
cbourke@cse.unl.edu

Strings

- Until now we have only dealt with single characters
- `char myChar = 'A', '\n'`
- Processing and manipulating single characters is too limiting
- Need a way for dealing with groups of characters

Strings

- A collection of characters is called a *string*
- C has no string data type
- Instead, strings are arrays of characters, `char myString[]`, `char myName[20]`
- Necessary to represent textual data, communicate with users in a readable manner

String Basics

- Calls to `scanf` or `printf` used a string constant as the first argument.
- We have also dealt with *static strings*: `"Hello World!"`
  ```
  printf("a = %d\n", a)
  printf("Average = %.2f", avg)
  ```
- Each string above is a string of 12, 7, and 14 characters respectively
- Its possible to use a preprocessor directive:
  ```
  #define INSUFF_DATA "Insufficient Data"
  ```

Static Strings

- Static strings cannot be changed during the execution of the program
- They cannot be manipulated or processed
- May only be changed by recompiling
- Stored in an array of a fixed size

Declaring and Initializing String Variables

- Strings are character arrays
- Declaration is the same, just use`char`
  ```
  char string_var[100];
  char myName[30];
  ```
- `myName` will hold strings anywhere from 0 to 29 characters long
- Individual characters can be accessed/set using indices

```
1 myName[0] = 'C';
2 myName[1] = 'h';
3 myName[2] = 'r';
4 myName[3] = 'i';
5 myName[4] = 's';
6 printf("First initial: %c.\n", myName[0]);
```

Declaring and Initializing String Variables

- You can declare and initialize in one line
- Be sure to use the double quotes
- `char myName[30] = "Chris";`
- You need not specify the size of the array when declaring-initializing in one line:
- `char myName[] = "Chris";`
- C will create a character array large enough to hold the string

Null Terminating Character

- C needs a way to tell where the *end* of a string is
- With arrays, it is the programmer's responsibility to ensure they do not access memory outside the array
- To determine where the string ends, C uses the *null-terminating character*, \0
- ASCII text character 0

Null Terminating Character

Example

`char str[20] = "Initial value";` will produce the following in memory:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
I	n	i	t	i	a	l		v	a
[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]	[19]
l	u	e	\0	?	?	?	?	?	?

Arrays of Strings

- Without the null terminating character, C would not know where the string ends
- Many functions parse a string until it sees \0
- Without it, the program would run into memory space that doesn't belong to it
- `char str[20]` can only hold **19** characters: at least one character is reserved for \0
- In declarations, `char myName[] = "Chris"`, C automatically inserts the null-terminating character

Printing Strings

- You can use `printf` to print strings
- Use `%s` as a placeholder:
  `printf("My Name is %s.\n", myName);`
- `printf` prints the string until the *first* null-terminating character

Arrays of Strings

- One string is an array of characters; an array of strings is a two-dimensional array of characters

```
1 #define NUM_PEOPLE 30
2 #define NAME_LEN 25
3 ...
4 char names[NUM_PEOPLE][NAME_LEN];
```

- `names` can hold 30 names each of up to *24* characters long

Arrays of Strings

We can initialize an array of strings at declaration in the following manner:

```
1 char month[12][10] = {"January", "February",
2      "March","April", "May", "June", "July",
3      "August", "September","October",
4      "November", "December"};
```

- As with other arrays, the [12] is optional
- Why [10]?
- September is the longest string with 9 characters
- Needs an additional character for the null-terminating character

Reading Strings I

- You can use scanf and %s to read strings
- ```
  printf("Enter Topic: ");
  scanf("%s", string_var);
  ```
  - scanf skips leading whitespace characters such as blanks, newlines, and tabs
  - Starting with the first non-whitespace character, scanf copies the characters it encounters into successive memory cells of its character array argument
  - When a whitespace character is reached, scanning stops, and scanf places the null character at the end of the string in its array argument

Reading Strings II

- Note: no & is used
- The array is already represented by a memory address
- Dangerous: the user can put as many characters as they want
- If they input more characters than the string can hold: overflow
- Segmentation Fault, may not even crash
- Rest of the program may produce garbage results

String Library Functions: Assignment and Substrings

- The assignment operator, = works for simple data types
- For strings, = *only* works in the declaration

```
1 char message[30];
2 message = "Hello!"; ← Illegal
```

- This is because arrays point to a *memory location*
- Cannot assign arbitrary values to memory pointers
- Must use library functions to do so

String Library

- C provides a standard *string library*
- Use #include<string.h>
- Table 9.1 on page 441 summarizes which functions are provided
- Copy, concatenation, comparison, length, tokenizer, etc.

String Assignment I

- To *assign* a value to a string, we actually *copy* it
- char *strcpy(char *dest, const char *src) copies string src (source) into dest (destination)
- Note:
  - Second argument has the keyword const: guarantees the source string is not modified
  - First argument *must* point to a memory location large enough to handle the size of dest
  - This is *your* responsibility, C does not do it for you
  - Returns a pointer to the first character of dest
```

String Assignment II

```
1 char myEmail[30];
2 strcpy(myEmail, "cbourke@cse.unl.edu");
```

- Be very careful:

```
1 char myEmail[10];
2 strcpy(myEmail, "cbourke@cse.unl.edu");
```

- In this case, `e.unl.edu` would overwrite adjacent memory cells

String Assignment I

Byte-wise

- C provides another copying function called `strncpy`:
  `char *strncpy(char *dest, const char *src, size_t n);`
- Copies (*up to*) `n` character values of `src` to `dest`
- Actually copies n bytes, but 1 `char` is one byte

```
1 char myEmail[] = "cbourke@cse.unl.edu";
2 char myLogin[30];
3 //copy first 7 characters:
4 strncpy(myLogin, myEmail, 7);
```

String Assignment II

Byte-wise

- Pitfall: If there is no null-terminating character in the first n bytes of `src`, `strncpy` will *not* insert one for you
- *You* must add the null terminating character yourself

```
1 char myEmail[] = "cbourke@cse.unl.edu";
2 char myLogin[30];
3 //copy first 7 characters:
4 strncpy(myLogin, myEmail, 7);
5 myLogin[7] = '\0';
```

String Assignment III

Byte-wise

- If n is larger than `src`, the null-terminating character is copied multiple times:
  `strncpy(aString, "Test", 8);`
- Four null terminating characters will be copied
- Thus, `aString` contains `"Test\0\0\0\0"`

Concatenation I

- *Concatenation* is the operation of appending two strings
- C provides concatenation functions:
  `char *strcat(char *dest, const char *src);`
  `char *strncat(char *dest, const char *src, size_t n);`
- Both append `src` onto the end of `dest`

Concatenation II

```
1 char fullName[80];
2 char firstName[30] = "Chris";
3 char lastName[30] = "Bourke";
4 strcpy(fullName,lastName);
5 strcat(fullName,", ");
6 strcat(fullName,firstName);
7 printf("My name is %s\n", fullName);
```

- Result: `My name is Bourke, Chris`

Concatenation III

- `strncat` copies at most n bytes
- From the documentation:
  If `src` contains n or more characters, `strncat()` writes $n+1$ characters to `dest` (n from `src` plus the terminating null byte). Therefore, the size of `dest` must be at least the length of `dest` $+n+1$

Comparisons I

- We can do character comparisons, `'A' < 'a'`
- We can also do string comparisons (lexicographic order)
- As before, we cannot use the usual operators `<, > <=`, etc.
- Strings (arrays of characters) are *memory addresses*
- `string_1 < string_2`, would compare the memory locations

Comparisons II

- String library provides several comparison functions:
  `int strcmp(const char *s1, const char *s2);`
  `int strncmp(const char *s1, const char *s2, size_t n);`
- Both compare `s1, s2`
  - If `s1 < s2`, returns a *negative* integer
  - If `s1 > s2`, returns a *positive* integer
  - If `s1 == s2` returns zero
- `strncmp` compares only the first n characters

Comparisons III

```
1 char nameA[] = "Alpha";
2 char nameB[] = "Beta";
3 char nameC[] = "Alphie";
4 if(strcmp(nameA,nameB) < 0)
5   printf("%s comes before %s\n", nameA, nameB);
6 if(strncmp(nameA,nameC,4) == 0)
7   printf("Almost the same!\n");
```

String Length

- The string library also provides a function to count the number of characters in a string:
  `size_t strlen(const char *s);`
- Returns the number of (bytes) characters appearing *before* the null terminating character
- Does *not* count the size of the array!

```
1 char message[50] = "You have mail";
2 int n = strlen(message);
3 printf("message has %d characters\n",n);
```

Result: `message has 13 characters`

Substrings I

- A *substring* is a portion of a string, not necessarily from the beginning
- `strncpy` can be used to extract a substring (of n characters), but only from the beginning
- However, we can use *referencing* to get the *memory address* of a character
- `&aString[3]` is the memory address of the 4th character in `aString`
- We can exploit this fact to copy an arbitrary substring

Substrings II

```
1 char aString[100] = "Please Email me at the address cbourke@cse.u
2 char myEmail[20];
3 //copy a substring
4 strncpy(myEmail, &aString[31], 19);
5 printf("email is %s\n",myEmail);
```

Result: `email = "cbourke@cse.unl.edu"`

Pitfalls & Strategies

Two most important questions when dealing with strings:

1. Is there enough room to perform the given operation?
2. Does the created string end in `'\0'`?

- Read the documentation
- Each string function has its own *expectations* and *guarantees*

Scanning a Full Line I

- The `scanf` only gets non-whitespace characters
- Sometimes it is necessary to get *everything*, including whitespace
- Standard function (in `stdio` library):
  `char *gets(char *s);`
  `char *fgets(char *s, int size, FILE *stream);`
- `gets` works with the standard input, `fgets` works with any buffer (more in Chapter 12)
- `gets` (**get** a **s**tring)

Scanning a Full Line II

```
1 char read_line[80];
2 gets(read_line);
3 printf("I read your line as \"%s\"\n", read_line);
```

- Dangerous: If the user enters more than 79 characters, no room for null-terminating character
- If user enters more than 80 characters: overflow
- Compiler message:

```
(.text+0x2c5): warning: the 'gets'function is dangerous and should not be used.
```

Scanning a Full Line III

- `fgets` is safer since you can limit the number of bytes it reads:
  `char read_line[80];`
  `fgets(read_line,80,stdin)`
- Reads at most `size-1` characters (automatically inserts null-terminating character)
- Takes the endline character out of the standard input, but retains it in the string

Comparison and Swapping

We can perform a sorting algorithm to a list of strings:

```
1 for(i=0; i<num_string; i++)
2 {
3    for(j=i; j<num_string; j++)
4    {
5       if(strcmp(list[i], list[j]) < 0)
6          Swap(list[i],list[j]);
7    }
8 }
```

What would `Swap` look like?

Comparison and Swapping

Swapping two strings:

```
1 strcpy(tmp, list[i]);
2 strcpy(list[i], list[j]);
3 strcpy(list[j], tmp);
```

Careful: how big does `tmp` need to be?

Tokenizing

- Data is often *delimited* by some marker
- CSV files: Comma Separated Value
- TSV: Tab Separated Value
- Useful to have a function to split strings into *tokens* according to some delimiter(s)

Tokenizing

- C tokenizer function:
  `char *strtok(char *str, const char *delim)`
- First call: pass `str`, string to be tokenized
- Each subsequent call: pass `NULL` as `str` (otherwise, it starts over)
- `delim`: a collection of delimiters, examples: `" "`, `",;:"`, `"\t"`
- Function returns a pointer to a null-terminated copy of the string (token) *without* the delimiter(s)

Tokenizing Example

```
1  #include<stdio.h>
2  #include<string.h>
3
4  int main(void)
5  {
6    char sent[] = "I am taking CSE 150A - Introduction to C";
7    char str[15][100];
8    char *tempStr = NULL;
9    int i=0;
10
11   tempStr = strtok(sent, " ");
12   while(!(tempStr == NULL))
13   {
14     strcpy(str[i], tempStr);
15     tempStr = strtok(NULL, " ");
16     i++;
17   }
18
19   for(i=0; i<9; i++)
20     printf("str[%d] = %s\n", i, str[i]);
21   return 0;
22 }
```

Tokenizing Example

Output

```
1 str[0] = I
2 str[1] = am
3 str[2] = taking
4 str[3] = CSE
5 str[4] = 150A
6 str[5] = -
7 str[6] = Introduction
8 str[7] = to
9 str[8] = C
```

Command Line Arguments I

Up to now, your `int main(void)` functions have not taken any parameters. To read parameters (delimited by white space) in from the command line, you can use

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

- `argc` gives you a **c**ount of the number of **arg**uments which are stored in `argv`
- `argv` is an array of strings (two dimensional array of characters)

Command Line Arguments II

- `argv`: the first element is the program name (ex: `argv[0] = a.out`)
- Subsequent elements of `argv` contain strings read from the command line
- Arguments are delimited by whitespace
- You can encapsulate multiple words from the command line using the double quotes

Command Line Arguments III

```
~>a.out hello world "hi yall"abc 123
```

would result in:

```
argc = 6
argv[0] = a.out
argv[1] = hello
argv[2] = world
argv[3] = hi yall
argv[4] = abc
argv[5] = 123
```

Command Line Arguments IV

```c
/*
  commandLineArgs.c

  Demonstrates the usage of command line arguments
  by printing the arguments back to the command
  line.

*/

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

int main(int argc, char *argv[])
{
  printf("You entered %d arguments.\n",argc-1);
  printf("Program Name: %s\n",argv[0]);
  int i;
  for(i=1; i<argc; i++)
    printf("\targv[%d] = %s\n",i,argv[i]);

  return 0;
}
```

Character Analysis and Conversion I

- The C `ctype.h` library provides several useful functions on *characters*
- `isalpha(char ch)` is true if `ch` is an alphabetic character (upper or lower case)
- `isdigit(char ch)` is true if `ch` is a character representing a digit

Character Analysis and Conversion II

- `islower(char ch)` is true if `ch` is a lower-case character
- `isupper(char ch)` (guess)
- `toupper` and `tolower` convert alphabetic characters (no effect otherwise)
- `ispunct(char ch)`
- `isspace(char ch)` true if `ch` is any whitespace character

String-to-Number and Number-to-String Conversions I

- C provides several functions for converting between strings and numbers
- String to numbers:
  `int atoi(const char *nptr);`
  `double atof(const char *nptr);`
- Returns the value of the number represented in the string `nptr`
- **a** (alpha-numeric) to **i**nteger, **f**loating point
- Does not handle errors well: returns zero if it fails (see `strtol` for advanced behavior)

String-to-Number and Number-to-String Conversions II

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

int main(int argc, char *argv[])
{
  if(argc != 3)
  {
    printf("Usage: %s integer double\n", argv[0]);
    exit(-1);
  }
  int a = atoi(argv[1]);
  double b = atof(argv[2]);
  printf("You gave a = %d, b = %f ",a,b);
  printf("as command line args\n");
  return 0;
}
```

String-to-Number and Number-to-String Conversions I

- `sprintf` takes numbers, doubles, characters, and strings and concatenates them into one large string.
  `sprintf(string_1, "%d integer %c - %s", int_val, char_val, string_2);`
  - If `int_val` = 42, `char_val` = 'a', and `string_2` = `"Tom Waits"`
  - then `string_1` would be `"42 integer a - Tom Waits"`
- `sscanf` takes a string and parses it into integer, doubles, characters, and strings

String-to-Number and Number-to-String Conversions II

```c
int num;
double pi;
char a[50], b[50];
sscanf("42 3.141592 Tom Waits", "%d %lf %s %s", &num,
                                                &pi,
                                                a,
                                                b);
printf("num = %d\n", num);
printf("pi  = %f\n", pi);
printf("a   = %s\n", a);
printf("b   = %s\n", b);
```

String-to-Number and Number-to-String Conversions III

Result:

```
num = 42
pi  = 3.141592
a   = Tom
b   = Waits
```

Common Programming Errors I

- We usually use functions to compute some value and use the return to send that value back to the main function. However functions are not allowed to return strings, so we must use what we learned about input/output parameters
- Know when to use & and when not to
  - Use them for simple data types`int`, `char`, and `double`
  - Do not use them for whole arrays (strings)

Common Programming Errors II

- Be careful not to overflow strings
- Always follow expected formats
- Read the documentation!
- Most important: make sure all strings are null-terminated (a `'\0'` at the end)
- Just because your program *seems* to work, doesn't mean it always does (ex: add & to `a, b` in the `sscanf` snippet above)

Exercises I

1. Write a program that takes command line arguments and prints them out one by one. Then sort them in lexicographic order and print them out again.
2. A *palindrome* is a string that is the same backwards and forwards (example: tenet, level). Write a program that reads a string from the command line and determines if it is a palindrome or not. In the case that it is not, make the string a palindrome by concatenating a reversed copy to the end.