CMPE-013/L

Introduction to “C” Programming
Gabriel Hugh Elkaim
Winter 2015

Text I/O
Text I/O

- Within `<stdio.h>`:
  - Formatted text: `scanf()`/`printf()`
  - Characters: `getchar()`/`putchar()`
  - Strings/lines: `fgets()`/`puts()`
  - *NEVER EVER EVER USE `gets()`*

**fgets()**

- `str` is where received data is stored
  - Needs to be an array
- `count` is how many characters to process
  - Stops when \n or `(count-1)` chars are received
- `stream` is `stdin`
Text I/O

fgets() example

Example

```c
#include <stdio.h>

int main(void)
{
    // Create enough memory for a 50 char string
    char inputData[50 + 1];

    fgets(inputData, sizeof(inputData), stdin);
}
```

String Processing
String Processing

• Within `<string.h>`
  – Examination
    • Length: `strlen()`
    • Comparing: `strcmp()`/`strncmp()`
    • Splitting: `strtok()`
  – Manipulation
    • Copying: `strncpy()` (Don’t use `strcpy()`!)
    • Appending: `strncat()`

---

```c
size_t strlen(const char *str);
```

• `str` is the string to calculate the length of
• `size_t` can be treated as an `int`

---

```c
int x = strlen("My string"); // x = 9
char str[] = "asdf";
int y = strlen(str); // y = 4
```
String Processing

\textbf{strcmp()}

Syntax

\begin{verbatim}
int strcmp(const char *s1, const char *s2);
\end{verbatim}

- Ignores size of the strings, purely alphabetical comparison
- Return value is > 0 if \texttt{s1} alphabetically before \texttt{s2}, 0 if they're equal, < 0 if \texttt{s2} alphabetically before \texttt{s1}

Examples

\begin{verbatim}
char *s1 = "apple", *s2 = "zed";
int cmpResult = strcmp(s1, s2);
if (cmpResult > 0) {
    printf("apple > zed\n");
} else if (cmpResult == 0) {
    printf("apple == zed\n");
} else {
    printf("apple < zed\n");
}
\end{verbatim}

\textbf{strtok()}

Syntax

\begin{verbatim}
char *strtok(char *s1, const char *s2);
\end{verbatim}

- \texttt{s1} (input/output), string to be tokenized
  - Will be modified!
- \texttt{s2} (input) – Delimiters

Examples

\begin{verbatim}
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " "); // firstToken = "This"
char *secondToken = strtok(NULL, " "); // secondToken = "is"
char *thirdToken = strtok(NULL, " "); // thirdToken = "an"
char *fourthToken = strtok(NULL, " "); // fourthToken = "example!"
\end{verbatim}
String Processing
strtok() Details

Example

```c
char s1[] = "This is an example!";

char *firstToken = strtok(s1, " ");
```

- `s1[]` is initialized with the string "This is an example!".
- `strtok()` is called with `s1` and a delimiter " ".
- The first token is saved in `firstToken`.

The string `s1` is then updated to "This is an example!\0".
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
```

```
firstToken
  s1

T h i s \0 i s \0 a n e x a m p l e ! \0
```

```
secondToken

T h i s \0 i s \0 a n e x a m p l e ! \0
```

String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
```

```
firstToken
  s1

T h i s \0 i s \0 a n \0 e x a m p l e ! \0
```

```
thirdToken

T h i s \0 i s \0 a n \0 e x a m p l e ! \0
```

```
secondToken

T h i s \0 i s \0 a n \0 e x a m p l e ! \0
```
String Processing

strtok() Details

Example

```c
char s1[] = "This is an example!";
char *firstToken = strtok(s1, " ");
char *secondToken = strtok(NULL, " ");
char *thirdToken = strtok(NULL, " ");
char *fourthToken = strtok(NULL, " ");

firstToken
s1

secondToken
thirdToken

T h i s \0 i s \0 a n \0 e x a m p l e ! \0

fourthToken
```

String Processing

strncpy()

Syntax

```c
char *strncpy(char *s1, const char *s2, size_t n);
```

- **s1** (output) – where the string will be copied to
- **s2** (input) - the string that to be copied
- **n** - how many characters can be copied
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50];
strncpy(s1, "asdf", 4); // s1 = "asdf\0"
strncpy(s1 + strlen(s1), "asdf", 4); // s1 = "asdfsdf\0"
```
String Processing

strncat()

Syntax

```c
char *strncat(char *s1, const char *s2, size_t n);
```

- **s1** (input/output) - is the base string
- **s2** (input) - the string that will be appended
- **n** - how many characters can be appended
- Undefined if s1 and s2 overlap!

Examples

```c
char s1[50] = "This is an example!";
strncat(s1, "asdf", 4);
```

String Processing

- Within `<stdlib.h>`:
  - Conversion
    - Integer: atoi(), xtoi()
    - Floats: atof()
  - Within `<stdio.h>`:
    - Conversion
      - Any: sscanf()
String Processing

atof()

Syntax

```c
double atof(const char *s);
```

- `s` (input) – The string to parse
- Returns the converted value or 0.0

Examples

```c
char s1[] = "1.03";
double x = atof(s1); // y = 1.03

char s2[] = "efg";
double y = atof(s2); // y = 0.0
```
Pointers
Address versus value

- In some situations, we will want to work with a variable's address in memory, rather than the value it contains...

Pointers
What are pointers?

- A pointer holds the address of another variable or function
Pointers

What do they do?

- A pointer allows us to indirectly access a variable (just like indirect addressing in assembly language)

Pointers

Why would I want to do that?

- Pointers make it possible to write a very short loop that performs the same task on a range of memory locations / variables.

Example: Data Buffer

```c
// Point to RAM buffer starting address
char *bufPtr = &buffer;

while ((DataAvailable()) && (receivedCharacter != '\0')) {
    // Read byte from UART and write it to RAM buffer
    ReadUart(bufPtr);
    // Point to next available byte in RAM buffer
    bufPtr++;
}
```
Pointers

Why would I want to do that?

Example: Data Buffer

RAM buffer allocated over a range of addresses (perhaps an array)

Pseudo-code:
1. Point arrow to first address of buffer
2. Write data from UART to location pointed to by arrow
3. Move arrow to point to next address in buffer
4. Repeat until data from UART is 0, or buffer is full (arrow points to last address of buffer)

Pointers

Where else are they used?

- Provide method to pass arguments by reference to functions
- Provide method to pass more than one piece of information out of a function
- Another means of accessing arrays and dealing with strings
- Used in conjunction with dynamic memory allocation (creating variables at runtime)
Pointers

How to Create a Pointer Variable

Syntax

```
    type *ptrName;
```

- In the context of a declaration, the `*` merely indicates that the variable is a pointer
- `type` is the type of data the pointer may point to
- Pointer usually described as “a pointer to `type`”

Example

```
    int *iPtr;      // Create a pointer to int
    int *iPtr, x;  // Create a pointer to int and an int
    float *fPtr1, *fPtr2; // Create 2 float pointers
```