

ENGG1003 - Tuesday Week 6

Strings & ASCII Codes
The Char Data Type
File I/O

Brenton Schulz

University of Newcastle

April 1, 2019

Quiz Content

- ▶ Because people have asked, the quiz will cover:
 - ▶ Everything up to static variables (Tuesday Week 5)
 - ▶ Flow control
 - ▶ Variables
 - ▶ Arithmetic
 - ▶ Assignment (= symbol)
 - ▶ Data types
 - ▶ Functions
 - ▶ Arrays
 - ▶ Static variables
- ▶ 1 A4 “memory aid” allowed
 - ▶ Same as final exams, double sided
- ▶ A “very brief” C syntax summary will be provided

Example

- ▶ Write a function which zeros the first N elements of an array of `ints`
 - ▶ Function prototype:

Example

- ▶ Write a function which zeros the first N elements of an array of `ints`
 - ▶ Function prototype:
 - ▶ `void zero(int *x, int N);`

Example

- ▶ Write a function which zeros the first N elements of an array of `ints`
 - ▶ Function prototype:
 - ▶ `void zero(int *x, int N);`
 - ▶ The value of N is needed because C won't tell you how long an array is *within the context of the function*
 - ▶ (Advanced) `sizeof(x)` will just be the size of the pointer - 4, or 8 bytes

Example

► Function definition:

```
1 // Zeros first N elements of x
2 void zero(int *x, int N) {
3     int i; // Array index loop counter
4     for(i = 0; i < N; i++)
5         x[i] = 0; // Use array syntax
6     return; // Optional
7 }
```

Other Examples

- ▶ Lets write and test these live...
- ▶ Write a function which:
 - ▶ Returns the sum of an array of length N
 - ▶ Returns the maximum value in an array of length N
 - ▶ Fills an array with integers between two given numbers `min` and `max`
 - ▶ Prototype:

```
void countArray(int *x,  
                int min, int max);
```
 - ▶ eg: `countArray(x, 10, 15)` sets:

```
x[] = {10, 11, 12, 13, 14, 15}
```

Strings

- ▶ A *string* is the “data type” which stores human-readable text
- ▶ C does not have a `string` data type
 - ▶ Most newer languages do
- ▶ In C, strings are stored in arrays of type `char`
 - ▶ Their “length” is defined by a terminating zero
 - ▶ Terminating means it goes after the last character

String Syntax

- ▶ Since C strings are arrays of type `char` they are declared with normal array syntax:

```
1 char name[200];
```

- ▶ The “size” of a string is known as the *length*
- ▶ Strings get terminated with a 0
 - ▶ Ok, technically NULL but its just a zero in memory
 - ▶ Often NULL is written `\0`
- ▶ The length is the number of bytes from (and including) the “start” pointer and the `\0`

Strings in Memory

- ▶ Each character is a single byte
- ▶ The terminating NULL is also a single byte
 - ▶ Be aware of this when declaring array sizes
- ▶ Everything beyond the NULL is “garbage”
 - ▶ Doesn't matter what the array size is
- ▶ The string "hello" would be stored as:

(Addresses are made up numbers)

Address:	10	11	12	13	14	15	16	17
Data:	??	h	e	l	l	o	\0	??

Using Strings

- ▶ String initialisation uses the syntax:

```
1 char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

Using Strings

- ▶ String initialisation uses the syntax:

```
1 char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- ▶ Terrible, isn't it?

Using Strings

- ▶ String initialisation uses the syntax:

```
1 char str[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
```

- ▶ Terrible, isn't it?
- ▶ If the string is *constant* you can do this:

```
1 char str[] = "This is a constant string.";
```

- ▶ Attempting to modify `str[]` will cause a crash
- ▶ The compiler automatically inserts the `\0`
- ▶ (Advanced) Strings between `"` and `"` are stored in the “program memory” and can't be modified for security reasons

Constants

- ▶ Aside: any variable which must not be modified can be declared `const`:

```
1 const char str[] = "This is a help message.";
```

- ▶ The `const` keyword causes a compiler error, instead of a segmentation fault, if you try to modify the variable
- ▶ You can do this to any data type, eg:

```
1 const float pi = 3.14159;
```

String Usage

- ▶ Normally strings are not initialised with `{ 'a' , 'b' , }` syntax
- ▶ Command line programs use a lot of constant strings
 - ▶ Text inside `printf()` is a constant string
- ▶ Most strings are read from the user or a file
 - ▶ In embedded systems they also come from communications peripherals like a UART

String Format Specifiers

- ▶ To `printf()` or `scanf()` a string use the `%s` format specifier
- ▶ Eg:

```
1 #include <stdio.h>
2 main() {
3     char str[] = "Hello world!";
4
5     // NB: Passing array pointer to function
6     // just uses the array name as argument
7     printf("%s\n", str);
8 }
```


Strings and `scanf()`

- ▶ Because an array name is a pointer to the first element **do not use &**

```
1 char str[1024];  
2 scanf("%s", str); // NO & SYMBOL
```

Strings and `scanf()`

- ▶ Because an array name is a pointer to the first element **do not use &**

```
1 char str[1024];  
2 scanf("%s", str); // NO & SYMBOL
```

- ▶ How much data does `%s` read?

Strings and scanf()

- ▶ Because an array name is a pointer to the first element **do not use &**

```
1 char str[1024];  
2 scanf("%s", str); // NO & SYMBOL
```

- ▶ How much data does %s read?
- ▶ Lets test with an example

```
1 #include<stdio.h>  
2 int main() {  
3     char str[1024];  
4     scanf("%s", str);  
5     printf("Read: %s\n", str);  
6 }
```

Strings and `scanf()`

- ▶ Experiment results:
 - ▶ `%s` stops at the first whitespace character
 - ▶ It ignores whitespace
 - ▶ Interpretation: `%s` reads a single word or number
- ▶ This changes if more complicated “pattern matching” is included in the `scanf()` argument
 - ▶ Beyond ENGG1003

String Function Example

Write a C function which returns the length of a string.

The function prototype is:

```
1 int stringLength(char *str);
```

(This is implemented in a library function, more on that after the example...)

String Library Functions

- ▶ The C standard string library is in `string.h`
- ▶ Previous example re-implemented `strlen()`
- ▶ Many other functions, consult an online reference for more details
 - ▶ `strcasecmp()` or `strcmp()` *might* be useful in your project for checking spelling
- ▶ Also be aware of `sprintf()`
 - ▶ Just like `printf()`
 - ▶ Stores output in a string, not `stdout`
 - ▶ Do we have time for an example?

ASCII Codes

- ▶ In C, constant letters in code are typed: `'a'`
- ▶ The single quote indicates that it is a *literal* letter, not a string

ASCII Codes

- ▶ In C, constant letters in code are typed: 'a'
- ▶ The single quote indicates that it is a *literal* letter, not a string
- ▶ But what is *actually* stored? Doesn't `char` just store a number from -128 to +127?

ASCII Codes

- ▶ In C, constant letters in code are typed: `'a'`
- ▶ The single quote indicates that it is a *literal* letter, not a string
- ▶ But what is *actually* stored? Doesn't `char` just store a number from -128 to +127?
- ▶ Yes! The ASCII standard converts “letters” to numbers

ASCII Codes

- ▶ The ASCII standard allocates a number to all letters, numbers, punctuation characters, and several “control” characters
- ▶ ASCII is used almost everywhere
 - ▶ The unicode standard UTF-8 is a superset of ASCII
- ▶ Lets check one out [here](#)
- ▶ Knowledge of ASCII and `char` processing in C is necessary for Programming Assignment 1

Char Variables

- ▶ There are two ways to interpret a `char` variable:
 - ▶ As a text character
 - ▶ As a number
- ▶ The `%c` format specifier tells `printf()` and `scanf()` to convert between ASCII characters and numbers
- ▶ Eg, this will read a character from `stdin` and store its ASCII value in `c`:

```
1 char c;  
2 scanf("%c", &c)
```

Char Variables

- ▶ What happens if you enter the number 5?

```
1 char c;  
2 scanf("%c", &c)
```

Char Variables

- ▶ What happens if you enter the number 5?

```
1 char c;  
2 scanf("%c", &c)
```

- ▶ It will store the number 53, as that is the ASCII code for '5'

ASCII Letters

- ▶ The project requires you to process text and identify letters
- ▶ The following table shows the numerical values which letters can occupy under the ASCII standard:

A	65		a	97
B	66		b	98
C	67		c	100
...			...	
Y	89		y	121
Z	90		z	122

Char Variables

- ▶ The numerical value of a character can be printed with `%d` and a cast:

```
1 char c;  
2 printf("%d", (int)c);
```

- ▶ Characters can be used in *arithmetic* without a problem, eg:

```
1 char c;  
2 scanf("%c", c);  
3 c = c - 65;
```

Char Variables

- ▶ The numerical value of a character can be printed with `%d` and a cast:

```
1 char c;  
2 printf("%d", (int)c);
```

- ▶ Characters can be used in *arithmetic* without a problem, eg:

```
1 char c;  
2 scanf("%c", c);  
3 c = c - 65;
```


Char Variables

- ▶ The code “`c = c - 65`” will convert each letter of the alphabet to a number with the allocation:

$$A = 0$$

$$B = 1$$

$$C = 2$$

...

$$Z = 25.$$

Char Variables

- ▶ The code “`c = c - 65`” will convert each letter of the alphabet to a number with the allocation:

$$A = 0$$

$$B = 1$$

$$C = 2$$

...

$$Z = 25.$$

- ▶ You use this in the project

char Example

Write a C program which reads a single char from the user and uses it to select from a text-based menu.

This is useful for Programming Assignment 1

char Example

Write a C program which reads a single char from the user and uses it to select from a text-based menu.

This is useful for Programming Assignment 1

- ▶ We should `printf()` a menu
- ▶ A single character should be read back
- ▶ Use `switch` to call off the appropriate function

char Example

- We can print a menu like this:

```
1 printf("Please select an option: \n");  
2 printf("a) Start a new game\n");  
3 printf("b) Load a saved game\n");  
4 printf("c) Options\n");  
5 printf("d) Quit\n\n");  
6 printf("Selection: ");
```

char Example

- We can print a menu like this:

```
1 printf("Please select an option: \n");  
2 printf("a) Start a new game\n");  
3 printf("b) Load a saved game\n");  
4 printf("c) Options\n");  
5 printf("d) Quit\n\n");  
6 printf("Selection: ");
```

- And read the user's input with:

```
1 char c;  
2 scanf("%c", &c); // &c, not a string
```

char Example

- ▶ We need to read user input *at least once*
- ▶ The user could make a mistake
- ▶ Lets use a `do { } while()`

char Example

- ▶ We need to read user input *at least once*
- ▶ The user could make a mistake
- ▶ Lets use a `do { } while()`
- ▶ What would be used in the condition?

char Example

- ▶ We need to read user input *at least once*
- ▶ The user could make a mistake
- ▶ Lets use a `do { } while()`
- ▶ What would be used in the condition?
- ▶ `c` needs to be `a`, `b`, `c`, or `d` to continue

char Example

- ▶ We need to read user input *at least once*
- ▶ The user could make a mistake
- ▶ Lets use a `do { } while()`
- ▶ What would be used in the condition?
- ▶ `c` needs to be `a`, `b`, `c`, or `d` to continue
- ▶ Couple of options:
 - ▶ Naive solution:

```
1 while (c != 'a' && c != 'b' && c != 'c' &&  
    c != 'd');
```

- ▶ Use knowledge of ASCII codes:

```
1 while (c < 'a' || c > 'd');
```

char Example

- Once input is taken lets use `switch()`:

```
1 switch(c) {  
2     case 'a': newGame(); break;  
3     case 'b': loadGame(); break;  
4     case 'c': options(); break;  
5     case 'd': quit(); break;  
6     default: printf("Unknown option %c\nPlease  
7         enter a, b, c, or d\n");  
8 }
```

char Example

Lets see the full program in Che...

File I/O

- ▶ So far: all input-output has been to `stdout` and from `stdin`
- ▶ These are known as *streams*
 - ▶ A “stream” is any communications channel which can provide and/or accept data
- ▶ The C file I/O library allows data to be read from, or written to, a file
- ▶ Files are stored on the computer’s hard drive (or USB flash drive, network drive, etc)

File I/O

- ▶ A stream is kept in a variable of type `FILE` *
- ▶ Read as “pointer to `FILE`” or “`FILE`-star”
- ▶ Three already exist in your C programs:
 - ▶ `stdin`
 - ▶ `stdout`
 - ▶ `stderr`
- ▶ Additional streams are declared like other variables, eg:

```
1 FILE *input, *output;
```

File I/O

- ▶ Before a file can be accessed you must *open* it with the `fopen()` function
- ▶ In order to open files you need two pieces of information:
 - ▶ The file's name
 - ▶ The data direction (mode)
 - ▶ Reading
 - ▶ Writing
 - ▶ Both

File I/O

- ▶ `fopen()`'s function prototype is:

```
1 FILE *fopen(const char *name, const char *mode);
```

- ▶ `const char *name` is a string holding the file's name
- ▶ `const char *mode` is a string describing the desired data direction
- ▶ Both of these can be passed as variable strings or hard-coded

File I/O

- ▶ The `*mode` argument can be one of the following:
 - ▶ `"r"` (reading)
 - ▶ `"r+"` (reading and writing)
 - ▶ `"w"` (writing)
 - ▶ `"w+"` (reading and writing, file truncated)
 - ▶ `"a"` (appending)
 - ▶ `"a+"` (reading and appending)
- ▶ Read documentation for details
- ▶ `fopen()` example:

```
1 FILE *input;  
2 input = fopen("data.txt", "r");
```

fclose() Errors

- ▶ The return value of `fopen()` is `NULL` on error
- ▶ Check it! Attempting to access a `NULL` stream will result in a segmentation fault!

```
1 FILE *input;  
2 input = fopen("data", "r");  
3 if(input == NULL) {  
4     perror("fopen()");  
5     return;  
6 }
```

- ▶ `perror()` prints a user-friendly error message

File I/O

- ▶ Once opened, a file can be accessed with:
 - ▶ `fscanf()`
 - ▶ `fprintf()`
- ▶ These functions behave just like `scanf()` and `printf()` except they take an extra argument:

```
1 int fscanf(FILE *stream, const char *format,  
    ...);
```

- ▶ The first argument is a `FILE *`
- ▶ The rest is identical to `printf()` and `scanf()`

File I/O - Position Indicators

- ▶ Concept: bytes in files have an address known as a *position indicator*
- ▶ The address is the number of bytes, starting at zero, from the start of the file
- ▶ Unless otherwise controlled, files are only read from and written to *sequentially*
- ▶ The position indicator automatically increments when a byte is read or written

File I/O - Position Indicators

- ▶ Some useful functions:
 - ▶ `ftell()` - Returns the position indicator
 - ▶ `fseek()` - Sets the position indicator
 - ▶ `feof()` - Returns TRUE if the position indicator is at the end of the file
- ▶ For example, to process data until the end of file is reached:

```
1 FILE *stream;  
2 // open file etc  
3 while(!feof(stream)) {  
4     // Read from file  
5     // Do stuff  
6 }
```

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

► Declare `FILE *input;`

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

- ▶ Declare `FILE *input;`
- ▶ Use `fopen()` to open it for reading

File I/O Example

Write a C program which opens a file, `test.txt`, and prints its contents to `stdout`, reading and writing one character at a time.

- ▶ Declare `FILE *input;`
- ▶ Use `fopen()` to open it for reading
- ▶ Write a loop which reads and writes characters until the whole file has been read
 - ▶ Read with: `fscanf(input, "%c", &c);`
 - ▶ Write with: `fprintf("%c", c);`

File I/O Example

Write a C program which opens a file, `input.txt`, and counts the number of times the string "the" appears.

(I might run out of time here, see how we got?)