COMPSCI 1XC3 - Computer Science Practice and Experience: Development Basics

Topic 10 - Strings, Formatting, and File I/O

NCC Moore

McMaster University

Summer 2021

Adapted from Chapters 8, 9 and 11 of C: How to Program 8th ed., Deitel & Deitel

Destringing and Restringing

String Manipulations

Advanced Formatting

Permanent Memory Interaction

A Quick Recap of Characters and Strings

In C, strings are encoded as as **character arrays**, which are *null terminated.*

```c
char foo[] = "bar";
```

In the above declaration, `foo` will be written into memory as:

...	'b'	'a'	'r'	'\0'	...

Remember:

- All strings are arrays of `char`s, which have a bit-width of 1 byte.
- Strings are delimited with “double quotes”.
- Characters are delimited with 'single quotes'.

The Importance of Null Termination

The null character has a very important function in C.

- You may have noticed that C does not “know” when an array ends, though it always knows when one begins.
- The null character `\0` is used to indicate the end of a string in all standard library functions.
- Using the fact of null termination in your own code is also a very good idea!
- A common pitfall is not allocating enough space in your character array for the null character.
  - If a string is missing it's null character, the functions in the standard string library will continue operating into memory space not allocated to the character array.
  - This will introduce smelly garbage data into your program, and may even cause a segfault!

Converting String to double

The C standard library `stdlib.h` defines `strtod()`, which converts decimal numbers to double-precision floating point numbers (or `double`s).

```
double strtod(const char *str, char **endPtr);
```

- `str` is a pointer to the string to be converted.
  - Leading whitespace will be ignored.
- `strtod` converts as much of the string as it can to double format, and then returns that double.
- `endPtr` is the address where `strtod` stores the address of the character that it stopped on.
- This is an example of a function using pointers to, in effect, *return two values*.

For Example...

```c
// Fig. 8.6: fig08_06.c
// Using function strtod
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   const char *string = "51.2% are admitted"; // initialize string
   char *stringPtr; // create char pointer

   double d = strtod(string, &stringPtr);

   printf("The string \"%s\" is converted to the\n", string);
   printf("double value %.2f and the string \"%s\"\n", d, stringPtr);
}
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

Fig. 8.6 | Using function `strtod`.

If `strtod()` fails to convert a string, it will return `0`.

Converting String to `int`

Similarly to `strtod()`, `stdlib.h` provides a function for converting strings to integers: `strtol()`

```
long int strtol(const char *str, char **endptr, int
    base);
```

- Key Differences:
  - the return type is an 8-byte `long int`.
  - `strtol()` accepts a third argument, the base of the number we are interpretting.
    - 0 ← Octal, Decimal or Hexadecimal
    - 2-32 ← The base indicated.
    - Bases higher than 10 use alphabetic characters in the same manner as Hexadecimal.

Variations on a Theme in D Minor

There are a number of conversion functions that work in a similar manner.

Function	Converts String To...
`strtod()`	`double`
`strtof()`	`float`
`strtol()`	`long int`
`strtoul()`	`unsigned long int`
`strtoll()`	`long long int`
`strtoull()`	`unsigned long long int`

- Note the absence of functions to convert to `int` and `short`
- This is possible directly using the unsafe `atoi()` function.
- `atoi()` is unnecessary, as `strtol()` can be easily type-cast to either `int` or `short`.

Converting Things to Strings

Now that we know how to read numeric values from strings, the question remains, how do we write numeric values into strings?

- Trick Question! We've been doing this all along!
- `printf()` writes to `stdout`, but `snprintf()` writes to a specified memory address.

```
int snprintf(char *str, size_t size, const char *format
    , ...);
```

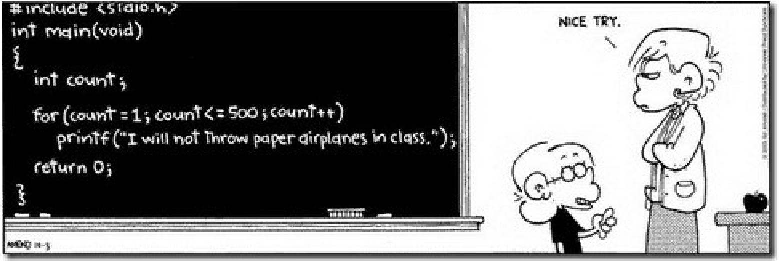
- `str` is a pointer to the memory address the characters will be written to
- `size` sets a maximum number of characters to write.
  - Remember to leave enough space in both of the above for null termination!
- Every argument past the second is used precisely the same as `printf()`.

Converting Things to Strings (cont.)

- The return value of `snprintf()` is the size of the format string after substitution.
  - Note, that this may be distinct from the size of the string written into memory.
  - This means that we can check to see if the string had to be truncated by comparing the input size to the output size.

```c
int i = 1337;
int size = (int)((ceil(log10(i))+1);
char *buffer = (char*) malloc(size*sizeof(char));
int j = snprintf(buffer, size, "%d", i);
if (j > size) {
  printf("Write operation was truncated!");
}
```

String.h: Manipulation Tactics



Data Duplication with `strcpy()`

Have you ever wanted to copy a string? Well `strcpy()` and `strncpy()` are the functions for you!

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

- Both take the string stored at `s2` and copy it to `s1`.
- The extra argument in `strncpy()` is similar to `size` in `snprintf()`.
  - `strncpy()` will copy *at most* `n` characters.
  - This guards against buffer overflow, making `strncpy()` the "safe version" of `strcpy()`.
- Once again, *you* must make sure:
  - `s1` is large enough to contain `s2`
  - `n` takes null termination into account.

Yet More Functions!

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

- `strcat()` usage is similar to `strcpy()`
  - The null character terminating `s1` is overwritten with the first character of `s2`.
- `strlen()` accepts a string and produces the number of characters in it, null character excluded.

My String's Better Than Your String... `strcmp`

Ever needed to tell if two strings are the same string? Try `strcmp()` on for size!

```
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
```

- Inputs are the same as previously
- Compares the characters in each string sequentially and returns:
  - `0` if the strings are the same.
  - A value less than 0 if s1 is "less than" s2
  - A value greater than 0 if s1 is "greater than" s2.
- In this context, strings are ordered by the ASCII values of their characters, using alphabetization rules.
- This is known as **Lexographical Ordering.**

A Table with Equivalent Python Operations

C function	Rough Python Equivalent
`strtol()`	`int(myString)`
`snprintf()`	`str(myInt)`
`strcpy()`	`copy.deepcopy(foo)`
`strcat()`	`foo + bar`
`strlen()`	`len(foo)`
`strcmp()`	[`==`, `<`, `>`, etc.]
`strchr()`	`foo.index(bar)`
`strtok()`	`foo.split(bar)`

Surely This is Generalizable.

Given that in C, data types as a concept are somewhat ephemeral, it should be the case that the above operations on `char*`s should be reasonably generalizable.

- `string.h` contains a set of memory operations which are highly similar to the operations on strings previously discussed.
- Rather than accepting and returning `char*` pointers, these functions use `void*` pointers.

C function	C String Equivalent	Rough Python Equivalent
`memcpy()`	`strcpy()`	`copy.deepcopy(foo)`
`memcmp()`	`strcmp()`	[==, $<$, $>$, etc.]
`memchr()`	`strchr()`	`foo.index(bar)`

Formatting: What can't it do?

Up to this point, we have only briefly touched on the advanced features of the string formatting tag `%`. We will discuss the following features:

- Rounding of floating point numbers, and displaying a specified number of decimal places.
- Aligning columns of numbers
- Right and Left Justification of outputs
- Exponential formats for floating point numbers
- Fixed field widths for various data types

All of the format specifiers we are about to discuss are used in `printf()`, `scanf()` and their cousins.

Format Specifiers for Integer Formats

Conversion specifier	Description
`d`	Display as a *signed decimal integer.*
`i`	Display as a *signed decimal integer.* [*Note:* The `i` and `d` specifiers are *different* when used with `scanf`.]
`o`	Display as an *unsigned octal integer.*
`u`	Display as an *unsigned decimal integer.*
`x` or `X`	Display as an *unsigned hexadecimal integer.* `X` causes the digits `0-9` and the *uppercase* letters `A-F` to be used in the display and `x` causes the digits `0-9` and the *lowercase* letters `a-f` to be used in the display.
`h`, `l` or `ll` (letter "ell")	Place *before* any integer conversion specifier to indicate that a `short`, `long` or `long long` integer is displayed, respectively. These are called **length modifiers**.

Fig. 9.1 | Integer conversion specifiers.

Printing Integers in Various Ways

```c
// Fig. 9.2: fig09_02.c
// Using the integer conversion specifiers
#include <stdio.h>

int main(void)
{
   printf("%d\n", 455);
   printf("%i\n", 455); // i same as d in printf
   printf("%d\n", +455); // plus sign does not print
   printf("%d\n", -455); // minus sign prints
   printf("%hd\n", 32000);
   printf("%ld\n", 2000000000L); // L suffix makes literal a long int
   printf("%o\n", 455); // octal
   printf("%u\n", 455);
   printf("%u\n", -455);
   printf("%x\n", 455); // hexadecimal with lowercase letters
   printf("%X\n", 455); // hexadecimal with uppercase letters
}
```

Fig. 9.2 | Using the integer conversion specifiers. (Part 1 of 2.)

Printing Integers in Various Ways (cont.)

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

Fig. 9.2 | Using the integer conversion specifiers. (Part 2 of 2.)

Printing Floating Point Numbers

Conversion specifier	Description
`e` or `E`	Display a floating-point value in *exponential notation*.
`f` or `F`	Display floating-point values in *fixed-point notation* (`F` is supported in the Microsoft Visual C++ compiler in Visual Studio 2015 and higher).
`g` or `G`	Display a floating-point value in either the *floating-point form* `f` or the exponential form `e` (or `E`), based on the magnitude of the value.
`L`	Place before any floating-point conversion specifier to indicate that a `long double` floating-point value should be displayed.

Fig. 9.3 | Floating-point conversion specifiers.

Example Floats

```c
// Fig. 9.4: fig09_04.c
// Using the floating-point conversion specifiers
#include <stdio.h>

int main(void)
{
   printf("%e\n", 1234567.89);
   printf("%e\n", +1234567.89); // plus does not print
   printf("%e\n", -1234567.89); // minus prints
   printf("%E\n", 1234567.89);
   printf("%f\n", 1234567.89); // six digits to right of decimal point
   printf("%g\n", 1234567.89); // prints with lowercase e
   printf("%G\n", 1234567.89); // prints with uppercase E
}
```

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

Fig. 9.4 | Using the floating-point conversion specifiers.

Floating Some Ideas...

- `e` and `f` show six digits of precision by default.
- `f` will always print at least one digit to the left of the decimal point.
- `g` will select `e` if the exponent would be greater than the specified precision (default 6) or less than -4, and `f` otherwise.
- `g` does not print trailing zeroes.
- `e` and `g` display *rounded* values, `f` displays *truncated* values.

Random Trivia: You can print the `%` character with either `%%`. Note that `\%` doesn't work.

Working with Field Widths

Ever been bummed out because getting C to print a table of values with properly aligned numbers is hard? Fear no more!

- Inserting an integer value between the `%` character and the format specifier sets the **field width**.
- Data will normally be *right justified* within this field.
- Field widths may be used with all format specifiers.
- If your field width is too narrow for your data, that data will “overhang” the specified width, rather than being truncated.

Field Widths, Baby

```c
// Fig. 9.8: fig09_08.c
// Right justifying integers in a field
#include <stdio.h>

int main(void)
{
   printf("%4d\n", 1);
   printf("%4d\n", 12);
   printf("%4d\n", 123);
   printf("%4d\n", 1234);
   printf("%4d\n\n", 12345);

   printf("%4d\n", -1);
   printf("%4d\n", -12);
   printf("%4d\n", -123);
   printf("%4d\n", -1234);
   printf("%4d\n", -12345);
}
```

Fig. 9.8 | Right justifying integers in a field. (Part 1 of 2.)

Field Widths, Baby (cont.)

```
    1
   12
  123
 1234
12345

   -1
  -12
 -123
-1234
-12345
```

Fig. 9.8 | Right justifying integers in a field. (Part 2 of 2.)

Field Widths with Precision

We can specify various things for different data types using the `.X` format tag.

- When applied to `int`s, leading zeros are included instead of spaces, to fill the field width.
- When applied to `float`s, this controls the number of decimal places that are displayed (i.e., the precision).
- When applied to strings, this sets the number of characters to display. The rest of the string will be truncated.
    - The `%s` tag looks for a terminating null character.
    - If you aren't careful, `% s` can produce a segfault!
    - Using `% s` instead of `% c` by mistake can also cause this.
    - Specifying a field width is an effective countermeasure against a non-null-terminated string.

Let's be Precise About This!

```c
// Fig. 9.9: fig09_09.c
// Printing integers, floating-point numbers and strings with precisions
#include <stdio.h>

int main(void)
{
   puts("Using precision for integers");
   int i = 873; // initialize int i
   printf("\t%.4d\n\t%.9d\n\n", i, i);

   puts("Using precision for floating-point numbers");
   double f = 123.94536; // initialize double f
   printf("\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f);

   puts("Using precision for strings");
   char s[] = "Happy Birthday"; // initialize char array s
   printf("\t%.11s\n", s);
}
```

Fig. 9.9 | Printing integers, floating-point numbers and strings with precisions. (Part 1 of 2.)

Let's be Precise About This! (cont.)

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+002
        124

Using precision for strings
        Happy Birth
```

Fig. 9.9 | Printing integers, floating-point numbers and strings with precisions. (Part 2 of 2.)

Field widths and precisions may be used together!

```
printf("%9.6f", 123.456789);
```

```
__123.456
```

(The underscores above were added to visualize the spaces)

Various Options

In addition, there are a number of flags that may be included to modify print format.

Flag	Description
- (minus sign)	*Left justify* the output within the specified field.
+	Display a *plus sign* preceding positive values and a *minus sign* preceding negative values.
space	Print a space before a positive value not printed with the + flag.
#	Prefix `0` to the output value when used with the octal conversion specifier `o`.
	Prefix `0x` or `0X` to the output value when used with the hexadecimal conversion specifiers `x` or `X`.
	Force a decimal point for a floating-point number printed with `e`, `E`, `f`, `g` or `G` that does *not* contain a fractional part. (Normally the decimal point is printed *only* if a digit follows it.) For `g` and `G` specifiers, trailing zeros are not eliminated.
`0` (zero)	Pad a field with *leading zeros*.

Fig. 9.10 | Format-control-string flags.

Remember Kids, Don't Drink and Code!

File Operations

The following five functions are the essential operations for reading from and writing to files.

- `fopen()`, `fclose()`, `fscanf()`, `fread()`, `fwrite()`, `fprintf`, `feof()`

Using these files requires knowledge of the `FILE` structure, which is defined in `<stdio.h>`. Think of it as a structure containing all the inforation relevant to a file that is currently being streamed.

- It will probably make more sense when we cover `struct` next week.
- This section assumes you at least vaguely remember how to do this in python.

fopen() for business!

Opening a file stream requires an invokation of `fopen()`.

```
FILE *fopen(const char *filename, const char *mode);
```

- As you may expect, `filename` is a string, containing the name of the file to be opened.
  - The file may be specified by either absolute or relative addressing.
- `mode` is a string specifying, among other things, whether the file will be opened in read or write mode.
- The output of this function is a `FILE` pointer to the data stream object. You don't have to worry about direct manipulation of this pointer, it is only interacted with via the file operation functions.

Modus Operandi

Mode	Description
`r`	Open an existing file for reading.
`w`	Create a file for writing. If the file already exists, *discard* the current contents.
`a`	Open or create a file for writing at the end of the file—i.e., write operations *append* data to the file.
`r+`	Open an existing file for update (reading and writing).
`w+`	Create a file for reading and writing. If the file already exists, *discard* the current contents.
`a+`	Open or create a file for reading and updating; all writing is done at the end of the file—i.e., write operations *append* data to the file.

Fig. 11.5 | File opening modes. (Part 1 of 2.)

fclose() but no cigar!

Just as a dynamically allocated pointer needs to be freed when it is no longer needed, a file stream must be closed when you're finished with it.

```
int fclose(FILE *stream)
```

- This one's pretty straightforward. Takes the file stream as an argument.
- Returns 0 on success, EOF on failure.

fscanf() is difficult to write a pun for...

If you want to treat your file like you are reading information from stdin, consider using fscanf() and fprintf().

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

- While the first argument is the file stream we are reading from, the rest of the arguments are used exactly like scanf().
- The return value is the number of input items successfully matched and assigned.
  - i.e., the number of format specifiers in the format string that were successful.

fprintf() is equally difficult to write a pun for...

```c
int fprintf(FILE *stream, const char *format, ...)
```

- Writes characters into a file in exactly the manner you would expect.
- First argument is the filestream.
- Returns the number of characters written if successful, and a negative number upon failure.
- nuff said.

fread() any good books lately?

If repeatedly invoking `fscanf()` isn't your style, perhaps you'd prefer writing chunks of files directly into arrays?

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
     *stream)
```

- `ptr` is a pointer to the memory block you're writing to. It needs to be at least as big as `size*nmemb`
- `size` is the size in bytes of each element to be read.
  - So, using `sizeof()` would be a good idea!
- `nmemb` is the number of elements to read, each the size of `size`.
- `stream` is, of course, the file stream.
- `fread()` returns the total number of elements successfully read.

fwrite() to Remain Silent

And in reverse...

```
size_t fwrite(const void *ptr, size_t size, size_t
    nmemb, FILE *stream)
```

- `ptr` is a pointer to the array of elements to be written.
- `size` is the size, in bytes, of each element of `ptr`
- `nmemb` is the number of elements to be written.
- `stream` is, obviously, our old friend the file stream.
- `fwrite()` returns the total number of elements successfully written to the file stream.

feof() World Cup!

So all this reading from files is fine, but how do we know when we're finished?

```
int feof(FILE *stream)
```

- feof() tests to see if the file stream has reached the end of the file.
- Input is the file stream (naturally)
- Output is non-zero if the end of the file has been reached, zero otherwise.

An Example!

```
// Fig. 11.6: fig11_06.c
// Reading and printing a sequential file
#include <stdio.h>

int main(void)
{
   FILE *cfPtr; // cfPtr = clients.txt file pointer

   // fopen opens file; exits program if file cannot be opened
   if ((cfPtr = fopen("clients.txt", "r")) == NULL) {
      puts("File could not be opened");
   }
   else { // read account, name and balance from file
      unsigned int account; // account number
      char name[30]; // account name
      double balance; // account balance

      printf("%-10s%-13s%s\n", "Account", "Name", "Balance");
      fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);

```

Fig. 11.6 | Reading and printing a sequential file. (Part 1 of 2.)

An Example! (cont.)

```
        // while not end of file
        while (!feof(cfPtr)  ) {
           printf("%-10d%-13s%7.2f\n", account, name, balance);
           fscanf(cfPtr, "%d%29s%lf", &account, name, &balance);
        }

        fclose(cfPtr); // fclose closes the file
     }
  }
```

```
Account   Name          Balance
100       Jones           24.98
200       Doe            345.67
300       White            0.00
400       Stone          -42.16
500       Rich           224.62
```

Fig. 11.6 | Reading and printing a sequential file. (Part 2 of 2.)

Cue Demo!

The Last Slide Comic