# CMSC 246: Systems Programming

Spring 2021
Instructor: Geoffrey Towell

## The Shell

- The “shell” is a program that runs other programs
    - An ordinary program, not a part of the OS (kernel)
- When you do things in Unix command line you do it through a shell
    - Lots of “flavors of shells”
        - defaults
            - bash on CS machines (also available: sh, tcsh, csh, git-shell, zsh
            - zsh on Macs (also bash, csh, dash, sh, tcsh)
            - powershell on Windows10
- Shells come with a small set of “builtins”: cs, ls, etc  (68 in bash, 100+ in zsh)
    - There are a lot of non-builtin (usually written in C)
        - more than 2500 on lab machines

## Unix Time!!!

- cd
    - absolute path
    - relative path
    - ~
- ls
    - flags: -l -a -r { -t -S }
- pwd
- PATHs
    - how to find executables
- more / less / cat
    - cat > file
- man
    - `man 3 cFunc`
        - the 3 says to show the man page from section 3 of the manual
            - section 3 contains C functions
        - Sadly man pages for C are NOT installed on powerpuff
        - They are installed on macs!

# More Unix … “the PATH”

- The one true PATH …
  - does not exist in Unix
- The PATH is the place where Unix looks for executable programs
  - /usr/local/sbin:/usr/local/bin:/usr/bin:/usr/lib/jvm/default/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl:/usr/bin
  - /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
- When you give a name like `a.out`, Unix looks in the path to find the first executable with that name, then runs it.
  - Order of directories in PATH matters!
- Override path search by putting directory information before the executable
  - so `./a.out` says “look only in the current directory for the executable a.out”

- Properly this is the “Executable path”. Other paths exists.

## Input

- `scanf()` is the C library's counterpart to `printf`.
  - somewhat akin to Java Scanner.
- Syntax for using `scanf()`

`scanf(<`***`format-string>`***`, <`***`variable-reference(s)>`***`)`

- Example: read an integer value into an `int` variable `data`.

```
scanf("%d", &data); //read an integer; store into data
```

- The `&` is a reference operator. More on that later!

## Reading Input

- Reading a `float`:

```
float x;
scanf("%f", &x);
```

  - `"%f"` tells `scanf` to look for an input value in `float` format (the number may contain a decimal point, but doesn’t have to).

- Reading a `double`:

```
double x;
scanf("%lf", &x);
```

- Reading an int:

```
int x;
scanf("%d", &x);
```

- Reading a long:

```
long x;
scanf("%ld", &x);
```

## Standard Input & Output Devices

- In Linux the standard I/O devices are, by default, the keyboard for input, and the terminal console for output.

- input and output in C, if not specified, is always from the standard input and output devices. That is,
    - printf() outputs to the standard output device
        - default: the terminal
    - scanf() always inputs from the standard input device
        - default: the keyboard

- Later, you will see how these can be reassigned/redirected to other devices.

## Program: Convert  Fahrenheit to Celsius

- The `c2f.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

- The program will allow temperatures that aren’t integers.

# Program: Convert Fahrenheit to Celsius f2c.c

```
#include <stdio.h>

int main(void)
{
  float f, c;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &f);

  c = (f - 32) * 5.0/9.0;

  printf("Celsius equivalent: %.1f\n", c);

  return 0;
} // main()
```

For scanf:
%f ==> float
%d ==> double

%lf ==> double
%ld ==> long

printf: just %f, %d

Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

## Improving ctof.c

Look at the following command:

```
c = (f - 32) * 5.0/9.0;
```

First, 32, 5.0, and 9.0 should be floating point values: 32.0, 5.0, 9.0

Second, by default, in C, they will be assumed to be of type `double`
Instead, we should write

```
c = (f - 32.0f) * 5.0f/9.0f;
```

What about using constants/magic numbers?

## Defining constants - macros

```
#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)
```

So we can write:

```
c = (f – FREEZING_PT) * SCALE_FACTOR;
```

When a program is compiled, the preprocessor replaces each macro by the value that it represents.

During preprocessing, the statement

```
c = (f – FREEZING_PT) * SCALE_FACTOR;
```

will become

```
c = (f – 32.f) * 5.0f/9.0f;
```

This is a safer programming practice.

## Program: Convert Fahrenheit to Celsius `ctof.c`

```
#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f/9.0f)

int main(void)
{
  float f, c;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &f);

  c = (f - FREEZING_PT) * SCALE_FACTOR;

  printf("Celsius equivalent: %.1f\n", c);

  return 0;
} // main()
```

Sample program output:

```
      Enter Fahrenheit temperature: 212
      Celsius equivalent: 100.0
```

# Input from the command line

```c
#include <stdio.h>
#include <stdlib.h>  // needed for atof

#define GALLONS_PER_LITER   0.2641
#define KILOMETERS_PER_MILE 1.609

int main(int argc, char const *argv[])
{
    if (argc < 2) {
        printf("Usage: %s number\n", argv[0]);
        printf("       where: number is a US style MPG estimate\n");
        return 0;
    }
    double mpg = atof(argv[1]);
    double lp100km = (1 / mpg) * (1 / GALLONS_PER_LITER) * (1 / KILOMETERS_PER_MILE) * 100;
    printf("%5.2f liters per 100km \n", lp100km);
    return 0;
}
```

# Shooting yourself in the foot

- APL
    - You shoot yourself in the foot and then spend all day figuring out how to do it in fewer characters.
    - You hear a gunshot and there's a hole in your foot, but you don't remember enough linear algebra to understand what happened.
    - @#&^$%&%^ foot
- C
  - You shoot yourself in the foot and then nobody else can figure out what you did.

Java

- You write a program to shoot yourself in the foot and put it on the Internet. People all over the world shoot themselves in the foot, and everyone leaves your website hobbling and cursing.
- You amputate your foot at the ankle with a fourteen-pound hacksaw, but you can do it on any platform.

- Lisp
    - You shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot yourself in the appendage which holds the gun with which you shoot...
    - You attempt to shoot yourself in the foot, but the gun jams on a stray parenthesis.
- Linux
    - You shoot yourself in the foot with a Gnu.

Perl

- You separate the bullet from the gun with a hyperoptimized regexp, and then you transport it to your foot using several typeglobs. However, the program fails to run and you can't correct it since you don't understand what the hell it is you've written.
- You stab yourself in the foot repeatedly with an incredibly large and very heavy Swiss Army knife.
- You shoot yourself in the foot and then decide it was so much fun that you invent another six completely different ways to do it.

Python

- You shoot yourself in the foot and then brag for hours about how much more elegantly you did it than if you had been using C or (God forbid) Perl.

◦

## Keywords

- The following ***keywords*** can't be used as identifiers:

```
auto      enum      restrict*  unsigned
break     extern    return     void
case      float     short      volatile
char      for       signed     while
const     goto      sizeof     _Bool*
continue  if        static     _Complex*
default   inline*   struct     _Imaginary*
do        int       switch
double    long      typedef
else      register  union
```

- Keywords (with the exception of `_Bool`, `_Complex`, and `_Imaginary`) must be written using only lower-case letters.
- Names of library functions (e.g., `printf`) are also lower-case.

# If and Switch statements in C

- A compound statement has the form

  `{` *statements* `}`

- In its simplest form, the `if` statement has the form

  `if (` *expression* `)` *compound/statement*

- An `if` statement may have an `else` clause:

  `if (` *expression* `)` *compound/statement* `else` *compound/statement*

- Most common form of the `switch` statement:

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

## Arithmetic Operators

- C provides five binary ***arithmetic operators***:

| | |
|---|---|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| % | remainder |

- An operator is ***binary*** if it has two operands.
- There are also two ***unary*** arithmetic operators:

| | |
|---|---|
| + | unary plus |
| - | unary minus |

## Logical Expressions

- Several of C’s statements must test the value of an expression to see if it is “true” or “false.”
- In many programming languages, an expression such as `i < j` would have a special “Boolean” or “logical” type.
- In C, a comparison such as `i < j` yields an integer: either 0 (false) or 1 (true).

# Relational Operators

- C's ***relational operators***:

| | |
|---|---|
| `<` | less than |
| `>` | greater than |
| `<=` | less than or equal to |
| `>=` | greater than or equal to |

- C provides two ***equality operators***:

| | |
|---|---|
| `==` | equal to |
| `!=` | not equal to |

- More complicated logical expressions can be built from simpler ones by using the ***logical operators***:

| | |
|---|---|
| `!` | logical negation |
| `&&` | logical *and* |
| `||` | logical *or* |

These operators produce 0 (false) or 1 (true) when used in expressions.

## Logical Operators

- Both `&&` and `||` perform "short-circuit" evaluation: they first evaluate the left operand, then the right one.
- If the value of the expression can be deduced from the left operand alone, the right operand isn't evaluated.
- Example:

```
(0 != i) && (j / i > 0)
```

`(0 !=` i`)` is evaluated first. If `i` isn't equal to 0, then `(j / i > 0)` is evaluated.

- If `i` is 0, the entire expression must be false, so there's no need to evaluate `(j / i > 0)`. Without short-circuit evaluation, division by zero would have occurred.
- if (i=5) … is LEGAL in C!
  - it returns 5, which is NOT 0 so TRUE
  - Best to always put constants on LHS of comparison

## Relational Operators & Lack of Boolean Watch out!!!

- The expression

  ```
  i < j < k
  ```

  is legal, but does not test whether `j` lies between `i` and `k`.
- Since the `<` operator is left associative, this expression is equivalent to

  ```
  (i < j) < k
  ```

  The 1 or 0 produced by `i < j` is then compared to `k`.
- The correct expression is `i < j && j < k`.

# Loops

- The `while` statement has the form

  `while (` *expression* `)` *statement*

- General form of the `do` statement:

  `do` *statement* `while (` *expression* `) ;`

- General form of the `for` statement:

  `for (` *expr1* `;` *expr2* `;` *expr3* `)` *statement*

  *expr1*, *expr2*, and *expr3* are expressions.

- Example:

```
for (i = 10; i > 0; i--)
  printf("T minus %d and counting\n", i);
```

- Variables can be declared within for

```
for (int i = 0; i < n; i++)
  …
```

## The **`printf`** Function

- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

## The **`printf`** Function

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.
- Too many conversion specifications:

```
printf("%d %d\n", i);   /*** WRONG ***/
```

- Too few conversion specifications:

```
printf("%d\n", i, j);   /*** WRONG ***/
```

## The **`printf`** Function

- Compilers aren’t required to check that a conversion specification is appropriate.
- If the programmer uses an incorrect specification, the program will produce meaningless output:

```
printf("%f %d\n", i, x);  /*** WRONG ***/
```

# Conversion Specifications

- A conversion specification can have the form `%`*m*`.`*pX* or `%-`*m*`.`*pX*, where *m* and *p* are integer constants and *X* is a letter.
- Both *m* and *p* are optional; if *p* is omitted, the period that separates *m* and *p* is also dropped.
- In the conversion specification `%10.2f`, *m* is 10, *p* is 2, and *X* is `f`.
- In the specification `%10f`, *m* is 10 and *p* (along with the period) is missing, but in the specification `%.2f`, *p* is 2 and *m* is missing.

# Conversion Specifications

- The ***minimum field width***, *m*, specifies the minimum number of characters to print.
- If the value to be printed requires fewer than *m* characters, it is right-justified within the field.
    - `%4d` displays the number 123 as `•123`. (• represents the space character.)
- If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size.
- Putting a minus sign in front of *m* causes left justification.
    - The specification `%-4d` would display 123 as `123•`.

## Conversion Specifications

- The meaning of the ***precision***, *p*, depends on the choice of *X*, the ***conversion specifier***.
- The `d` specifier is used to display an integer in decimal form.
  - *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary).
  - If *p* is omitted, it is assumed to be 1.

## tprintf.c

```c
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

# Lab 2/18

- Write a program that
  - takes an integer as input from the keyboard (or the command line)
  - calculates the  Nth Fibonacci number (N is the integer input)
  - Use printf to write a nicely formatted table like:

```
3     1      1      2    2.000000
4     1      2      3    1.500000
5     2      3      5    1.666667
6     3      5      8    1.600000
7     5      8     13    1.625000
8     8     13     21    1.615385
9    13     21     34    1.619048
```

  - Columns are: index, fibb(n-2), fibb(n-1), fibb(n), fibb(n)/fibb(n-1)

## Escape Sequences

- The `\n` code that used in format strings is called an ***escape sequence***.
- Escape sequences enable strings to contain nonprinting (control) characters and characters that have a special meaning (such as ").
- A partial list of escape sequences:

| | |
|---|---|
| Alert (bell) | `\a` |
| Backspace | `\b` |
| New line | `\n` |
| Horizontal tab | `\t` |

## Escape Sequences

- A string may contain any number of escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

- Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

## How **`scanf`** Works

- `scanf` tries to match groups of input characters with conversion specifications in the format string.
- For each conversion specification, `scanf` tries to locate an item of the appropriate type in the input data, skipping blank space if necessary.
- `scanf` then reads the item, stopping when it reaches a character that can't belong to the item.
    - If the item was read successfully, `scanf` continues processing the rest of the format string.
    - If not, `scanf` returns immediately.

## How **`scanf`** Works

- As it searches for a number, `scanf` ignores ***white-space characters*** (space, horizontal and vertical tab, form-feed, and new-line).
- A call of `scanf` that reads four numbers:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- The numbers can be on one line or spread over several lines:

```
   1
-20   .3
   -4.0e3
```

- `scanf` sees a stream of characters (¤ represents new-line):

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```
(`s` = skipped; `r` = read)

- `scanf` “peeks” at the final new-line without reading it.

## How `scanf` Works

- When asked to read an integer, `scanf` first searches for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit.
- When asked to read a floating-point number, `scanf` looks for
  - a plus or minus sign (optional), followed by
  - digits (possibly containing a decimal point), followed by
  - an exponent (optional). An exponent consists of the letter `e` (or `E`), an optional sign, and one or more digits.
- `%e`, `%f`, and `%g` are interchangeable when used with `scanf`.

## How **`scanf`** Works

- When `scanf` encounters a character that can't be part of the current item, the character is "put back" to be read again during the scanning of the next input item or during the next call of `scanf`.

## How **`scanf`** Works

- Sample input:

```
1-20.3-4.0e3¤
```

- The call of `scanf` is the same as before:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Here’s how `scanf` would process the new input:
  - `%d`. Stores 1 into `i` and puts the `-` character back.
  - `%d`. Stores –20 into `j` and puts the `.` character back.
  - `%f`. Stores 0.3 into `x` and puts the `-` character back.
  - `%f`. Stores –4.0 × 103 into `y` and puts the new-line character back.

## Ordinary Characters in Format Strings

- When it encounters one or more white-space characters in a format string, `scanf` reads white-space characters from the input until it reaches a non-white-space character (which is "put back").
- When it encounters a non-white-space character in a format string, `scanf` compares it with the next input character.
    - If they match, `scanf` discards the input character and continues processing the format string.
    - If they don't match, `scanf` puts the offending character back into the input, then aborts.

## Ordinary Characters in Format Strings

- Examples:
  - If the format string is `"%d/%d"` and the input is `•5/•96`, `scanf` succeeds.
  - If the input is `•5•/•96` , `scanf` fails, because the `/` in the format string doesn’t match the space in the input.
- To allow spaces after the first number, use the format string `"%d /%d"` instead.

## Confusing **`printf`** with **`scanf`**

- Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two.
- One common mistake is to put `&` in front of variables in a call of `printf`:

```
printf("%d %d\n", &i, &j);  /*** WRONG ***/
```

## Confusing **`printf`** with **`scanf`**

- Incorrectly assuming that `scanf` format strings should resemble `printf` format strings is another common error.
- Consider the following call of `scanf`:

```
scanf("%d, %d", &i, &j);
```

  - `scanf` will first look for an integer in the input, which it stores in the variable `i`.
  - `scanf` will then try to match a comma with the next input character.
  - If the next input character is a space, not a comma, `scanf` will terminate without reading a value for `j`.

# Confusing **`printf`** with **`scanf`**

- Putting a new-line character at the end of a `scanf` format string is usually a bad idea.
- To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character.
- If the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character.
- A format string like this can cause an interactive program to “hang.”

## Program: Adding Fractions

- The `addfrac.c` program prompts the user to enter two fractions and then displays their sum.
- Sample program output:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

## addfrac.c

```
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 *denom1;
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n",result_num, result_denom)

  return 0;
}
```