CMSC 246 Systems Programming

Fall 2019
Bryn Mawr College

Unix Time!!!

- cd
  - absolute path
  - relative path
  - ~
- ls
  - flags: -l -a -r { -t -S }
- pwd
- more / less / cat
  - cat > file
- man
- apropos & which

The **`printf`** Function

- The `printf` function must be supplied with a ***format string***, followed by any values that are to be inserted into the string during printing:

  `printf(`*string*`, `*expr*`1, `*expr*`2, …);`

- The format string may contain both ordinary characters and ***conversion specifications***, which begin with the `%` character.
- A conversion specification is a placeholder representing a value to be filled in during printing.
  - `%d` is used for `int` values
  - `%f` is used for `float` values

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 ***/
```

- If the programmer uses an incorrect specification, the program will produce meaningless output:

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

tprintf.c

```c
#include <stdio.h>
int main(void)
{
    int i=40;
    float f=839.21;
    printf("%d|%5d|%05d|%5.3d|\n", i, i, i, i);
    printf("%f|%10.3f|%10.3e|%010.3f|\n",f,f,f,f);
    printf("\nRight\n");
    printf("%10s%10s%10s\n%10s%10s%10s\n", "Item", "Unit", "Purchase", "", "Price", "Date");
    printf("\nLeft\n");
    printf("%-10s%-10s%-10s\n%-10s%-10s%-10s\n", "Item", "Unit", "Purchase", "", "Price", "Date");
}
```

```
40|   40|00040|  040|
839.210022|   839.210| 8.392e+02|00000839.2|
Right
      Item      Unit  Purchase
                Price      Date

Left
Item      Unit      Purchase
          Price     Date
```

Escape Sequences beyond \n

- Another common escape sequence is `\"`, which represents the `"` character:

```
printf("\"Hello!\"");
  /* prints "Hello!" */
```

- To print a single `\` character, put two `\` characters in the string:

```
printf("\\");
  /* prints one \ character */
```

`Others \t – tab \b – backspace \a – alarm`

Input

- `scanf()` is the C library's counterpart to `printf`.
- 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`:

```
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).

Standard Input & Output Devices

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

- Thus, input and output in C, if not specified, is always from the standard input and output devices. That is,

  `printf()` always outputs to the terminal console

  `scanf()` always inputs from 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 `ctof.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()
```

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 32.0f
#define SCALE 1.8f
#define CC(v) (FREEZING + SCALE*v)

int main(void)
{
    float f, c;
    printf("Enter a Celcius temperature: ");
    scanf("%f", &c);  // use %lf for double
    f = CC(c);
    printf("%d: Celcius: %.1f  Fahrenheit: %.1f\n", i, c, f);
    return 0;
}
```

Sample program output:

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

Identifiers

- Names for variables, functions, macros, etc. are called ***identifiers***.
- An identifier may contain letters, digits, and underscores, but must begin with a letter or underscore:

  ```
  times10  get_next_char  _done
  ```

  It's usually best to avoid identifiers that begin with an underscore.
- Examples of illegal identifiers:

  ```
  10times  get-next-char
  ```

Identifiers

- C is ***case-sensitive:*** it distinguishes between upper-case and lower-case letters in identifiers.
- For example, the following identifiers are all different:

  `job  joB  jOb  jOB  Job  JoB  JOb  JOB`
- Convention (for this class — at least)
  - Use an upper-case letter to begin each word within an identifier:
    `symbolTable  currentPage  nameAndAddress`
  - Use all upper case separated by _ for defines
    `FREEZING_POINT`
- C places no limit on the maximum length of an identifier.

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*

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:

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

  `(i != 0)` 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.

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.