CMSC 246: Systems Programming

Spring 2021

Feb 22

Unix — Paths and IO

- absolute paths
    - start with /
        - the ‘root’ of the directory tree
- relative paths
    - anything that is not absolute
    - “relative” meaning w.r.t your current location
        - “.” here
        - “..” one directory up

```
pwd
   /home/gtowell
cd 246
cd ./246
cd ../gtowell/246
cd ../gtowell/../gtowell/246
```

```
#suppose a file named aaa
#exists in /home/gtowell
pwd
   /home/gtowell/246
cp ../aaa aaa
cp ./../aaa aaa
cp ../../gtowell/aaa aaa
cp /home/gtowell/aaa aaa
cp ../aaa .
```

IO redirection

cat with no args reads from stdin

CTRL-d to end input

- On Thursday
  - “standard in defaults to keyboard, standard out to console.”
- Override sidin, stdout, stderr
  - Pipes | (only stdin and stdout)
    - showed in first class
      - before a func, standard input
      - after a func, standard output
  - Overrides < >
    - always after
    - < takes standard input
    - > takes standard output, writes to a file
    - >> takes standard output, appends to file
    - 2> takes standard error
- Examples: cat, wc and echo

```
 ~ % cat > aaa
dfjds dfdsf rte dfsd
dfdsf fsdf sdfsd kt
dsfsdf
 ~ % wc aaa
       3       9      48 aaa
 ~ % wc < aaa
       3       9      48
 ~ % cat aaa | wc
       3       9      48
 ~ % echo "this is a test"
this is a test
 ~ % echo "this is a test" > bbb
 ~ % cat bbb
this is a test
 ~ % wc < bbb
       1       4      15
 ~ % wc < bbb > bbbwc
 ~ % cat bbbwc
       1       4      15
 ~ %
```

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.
- You can overwrite any library function … don't

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.

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
      if (1==2) {
            printf("Unlikely");
      }
```

is equivalent to

```
      if (1==2)
            printf("Unlikely")'
```

- 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! (and is almost always a mistake)
  - it sets i to 5 AND returns 5 which is NOT 0 so TRUE
  - Best practice: always put constants on LHS of comparison
    - `if (5=i) // NOT LEGAL`

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|compound*

- General form of the `do` statement:

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

- General form of the `for` statement:

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

  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++)
  …
```

`for (;;;) {}` is equivalent to `while (1) {}`

The **`printf`** Function

- No string concatenation within printf
  ```
  printf("AAA" + “BBB"); //WRONG
  ```
- Ordinary characters in a format string are printed as they appear in the string;
- Conversion specifiers start with %
  - 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
  ```

Every conversion specifier can include its width. e.g. `%10d`

`%f` can specify precision as well `%10.2f`

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.
- Java String.format() does check for match
- 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 ***/
```

Making a well formatted table

- printf used a monospaced font
  - decide number of characters each column in table needs
  - for floating point number decide correct precision
    - eg. for money: 2 after the decimal point
  - Always use printf formatting

```
printf("%3s%3s\n", "a", "b");
printf("  a  b\n");
```

  - Use #define for format strings

My Fibonacci program

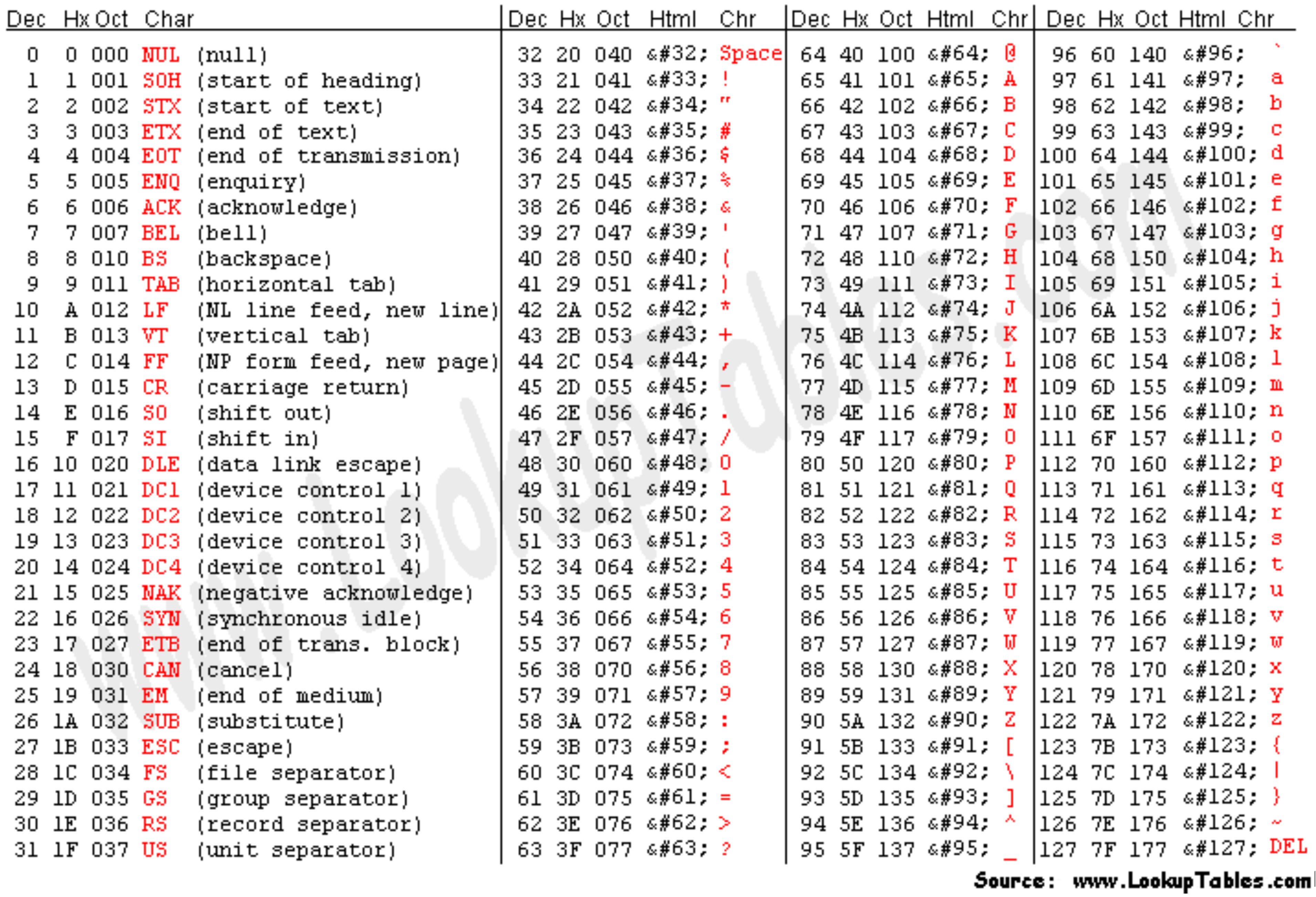
```c
#include <stdio.h>
#define HEADER "%6s%9s%9s%9s%9s\n"
#define DATA   "%6d%9d%9d%9d%9.5f\n"

void doTheFibb(int maxVal) {
    int f1 = 1;
    int f2 = 1;
    int n = 2;
    printf(HEADER, "", "fibb", "fibb", "fibb", "golden");
    printf(HEADER, "index", "n -2", "n -1", "n", "mean");
    while (f1 < maxVal)
    {
        n++;
        int f3 = f2 + f1;
        printf(DATA, n, f1, f2, f3, (float)f3 / f2);
        f1 = f2;
        f2 = f3;
    }}
int main(int argc, char const *argv[]) {
    doTheFibb(100000);
    return 0;
}
```

Ints and Chars

- Ints and chars in C can be used fairly interchangably
- A char is an int stored in a single byte
- The int representation of a char is from the ASCII table
- `'a' == 'b'-1`

```
int main(void)
{
    while (1) {

    char c = getchar();
        if (EOF==c)
            break;
        int i = (int)c;
        char cc = (char)i;
        printf("before:%c as int:%d after:%c\n", c, i, cc);
        printf("Printing a char:  as char:%c  as int:%d\n", c, c);
        printf("Printing an int:  as char:%c  as int:%d\n", i, i);
    }
}
```

ASCII Table



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH (start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX (start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX (end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT (end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ (enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK (acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL (bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS (backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB (horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF (NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT (vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF (NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR (carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO (shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI (shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE (data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1 (device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2 (device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3 (device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4 (device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK (negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN (synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB (end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN (cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM (end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB (substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC (escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS (file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS (group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS (record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US (unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Integer Types

- byte, short, int, long,
- Integer representation – "twos complement"
- so 1+maxPositive ==> maxNegative
- unsigned keyword – positive only numbers
  - 1+maxPositive==>0;

If there were 4 bit numbers

base 10	in bits
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
-8	1000
-7	1001

LAB

- Write a program with two loops:
  - First loop:reproduce the “dec” and “char” columns of the ASCII table for decimal values 33 - 126
  - Second loop. A single loop from 0..25 to produce the following table:

    -
      ```
      0  A  a
      1  B  b
      2  C  c
      …
      25  Z  z
      ```
  - You can do this using your knowledge of addition and the ASCII table.

- This program should have exactly 2 printf statements.

More on links

- ln [-s] filename linkname

- ls -l
- ls -i – show the inode

- Change aaaa both hard and soft have the change
- Delete aaaa
  - hard
    - No change!
  - soft
    - dead link!

Create a file named “aaaa” and put from keyboard into it

```
gtowell@mil:~$ cat > aaaa
this is a test
gtowell@mil:~$ ln aaaa aaaa_hard
gtowell@mil:~$ ln -s aaaa aaaa_soft
gtowell@mil:~$ ls -l aaaa*
-rw-r--r-- 2 gtowell faculty 15 Feb 19 18:49 aaaa
-rw-r--r-- 2 gtowell faculty 15 Feb 19 18:49
aaaa_hard
lrwxrwxrwx 1 gtowell faculty  9 Feb 19 18:49 aaaa
-> aaaa_soft
gtowell@mil:~$ ls -i aaaa*
169088825 aaaa  169088825 aaaa_hard
169088827 aaaa_soft
gtowell@mil:~$ rm aaaa
gtowell@mil:~$ cat aaaa_hard
thatt this is a test
gtowell@mil:~$ cat aaaa_soft
cat: aaaa_soft: No such file or directory
```

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.

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`.

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

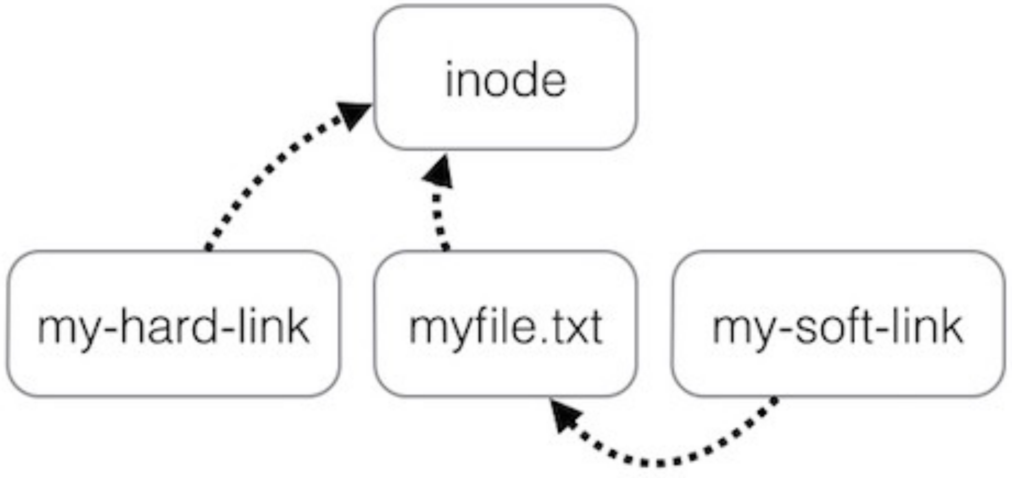
```
/* Adds two fractions */

#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;
}
```

Typical Unix directories

- / the beginning - the root
- /bin — executables
- /home — user directories
- /lib — libraries
    - parts of executables
    - usually a .so extension eg libc.so
        - this is the library that from “gcc -lc -xc xxx.c”
- /usr —
    - things that are also in /
        - /usr/bin, /usr/include, …
    - /usr/local — stuff NOT in standard UNIX …
- /proc
    - NOT actual files
        - /proc/cpuinfo, /proc/stat

Files and Hard/Soft Links

- in addition to files and
- directories, Unix has “links”
- Hard links
    - only to files
    - only within file systems
    - effectively creates a second user of disk space
- Soft links
    - files and directories
    - links to the file itself, not the disk space
        - dangling links


inode
my-hard-link
myfile.txt
my-soft-link