CMSC 246: Systems Programming

Spring 2021
Instructor: Geoffrey Towell

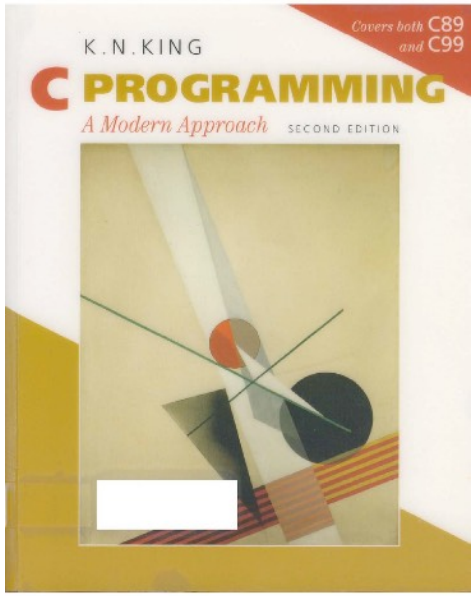
Important Info

- Class Web Page
  - http://cs.brynmawr.edu/cs246
  - All assignments
  - Lecture Notes
    - Should be posted before class
  - Important dates, etc
- Moodle — Nothing
- 2 mid-terms and a final
- Weekly Programming assignments
- Labs:
  - Rather than a single weekly event, there will be a problem to solve. You may, and are encouraged to work in groups on these.

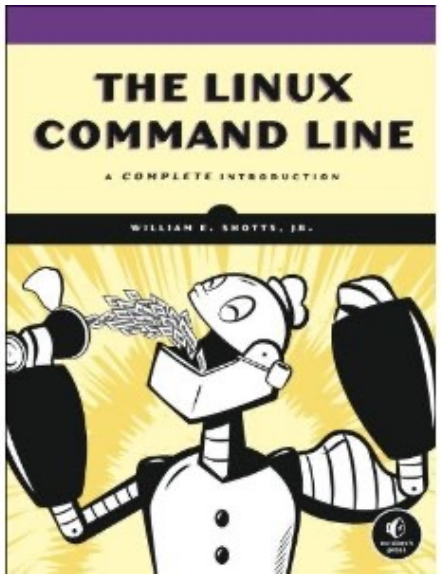
First Things

- CS account
  - Make sure you can log in to CS Unix servers
    - ssh YOUR_UNIX_NAME@powerpuff.cs.brynmawr.edu
  - If you cannot, let me know ASAP
- Software: the unix command line!!! (and gcc)
- You should have received from me a description of how to set up your laptops.
  - If your have not followed these directions do so ASAP.

Goals



- Learn Unix/Linux (CLI, not WIMP!)
  - Windows, Icons, Menus, Pointer
  - Command Line Interface
- Learn C
- Learn Linux tools



Why command line tools?

- Flexibility and Power!

- Problem: get a list of all “included” files in C programs I wrote in recently.
    - These files are all in subdirectories from a particular directory
    - First solution:
        - grep -R --include “*.c" include
            - This was really long and repetitive — how long?
                - grep -R --include “*.c" include | wc
    - Problem duplicates
        - after some thought
        - grep -R --include “*.c" include | sed 's/^.*\(:.*\).*$/\1/' | sort | uniq

Evolution of C

Algol60

Designed by an international committee, 1960

CPL

Combined Programming Language
Cambridge & Univ. of London, 1963
Was an attempt to bring Algol down To earth and retail contact with the Realities of an actual computer.
Features:

- Big
- Too many features
- Hard to learn
- Intended for numerical as well as non-numerical applications

BCPL

Basic CPL
Designed by Martin Richards, Cambridge 1967
Intended as a tool for writing compilers.
Designed to allow for separate compilation.
Features:

- Typeless language (only binary words)
- Introduced static variables
- Compact code
- Provodes access to address of data objects
- Stream-based I/O

B

Designed by Ken Thompson, Bell Labs 1970
A true forerunner of C
Features:

- Typeless (with floating pt. capabilities
- Designed for separate compilation
- Easily implementable
- Pre-processor facility
- Expensive library

C

1971-72
Developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others.
C is a by-product of UNIX.
Ritchie began to develop an extended version of B. He called his language NB ("New B") at first.
As the language began to diverge more from B, he changed its name to C.
The language was stable enough by 1973 that UNIX could be rewritten in C.

K&R C

Described in Kernighan and Ritchie, *The C Programming Language* (1978)
De facto standard
Features:

- Standard I/O Library
- long int data type
- Unsigned int data type
- Compound assignment operators

C89/C90

ANSI standard X3.159-1989
Completed in 1988
Formally approved in December 1989
International standard ISO/IEC 9899:1990
A superset of K&R C
Heavily influenced by C++, 1979-83

- Function prototypes
- void pointers
- Modified syntax for parameter declarations
- Remained backwards compatible with K&R C

C99

International standard ISO/IEC 9899:1999
Incorporates changes from Amendment 1 (1995)
Features:

- Inline functions
- New data types (long long int, complex, etc.)
- Variable length arrays
- Support for IEEE 754 floating point
- Single line comments using //

Onwards to C11…

Languages and Cars

C is a racing car that goes incredibly fast but breaks down every fifty miles.
Java is a family station wagon. It's easy to drive, it's not too fast, and you can't hurt yourself.
Perl is supposed to be a pretty cool car, but the driver's manual is incomprehensible. Also, even if you can figure out how to drive a Perl car, you won't be able to drive anyone else's.
Python is a great beginner's car; you can drive it without a license. Unless you want to drive really fast or on really treacherous terrain, you may never need another car.
Lisp: At first it doesn't seem to be a car at all, but now and then you spot a few people driving it around. After a point you decide to learn more about it and you realize it's actually a car that can make more cars. You tell your friends, but they all laugh and say these cars look way too weird. You still keep one in your garage, hoping one day they will take over the streets.

Properties of C

- Low-level
- Small
- Permissive
- Fast

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

Strengths of C

- Efficiency
- Portability
- Power
- Flexibility
- Standard library??
- Integration with UNIX

Weaknesses of C

- Programs can be error-prone.
- Programs can be difficult to understand.
    - International Obfuscated C Code Contest

```
#include<stdio.h>
int a = 256;int main(){for(char b[a+a+a],
*c=b ,*d=b+ a ,*e=b+a+a,*f,*g=fgets(e,(b[
a]=b [a+a] =a- a,a) , stdin);c[0]=a-a,f=c
,c=d ,d=e ,e=f, f= g,g =0,g = fgets(e,a+a
-a+ a -a+a -a+ a- +a,stdin ),f +a-a ; pu\
tchar(+10)) { for( int h= 1,i=1,j, k=0 ,l
=e[0]==32,m,n=0,o=c [ 0]== 32, p, q=0;d[q
];j=k,k=l,m=n,n=o,p=(j)+(k* 2 )+(l =(i =
e[ q]&&i ) &&e[q +1 ]== 32,l*4)+(m* 8 )+(
16*  n  )+(  o  =(h =c[ q]&&h)&&c[q+1]==
32,o* (16+16) )+0-0 +0, putchar(" ......"
/*\  (  |||  )  |/|/ / */".')|)\\\\\\\\'"
"" "|||"   "|||" "|'" ")|)\\\\\\\\'/|/(/"
"(/'/|/\\|\\|'/|/(/(/'/|/\\|\\|"[d[q++]==
32?p:0]));}}/* typographic tributaries */
```

- Programs can be difficult to modify.

Effective Use of C

- Learn how to avoid pitfalls.
- Use software tools to make programs more reliable.
- Take advantage of existing code libraries.
- Adopt a sensible set of coding conventions.
- Avoid “tricks” and overly complex code.
- Stick to the standard.
- Try and adapt the good habits from programming in Java!

First C Program: Hello, World!

```
/**************
 * Purpose: My first C Program, prints: Hello, World!
 * Author: gtowell
 * Created: Jan 29, 2021
 * Modified: Feb 1, 2021 by gtowell
 **************/

#include <stdio.h>

int main(void) {
  printf("Hello, World!.\n");
  return 0;
} // end of main()
```

- This program might be stored in a file named `hello`
- The file name doesn't matter
  - `.c` extension not required by Unix. Usually useful
    - used by gcc to indicate language

Compilation Process

-l “link” with the library c
-x compile with the language c

```
[xena@codewarrior cs246]$ gcc -lc -xc hello
```

What is a.out?
What is is the weird “./“
Why not have “./“ is front of gcc?

```
[xena@codewarrior cs246]$ ./a.out
Hello, World!
[xena@codewarrior cs246]$
```

Compilation Process – GNU C Compiler

```
Unix$ mv hello hello.c
Unix$ gcc hello.c
Unix$ ./a.out
Hello, World!
Unix$ gcc -o hello hello.c
```

The “-o xxx” specifies the name of the executable
Unlike Java, this is directly executable

what happens when you enter “java xxx”?

gcc interprets the “.c” file extension to mean “use -lc -xc”

```
Unix$ ./hello
Hello, World!
Unix$
```

Compilation Process

Compilation is a 3-step process

1. Preprocessing
   Source code commands that begin with a # are preprocessed. E.g.,

   ```
   #include <stdio.h>
   ```

2. Compiling
   Source code is translated into object code (m/c language)
   Single pass … function order matters!

3. Linking
   All libraries/modules used by the program are linked to produce an executable object code

Preprocessing is normally integrated into the compiler. Linking is done by a separate program/command.

C Program Structure (for now)

directives

```
int main(void) {
    statements
}
```

`#include <stdio.h>`

```
int main(void) {
  printf("Hello, World!.\n");
  return 0;
} // end of main()
```

- Before a C program is compiled, it is first edited by a preprocessor.
- Commands intended for the preprocessor are called directives.
- `<stdio.h>` is a ***header*** containing information about C's standard I/O library.

`main()`

- The `main()` function is mandatory.
- `main()` is special: it gets called automatically when the program is executed.
- `main` returns a status code; the value 0 indicates normal program termination.
- If there's no `return` statement at the end of the `main` function, many compilers will produce a warning message.

Comments – Two styles /*…*/ or //

- Like Java

Another Program (variables, assignment, formatted output)

```
File: small.c
#include <stdio.h>

int main(void) {

    int a, b, c;

    a = 24;
    b = 18;
    c = a + b;


    printf("c = %d\n", c);
} // main()

[xena@codewarrior cs246]$ gcc -o small small.c
[xena@codewarrior cs246]$ ./small
c = 42
[xena@codewarrior cs246]$
```

Java String.format

Unlike java "+" does NOT concatenate so `printf("c= " + c + "\n");` does not work

Printing Strings

- The statement

```
printf("To C, or not to C: that is the question.\n”);
```

kin to Java `System.out.print()`

could be replaced by two calls of `printf`:

```
printf("To C, or not to C: ");
printf("that is the question.\n");
```

- The new-line character can appear more than once in a string literal:

```
printf("Brevity is the soul of wit.\n  --Shakespeare\n");
```

Printing the Value of a Variable

- `%d` works only for `int` variables; use %f to print a `float` variable
- By default, `%f` displays a number with six digits after the decimal point.
- To force `%f` to display *p* digits after the decimal point, put `.`*p* between `%` and `f`.
- To print the line

  ```
  Profit: $2150.48
  ```

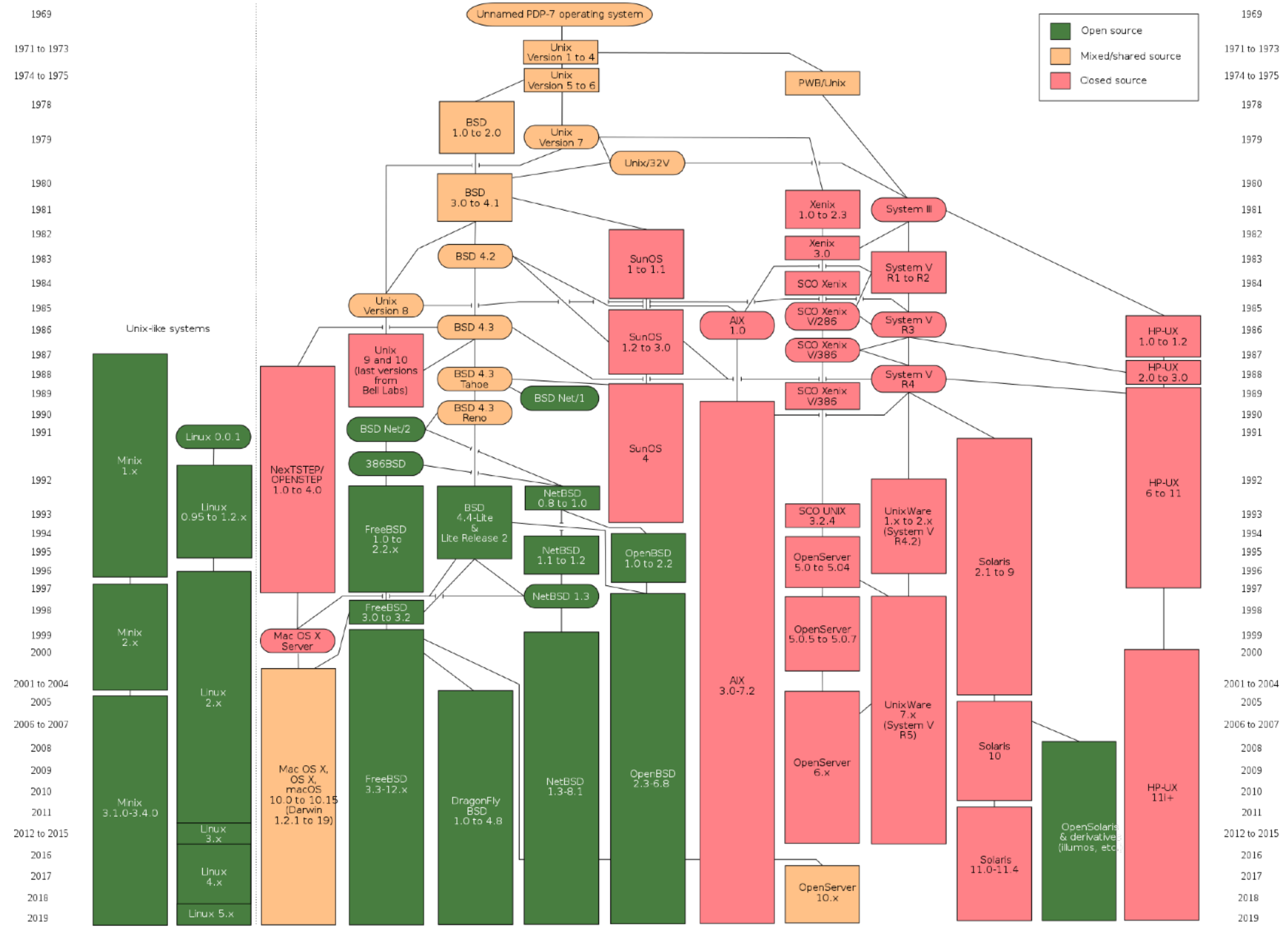
  use the following call of `printf`:

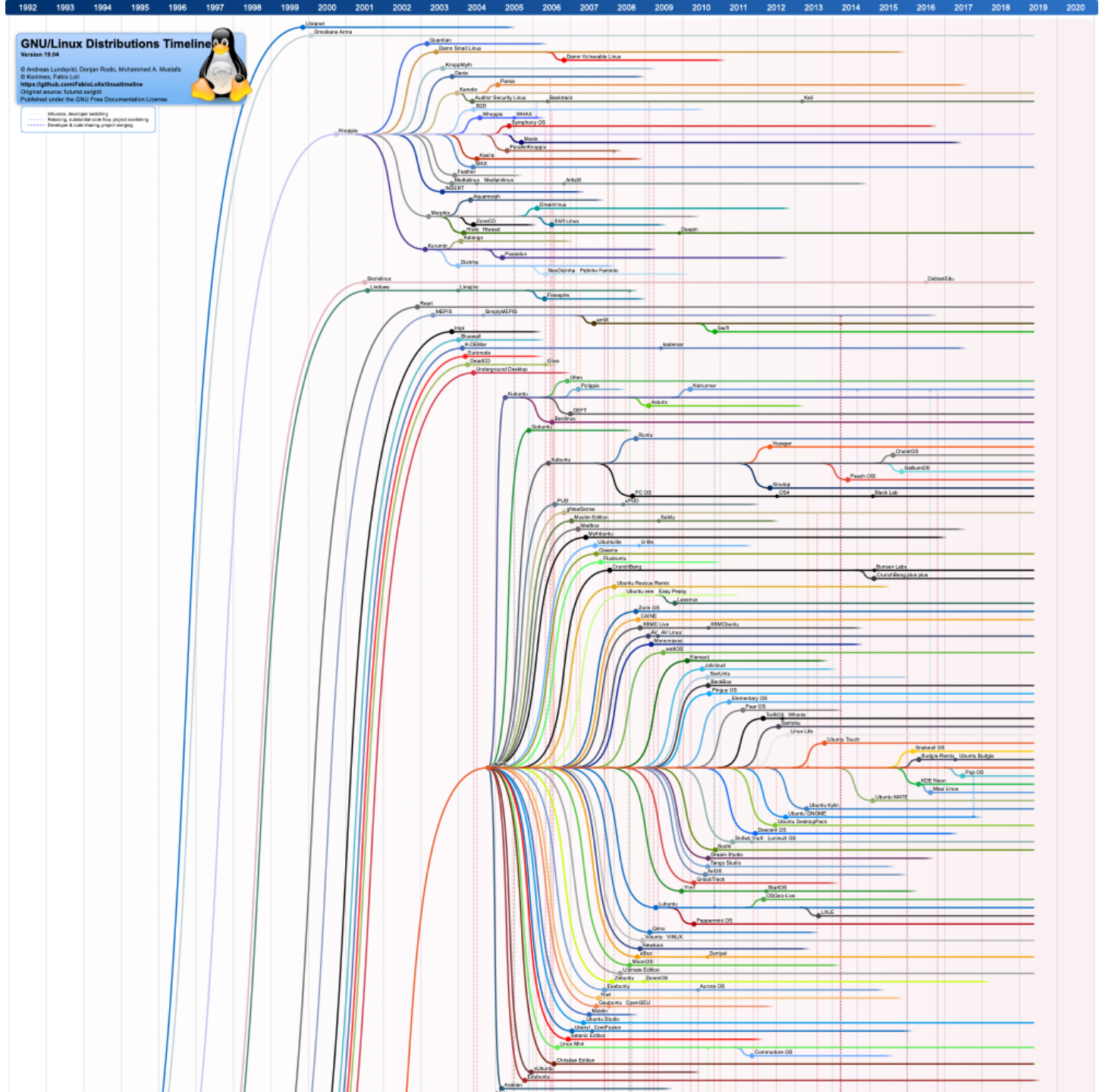
  ```
  printf("Profit: $%.2f\n", profit);
  ```
- There's no limit to the number of variables that can be printed by a single call of `printf`:

  ```
  printf("Height: %d  Length: %d\n", height, length);
  ```

Unix Time!!!

- cd
  - absolute path
  - relative path
  - ~
- ls
  - flags: -l -a -r { -t -S }
- pwd
- 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!


Open source
Mixed/shared source
Closed source
Unix-like systems
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2015
2016
2017
2018
2019
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
Unix Version 7
PWB/Unix
BSD 1.0 to 2.0
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
BSD 4.2
SunOS 1 to 1.1
System V R1 to R2
SCO Xenix
Unix Version 8
SCO Xenix V/286
System V R3
BSD 4.3
SunOS 1.2 to 3.0
AIX 1.0
HP-UX 1.0 to 1.2
SCO Xenix V/386
HP-UX 2.0 to 3.0
Unix 9 and 10 (last versions from Bell Labs)
BSD 4.3 Tahoe
System V R4
BSD Net/1
SCO Xenix V/386
BSD 4.3 Reno
HP-UX 6 to 11
Minix 1.x
Linux 0.0.1
BSD Net/2
NeXTSTEP/ OPENSTEP 1.0 to 4.0
SunOS 4
386BSD
Linux 0.95 to 1.2.x
NetBSD 0.8 to 1.0
SCO UNIX 3.2.4
UnixWare 1.x to 2.x (System V R4.2)
Solaris 2.1 to 9
FreeBSD 1.0 to 2.2.x
BSD 4.4-Lite & Lite Release 2
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
Minix 2.x
NetBSD 1.3
OpenServer 5.0.5 to 5.0.7
FreeBSD 3.0 to 3.2
Linux 2.x
Mac OS X Server
AIX 3.0-7.2
UnixWare 7.x (System V R5)
Solaris 10
Mac OS X, OS X, macOS 10.0 to 10.15 (Darwin 1.2.1 to 19)
FreeBSD 3.3-12.x
DragonFly BSD 1.0 to 4.8
NetBSD 1.3-8.1
OpenBSD 2.3-6.8
OpenServer 6.x
Solaris 11.0-11.4
OpenSolaris & derivatives (illumos, etc)
HP-UX 11i+
Minix 3.1.0-3.4.0
Linux 3.x
Linux 4.x
Linux 5.x
OpenServer 10.x


GNU/Linux Distributions Timeline
Version 19.04
© Andreas Lundqvist, Donjan Rodic, Mohammed A. Mustafa
© Konimex, Fabio Loli
https://github.com/FabioLoliLinuxtimeline
Original source: futurist.se/gldt
Published under the GNU Free Documentation License
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020

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 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()
```

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.

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
```

- Many programmers use only lower-case letters in identifiers (other than macros), with underscores inserted for legibility:

```
symbol_table  current_page  name_and_address
```

- Other programmers use an upper-case letter to begin each word within an identifier:

```
symbolTable  currentPage  nameAndAddress
```

- 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*
  `||` 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:

  ```
  (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.

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++)
  …
```

Acknowledgements

Some content from these slides is based on the book, C Programming – A Modern Approach, By K. N. King, 2nd Edition. W. W. Norton 2008.

Materials are also included from the lecture slides p
Prof. K. N. King. Thank You!

STOP!!

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 ***/
```

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*`.`*p*`X` or `%-`*m*`.`*p*`X`, 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.

Conversion Specifications

- Conversion specifiers for floating-point numbers:

  `e` — Exponential format. *p* indicates how many digits should appear after the decimal point (the default is 6). If *p* is 0, no decimal point is displayed.

  `f` — "Fixed decimal" format. *p* has the same meaning as for the `e` specifier.

  `g` — Either exponential format or fixed decimal format, depending on the number's size. *p* indicates the maximum number of significant digits to be displayed. The `g` conversion won't show trailing zeros. If the number has no digits after the decimal point, `g` doesn't display the decimal point.

Program: Using **`printf`** to Format Numbers

- The `tprintf.c` program uses `printf` to display integers and floating-point numbers in various formats.

tprintf.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    |
```

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
```

Escape Sequences

- 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 */
```

The **`scanf`** Function

- `scanf` reads input according to a particular format.
- A `scanf` format string may contain both ordinary characters and conversion specifications.
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.

The **`scanf`** Function

- In many cases, a `scanf` format string will contain only conversion specifications:

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

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

- Sample input:

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

`scanf` will assign 1, -20, 0.3, and -4000.0 to `i`, `j`, `x`, and `y`, respectively.

The **`scanf`** Function

- When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable.
- Another trap involves the `&` symbol, which normally precedes each variable in a `scanf` call.
- The `&` is usually (but not always) required, and it's the programmer's responsibility to remember to use it.

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;
}
```