

# CS631 - Advanced Programming in the UNIX Environment

—

## UNIX development tools

---

Department of Computer Science  
Stevens Institute of Technology  
Jan Schaumann

`jschauma@stevens.edu`

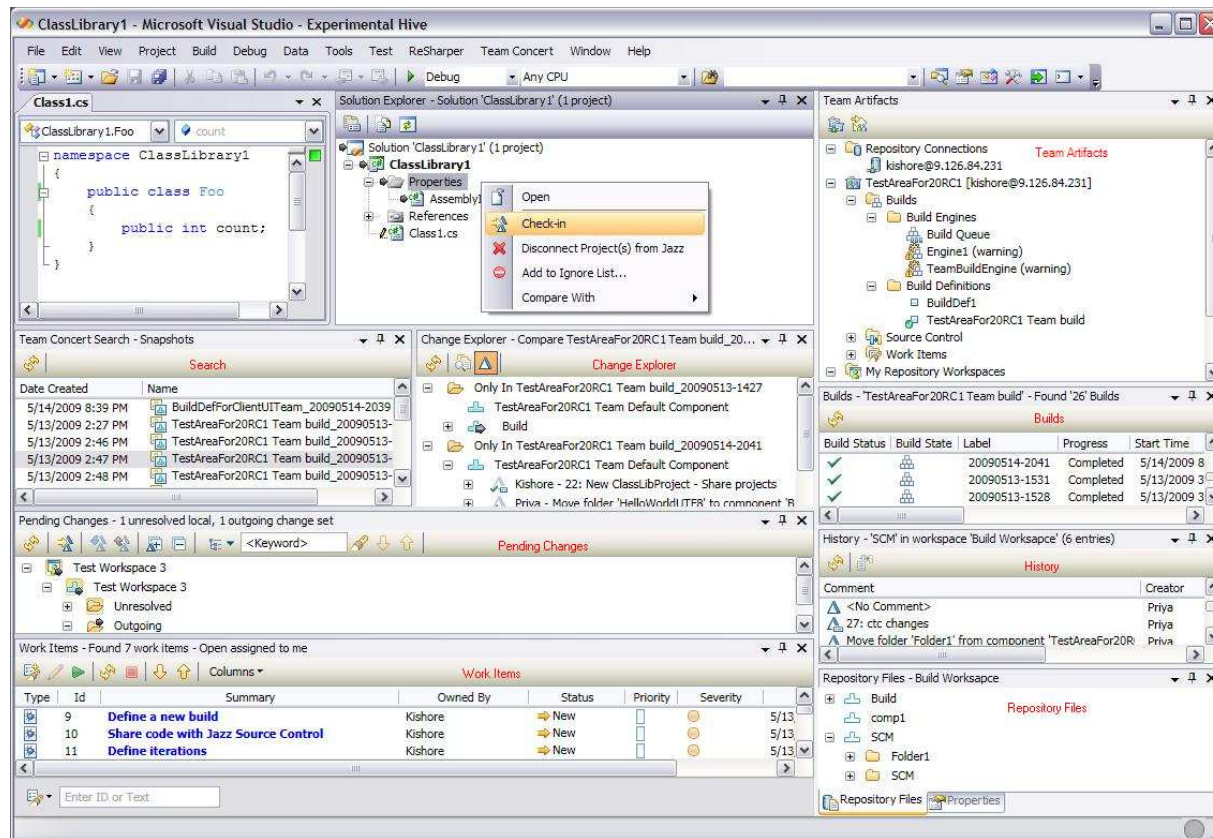
`https://stevens.netmeister.org/631/`

## HW1 revisited

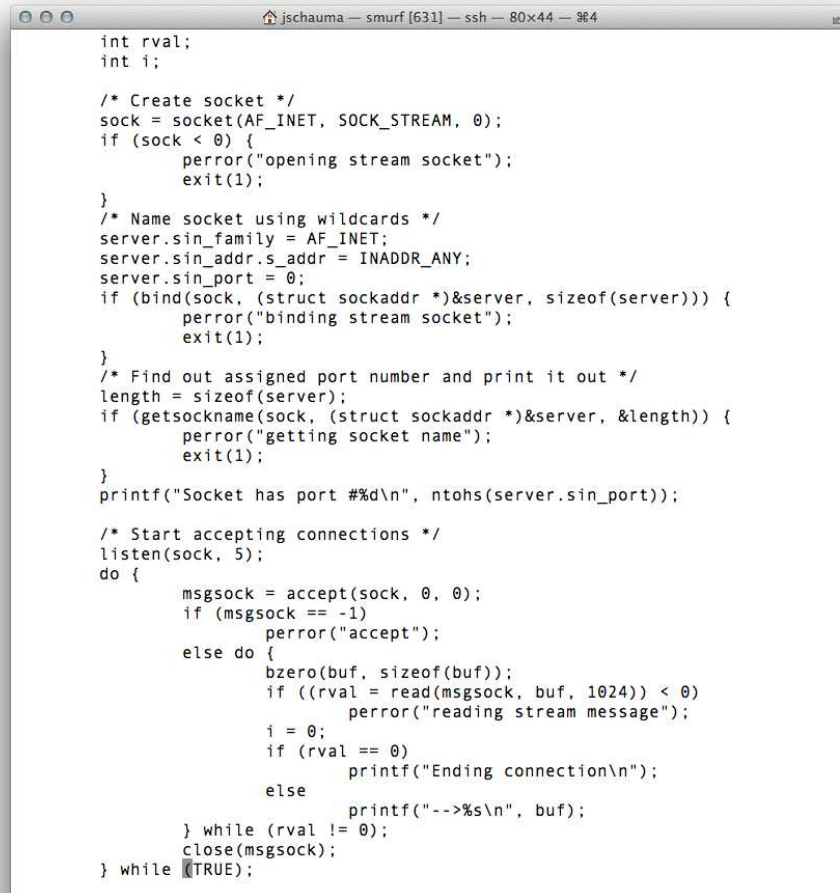
---

- know your pre-defined constants (e.g., BUFSIZ, PATH\_MAX, ...); see `limits.h` / `limit(3)`
- avoid code duplication in multiple blocks
- don't `strcat(3)` to `argv[]`
- the checklist is not there just to annoy you

# Software Development Tools



# Software Development Tools



```
int rval;
int i;

/* Create socket */
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    perror("opening stream socket");
    exit(1);
}

/* Name socket using wildcards */
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind(sock, (struct sockaddr *)&server, sizeof(server))) {
    perror("binding stream socket");
    exit(1);
}

/* Find out assigned port number and print it out */
length = sizeof(server);
if (getsockname(sock, (struct sockaddr *)&server, &length)) {
    perror("getting socket name");
    exit(1);
}
printf("Socket has port %d\n", ntohs(server.sin_port));

/* Start accepting connections */
listen(sock, 5);
do {
    msgsock = accept(sock, 0, 0);
    if (msgsock == -1)
        perror("accept");
    else do {
        bzero(buf, sizeof(buf));
        if ((rval = read(msgsock, buf, 1024)) < 0)
            perror("reading stream message");
        i = 0;
        if (rval == 0)
            printf("Ending connection\n");
        else
            printf("-->%s\n", buf);
    } while (rval != 0);
    close(msgsock);
} while (TRUE);
```

## Software Development Tools

---

UNIX Userland is an IDE – essential tools that follow the paradigm of “Do one thing, and do it right” can be combined.

The most important tools are:

- `$EDITOR`
- the compiler toolchain
- `gdb(1)` – debugging your code
- `make(1)` – project build management, maintain program dependencies
- `diff(1)` and `patch(1)` – report and apply differences between files
- `cvs(1)`, `svn(1)`, `git(1)` etc. – distributed project management, version control

# EDITOR

---

Know your \$EDITOR. Core functionality:

- syntax highlighting
- efficient keyboard maneuvering
- setting markers, using buffers
- copy, yank, fold e.g. blocks
- search and replace
- window splitting
- autocompletion
- jump to definition / manual page
- applying external commands and filters

# EDITOR

---

Examples given using `vim(1)`.

Efficient keyboard maneuvering:

- up, down, left, right (h, j, k, l)
- move by word, go to end (w, b, e)
- search forward, backward, move to beginning or end of line (/, /, ?, ^, \$)
- page up or down (^D, ^B)
- center page, top or bottom (zz, zt, zb)
- move to matching brace, move to beginning/end of code block (% , ] }, [ {)
- move through multiple files (:n, :prev, :rew)

# EDITOR

---

Examples given using `vim(1)`.

Copy, yank, fold, markers, buffers etc.:

- set and display markers (m [a-zA-Z], :marks)
- select visual blocks (v, V)
- format / indent selected block (=)
- delete, yank, use of buffers (d, y, "xy, "xp)
- fold sections (zf, zA)



# EDITOR

---

Examples given using `vim(1)`.

Look-ups:



```
find /usr/src -name '*[ch]' -print | xargs ctags -f ~/.ctags
```



```
echo "set tags+=~/.ctags" >> ~/.vimrc
```



Ctrl+], Ctrl+t – jump to definition and back



K – jump to manual page



Ctrl+N – autocomplete

## EDITOR

---

Examples given using `vim(1)`.

Integration with compiler, debugger, `make(1)` etc.

```
vim welcome.c
:make
Ctrl+]
:cnext
...
```

Finally, two of your most powerful Unix IDE integrations are a terminal multiplexer (e.g. `screen(1)` or `tmux(1)`) and copious use of `Ctrl+Z` (i.e., the shell's job control mechanisms).

# EDITOR

Examples given using `vim(1)`.

version 1.1  
April 1st, 06

## vi / vim graphical cheat sheet

<b>Esc</b> normal mode	<b>~</b> toggle case	<b>!</b> external filter	<b>@</b> play macro	<b>#</b> prev ident	<b>\$</b> eol	<b>%</b> goto match	<b>^</b> "soft" bol	<b>&amp;</b> repeat :s	<b>*</b> next ident	<b>(</b> begin sentence	<b>)</b> end sentence	<b>"</b> "soft" bol	<b>+</b> next line
<b>.</b> goto mark	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>0</b> "hard" bol	<b>-</b> prev line	<b>=</b> autoformat	
<b>Q</b> ex mode	<b>W</b> next WORD	<b>E</b> end WORD	<b>R</b> replace mode	<b>T</b> back 'till	<b>Y</b> yank line	<b>U</b> undo line	<b>I</b> insert at bol	<b>O</b> open above	<b>P</b> paste before	<b>{</b> begin parag.	<b>}</b> end parag.		
<b>q</b> record macro	<b>w</b> next word	<b>e</b> end word	<b>r</b> replace char	<b>t</b> 'till	<b>y</b> yank	<b>u</b> undo	<b>i</b> insert mode	<b>o</b> open below	<b>p</b> paste after	<b>[</b> misc	<b>]</b> misc		
<b>A</b> append at eol	<b>S</b> subst line	<b>D</b> delete to eol	<b>F</b> "back" find ch	<b>G</b> eof/goto ln	<b>H</b> screen top	<b>J</b> join lines	<b>K</b> help	<b>L</b> screen bottom	<b>.</b> ex cmd line	<b>"</b> reg. spec	<b>'</b> goto mk. bol	<b> </b> bol/goto col	
<b>a</b> append	<b>s</b> subst char	<b>d</b> delete	<b>f</b> find char	<b>g</b> extra cmds	<b>h</b> ←	<b>j</b> ↓	<b>k</b> ↑	<b>l</b> →	<b>.</b> repeat t/T/t/F	<b>'</b> goto mk. bol	<b> </b> not used!		
<b>Z</b> quit	<b>X</b> back-space	<b>C</b> change to eol	<b>V</b> visual lines	<b>B</b> prev WORD	<b>N</b> prev (find)	<b>M</b> screen mid	<b>&lt;</b> un-indent	<b>&gt;</b> indent	<b>?</b> find (rev.)				
<b>Z</b> extra cmds	<b>x</b> delete char	<b>c</b> change	<b>v</b> visual mode	<b>b</b> prev word	<b>n</b> next (find)	<b>m</b> set mark	<b>~</b> reverse t/T/t/F	<b>.</b> repeat cmd	<b>/</b> find				

**motion** moves the cursor, or defines the range for an operator  
**command** direct action command, if **red**, it enters insert mode  
**operator** requires a motion afterwards, operates between cursor & destination  
**extra** special functions, requires extra input  
**Q** commands with a dot need a char argument afterwards  
 bol = beginning of line, eol = end of line,  
 mk = mark, yank = copy  
 words: `quux(foo, bar, baz)`  
 WORDS: `quux(foo, bar, baz)`

**Main command line commands ('ex'):**  
 :w (save), :q (quit), :q! (quit w/o saving)  
 :e f (open file f),  
 :%s/x/y/g (replace 'x' by 'y' filewide),  
 :h (help in vim), :new (new file in vim),

**Other important commands:**  
 CTRL-R: redo (vim),  
 CTRL-F/-B: page up/down,  
 CTRL-E/-Y: scroll line up/down,  
 CTRL-V: block-visual mode (vim only)

**Visual mode:**  
 Move around and type operator to act on selected region (vim only)

**Notes:**  
 (1) use "x before a yank/paste/del command to use that register ('clipboard') (x=a..z,\*) (e.g.: "ay\$ to copy rest of line to reg 'a')  
 (2) type in a number before any action to repeat it that number of times (e.g.: 2p, d2w, 5i, d4j)  
 (3) duplicate operator to act on current line (dd = delete line, >> = indent line)  
 (4) ZZ to save & quit, ZQ to quit w/o saving  
 (5) zt: scroll cursor to top, zb: bottom, zz: center  
 (6) gg: top of file (vim only), gf: open file under cursor (vim only)

For a graphical vi/vim tutorial & more tips, go to [www.viemu.com](http://www.viemu.com) - home of ViEmu, vi/vim emulation for Microsoft Visual Studio

<https://duckduckgo.com/?q=vim+tutorial>

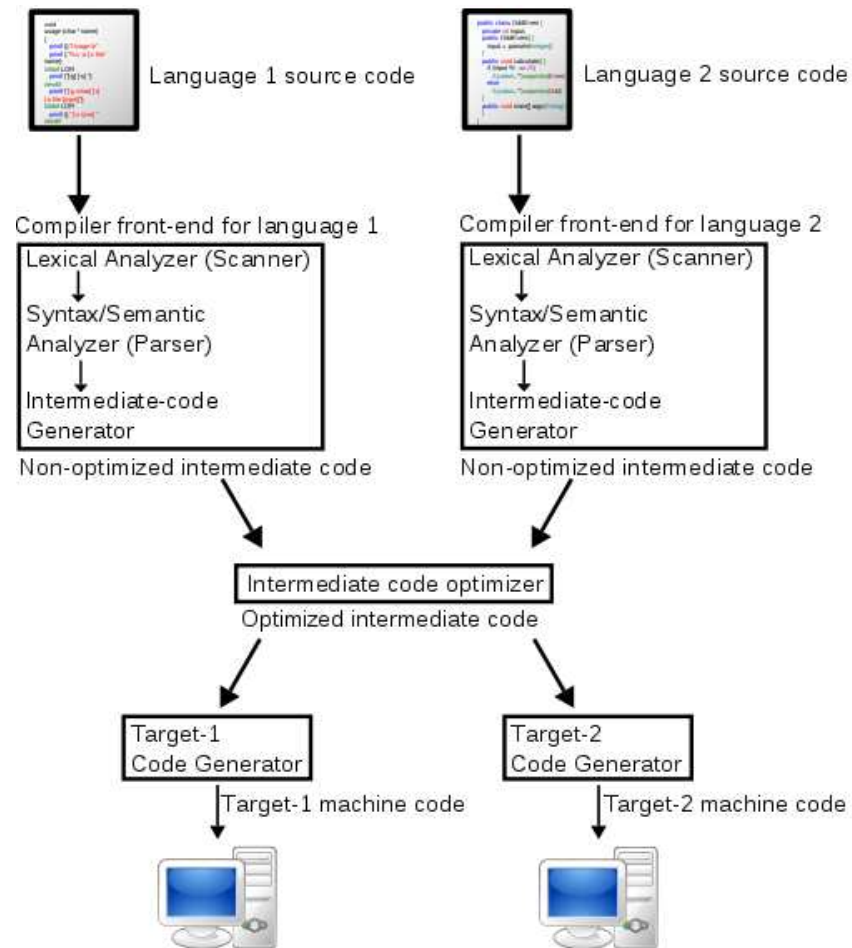
## Compilers

---

A compiler translates *source code* from a high-level programming language into *machine code* for a given architecture by performing a number of steps:

- lexical analysis
- preprocessing
- parsing
- semantic analysis
- code optimization
- code generation
- assembly
- linking

# Compilers



## Compilers

---

There are many different closed- and open-source compiler chains:

- Intel C/C++ Compiler (or `icc`)
- Turbo C / Turbo C++ / C++Builder (Borland)
- Microsoft Visual C++
- ...
  
- Clang (a frontend to LLVM)
- GNU Compiler Collection (or `gcc`)
- Portable C Compiler (or `pcc`)
- ...

## The compiler toolchain

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

## Preprocessing

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ cd compilechain
$ cat hello.c
$ man cpp
$ cpp hello.c hello.i
$ file hello.i
$ man cc
$ cc -v -E hello.c > hello.i
$ more hello.i
$ cc -v -DFOOD=\"Avocado\" -E hello.c > hello.i.2
$ diff -bu hello.i hello.i.2
```



## Compilation

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ more hello.i
$ cc -v -S hello.i
$ file hello.s
$ more hello.s
```

## Assembly

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ as -o hello.o hello.s
$ file hello.o
$ cc -v -c hello.s
$ objdump -d hello.o
[...]
```

## Linking

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ ld hello.o
[...]
```

The first command shows the linker taking a single object file. The second command shows it taking a single library. The third command shows the compiler driver taking a single object file. The fourth command shows the linker taking a single object file and a single library. The fifth command shows the linker taking a single object file and a single library. The sixth command shows the linker taking a single object file and a single library.

```
$ ld -dynamic-linker /usr/libexec/ld.elf_so \
    /usr/lib/crt0.o /usr/lib/crti.o /usr/lib/crtbegin.o \
    hello.o -lc /usr/lib/crtend.o /usr/lib/crtn.o
```

```
$ file a.out
$ ./a.out
```

## Linking

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

```
$ cc -v -DFOOD=\"Avocado\" hello.c 2>&1 | more
```

## `cc(1)` and `ld(1)`

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

```
$ cc -v -O2 -g hello.c 2>&1 | more
```

## `cc(1)` and `ld(1)`

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via `-L` and the resolving of undefined symbols via `-l` are examples of position sensitive flags.

```
$ cc -v main.c -L./lib2 -L./lib -lldtest 2>&1 | more
```

```
$ cc -v main.c -L./lib -L./lib2 -lldtest 2>&1 | more
```

## `cc(1)` and `ld(1)`

---

The compiler chain or driver usually performs preprocessing (e.g. via `cpp(1)`), compilation (`cc(1)`), assembly (`as(1)`) and linking (`ld(1)`).

Different flags can be passed to `cc(1)` to be passed through to each tool as well as to affect all tools.

The order of the command line flags *may* play a role! Directories searched for libraries via `-L` and the resolving of undefined symbols via `-l` are examples of position sensitive flags.

The behavior of the compiler toolchain may be influenced by environment variables (eg `TMPDIR`, `SGI_ABI`) and/or the compilers default configuration file (MIPSPro's `/etc/compiler.defaults` or gcc's `specs`).

```
$ cc -v hello.c
```

```
$ TMPDIR=/var/tmp cc -v hello.c
```

```
$ cc -dumpspec
```

## A Debugger

---





## `gdb(1)`

---

The purpose of a debugger such as `gdb(1)` is to allow you to see what is going on “inside” another program while it executes – or what another program was doing at the moment it crashed. `gdb` allows you to

- make your program stop on specified conditions (for example by setting *breakpoints*)
- examine what has happened, when your program has stopped (by looking at the *backtrace*, inspecting the value of certain variables)
- inspect control flow (for example by *stepping* through the program)

Other interesting things you can do:

- examine stack frames: *info frame*, *info locals*, *info args*
- examine memory: *x*
- examine assembly: *disassemble func*

## `gdb(1)`

---

```
$ cc simple-ls.c
```

```
$ ./a.out ~/testdir
```

```
Memory fault (core dumped)
```

```
$ gdb ./a.out
```

```
(gdb) run ~/testdir
```

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x0000000000400cc7 in main (argc=2, argv=0x7f7fffa71978) at simple-ls-stat.c:48
```

```
warning: Source file is more recent than executable.
```

```
48             printf("%s (%s)\n", dirp->d_name, pwd->pw_name);
```

```
(gdb) bt
```

```
(gdb) frame 0
```

```
(gdb) li
```

```
(gdb) print pwd
```

## `gdb(1)`

---

```
$ cc gdb2.c
$ gdb ./a.out
(gdb) run
[...]
(gdb) show environment BUFSIZ
[...]
(gdb) p num
[...]
```

make(1)

---



## make(1)

---

make(1) is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files

## make(1)

---

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file

## make(1)

---

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*

## make(1)

---

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined



## make(1)

---

`make(1)` is a command generator and build utility. Using a description file (usually *Makefile*) it creates a sequence of commands for execution by the shell.

- used to sort out dependency relations among files
- avoids having to rebuild the entire project after modification of a single source file
- performs *selective* rebuilds following a *dependency graph*
- allows simplification of rules through use of *macros* and *suffixes*, some of which are internally defined
- different versions of `make(1)` (BSD make, GNU make, Sys V make, ...) may differ (among other things) in
  - variable assignment and expansion/substitution
  - including other files
  - flow control (for-loops, conditionals etc.)

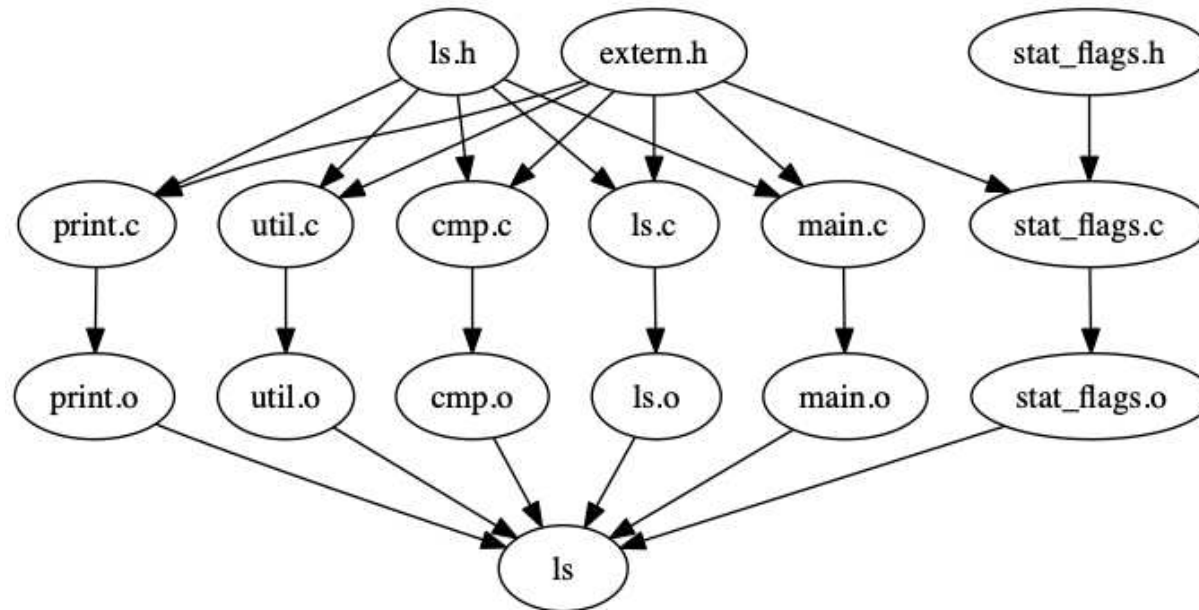
## make(1)

---

```
$ cd make-examples
```

```
$ ls *.*[ch]
```

cmp.c	ls.c	main.c	stat_flags.c	util.c
extern.h	ls.h	print.c	stat_flags.h	



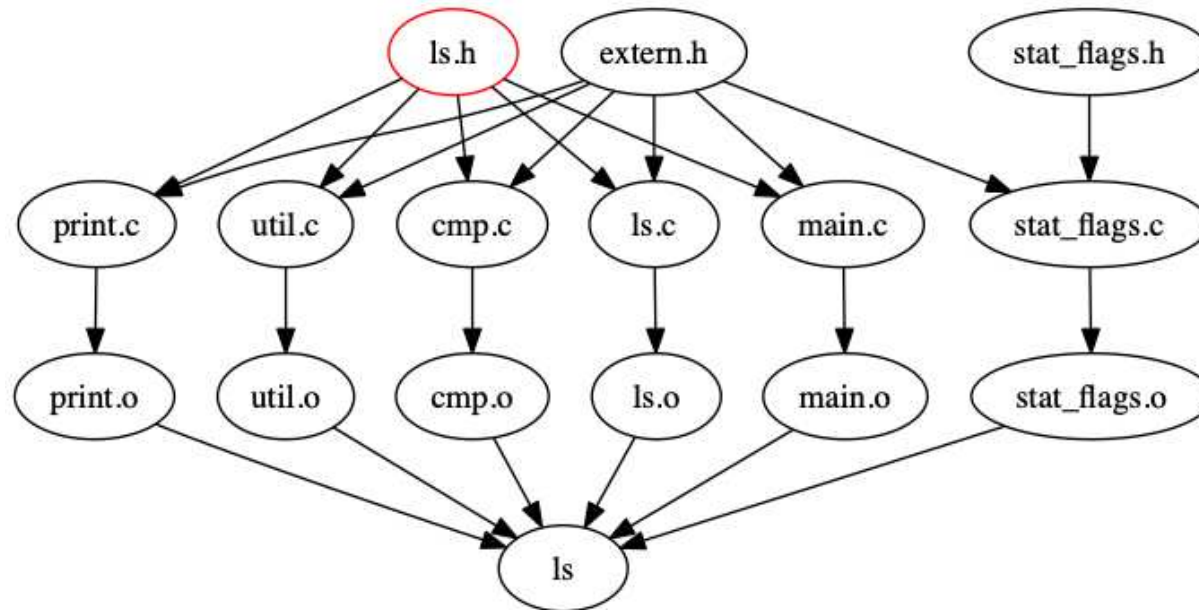
## make(1)

---

```
$ cd make-examples
```

```
$ ls *. [ch]
```

cmp.c	ls.c	main.c	stat_flags.c	util.c
extern.h	ls.h	print.c	stat_flags.h	



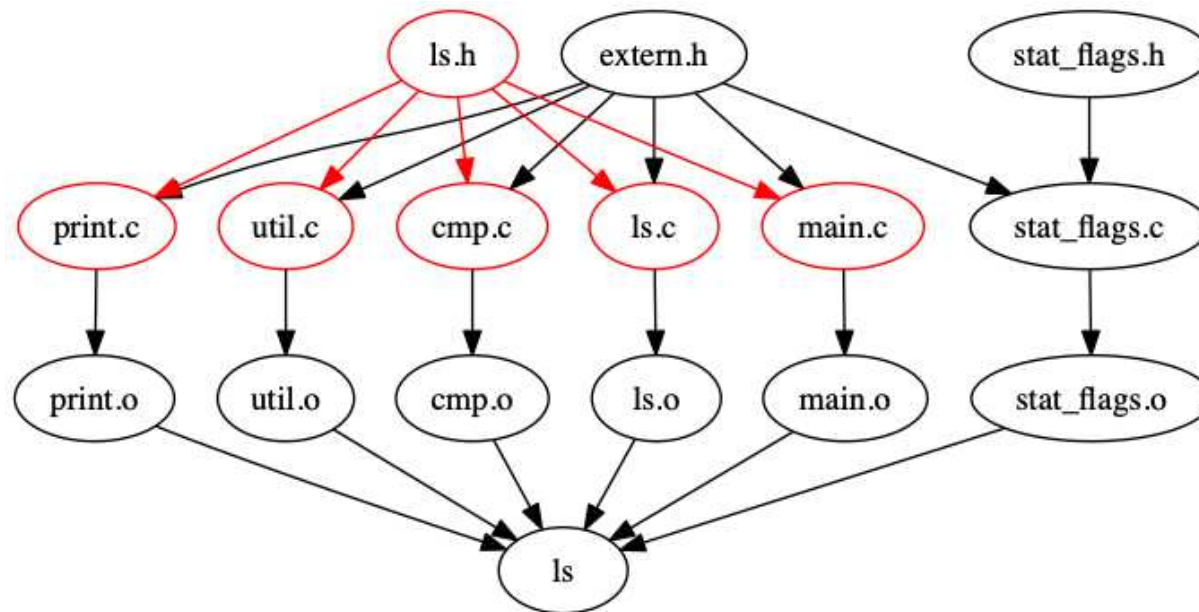
## make(1)

---

```
$ cd make-examples
```

```
$ ls *.[ch]
```

cmp.c	ls.c	main.c	stat_flags.c	util.c
extern.h	ls.h	print.c	stat_flags.h	



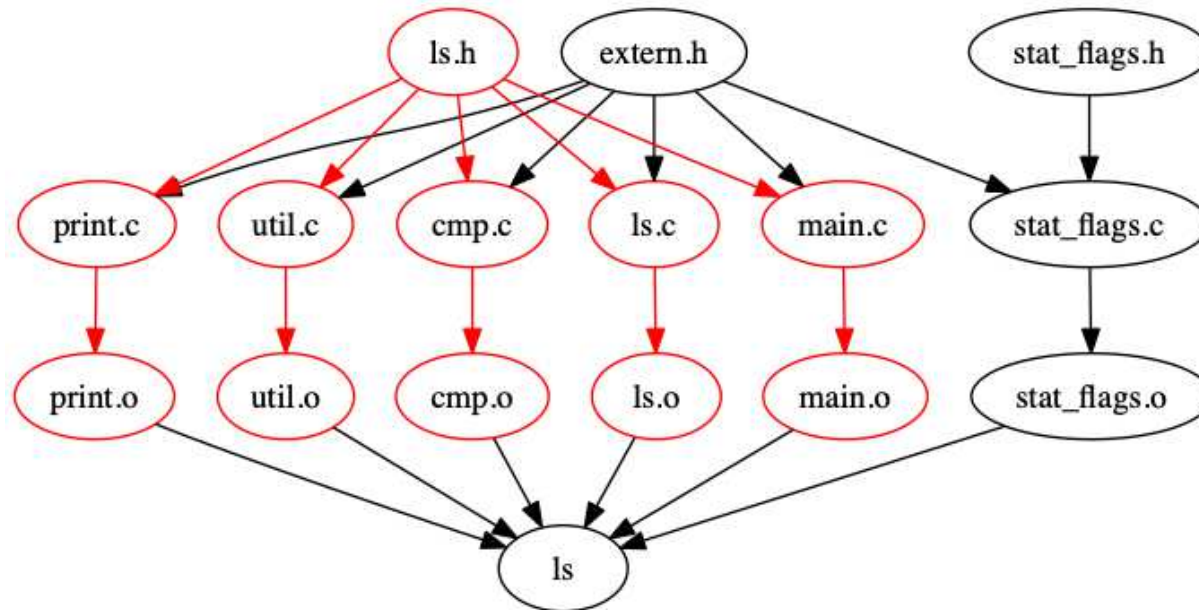
## make(1)

---

```
$ cd make-examples
```

```
$ ls *.[ch]
```

cmp.c	ls.c	main.c	stat_flags.c	util.c
extern.h	ls.h	print.c	stat_flags.h	



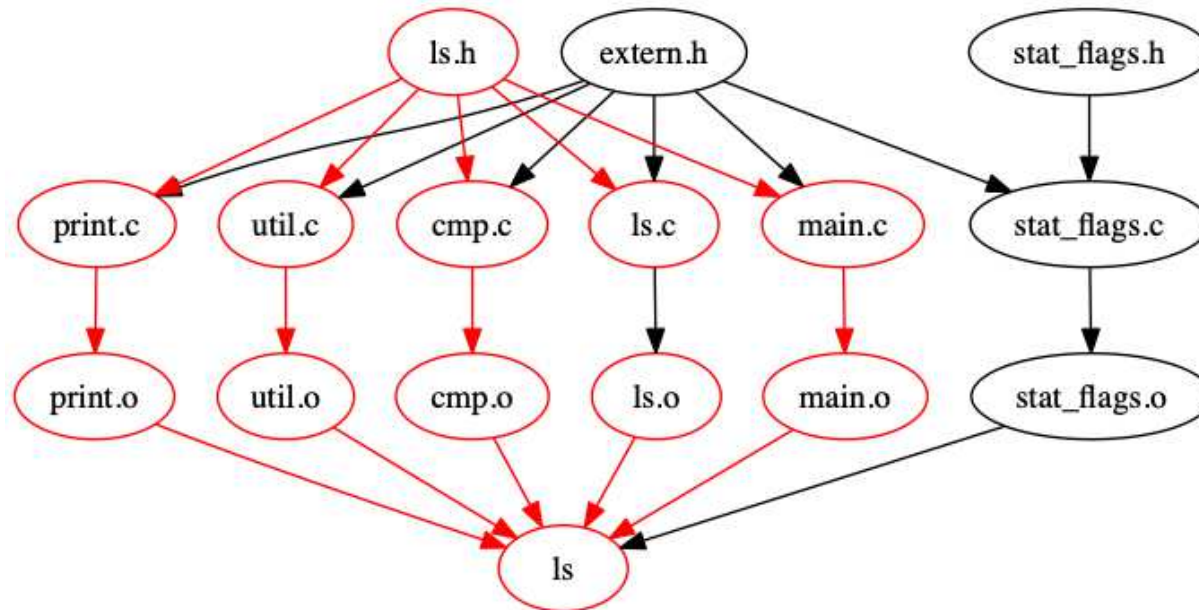
## make(1)

---

```
$ cd make-examples
```

```
$ ls *.[ch]
```

cmp.c	ls.c	main.c	stat_flags.c	util.c
extern.h	ls.h	print.c	stat_flags.h	



## make(1)

---

```
$ ln -s Makefile.1 Makefile
$ make # or: make -f Makefile.1
[...]
$ make
[...]
$ make clean
$ export CFLAGS="-Wall -Werror"
$ make
[...]
$ make clean
[...]
$ make showvars
[...]
$ make CFLAGS="${CFLAGS}" showvars
[...]
```

Repeat with other Makefiles.

## Priority of Macro Assignments for `make(1)`

---

1. Internal (default) definitions of `make(1)`
2. Current shell environment variables. This includes macros that you enter on the *make* command line itself.
3. Macro definitions in *Makefile*.
4. Macros entered on the `make(1)` command line, if they follow the *make* command itself.



## Ed is the standard text editor.

---

```
$ ed
?
help
?
quit
?
exit
?
bye
?
eat flaming death
?
^C
?
^D
?
```

## Ed is the standard text editor.

---

```
$ ed
a
ed is the standard Unix text editor.
This is line number two.
.
2i

.
%1
3s/two/three/
w foo
q
$ cat foo
```

## diff(1) and patch(1)

---

diff(1):

- compares files line by line
- output may be used to automatically edit a file
- can produce human “readable” output as well as diff entire directory structures
- output called a *patch*

## diff(1) and patch(1)

---

patch(1):

- applies a `diff(1)` file (aka *patch*) to an original
- may back up original file
- may guess correct format
- ignores leading or trailing “garbage”
- allows for reversing the patch
- may even correct context line numbers

## diff(1) and patch(1)

---

```
$ diff Makefile.2 Makefile.5
[...]
$ cp Makefile.2 /tmp
$ ( diff -e Makefile.2 Makefile.5; echo w; ) | ed Makefile.2
$ diff Makefile.[25]
$ mv /tmp/Makefile.2 .
$ diff -c Makefile.[25]
$ diff -u Makefile.[25] > /tmp/patch
$ patch </tmp/patch
$ diff Makefile.[25]
```

## diff(1) and patch(1)

---

Difference in `ls(1)` between NetBSD and OpenBSD:

```
$ diff -bur netbsd/src/bin/ls openbsd/src/bin/ls
```

Difference in `ls(1)` between NetBSD and FreeBSD:

```
$ diff -bur netbsd/src/bin/ls freebsd-1s/ls
```

## Revision Control

---

To be continued...

## Links

---

GDB:

<https://sourceware.org/gdb/current/onlinedocs/gdb/>

<http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html>

<http://www.unknownroad.com/rtfm/gdbtut/gdbtoc.html>