



# Building Multi-File Programs with the `make` Tool



# Agenda



## **Motivation for Make**

Make Fundamentals

Non-File Targets

Macros



# Multi-File Programs

## intmath.h (interface)

```
#ifndef INTMATH_INCLUDED
#define INTMATH_INCLUDED
int gcd(int i, int j);
int lcm(int i, int j);
#endif
```

## intmath.c (implementation)

```
#include "intmath.h"

int gcd(int i, int j)
{
    int temp;
    while (j != 0)
    {
        temp = i % j;
        i = j;
        j = temp;
    }
    return i;
}

int lcm(int i, int j)
{
    return (i / gcd(i, j)) * j;
}
```

## testintmath.c (client)

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

int main(void)
{
    int i;
    int j;
    printf("Enter the first integer:\n");
    scanf("%d", &i);
    printf("Enter the second integer:\n");
    scanf("%d", &j);
    printf("Greatest common divisor: %d.\n",
        gcd(i, j));
    printf("Least common multiple: %d.\n",
        lcm(i, j));
    return 0;
}
```

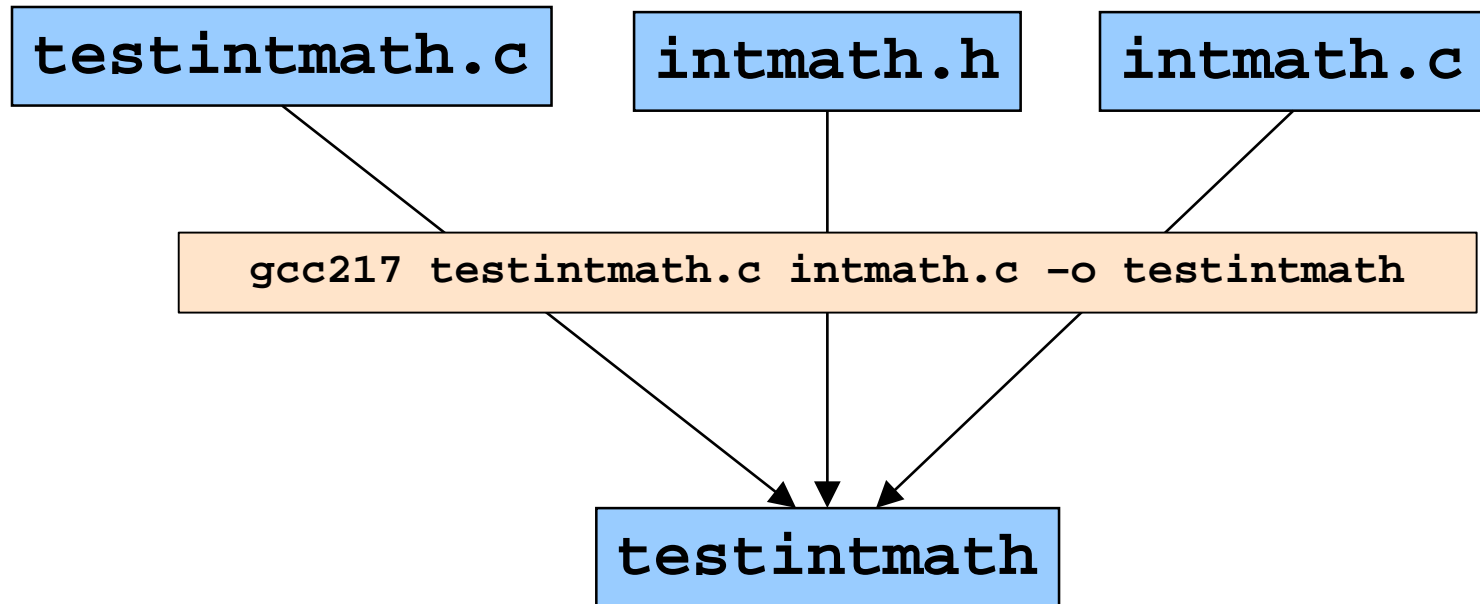
Note: intmath.h is  
#included into intmath.c  
and testintmath.c



# Motivation for Make (Part 1)

## Building `testintmath`, approach 1:

- Use one `gcc217` command to preprocess, compile, assemble, and link

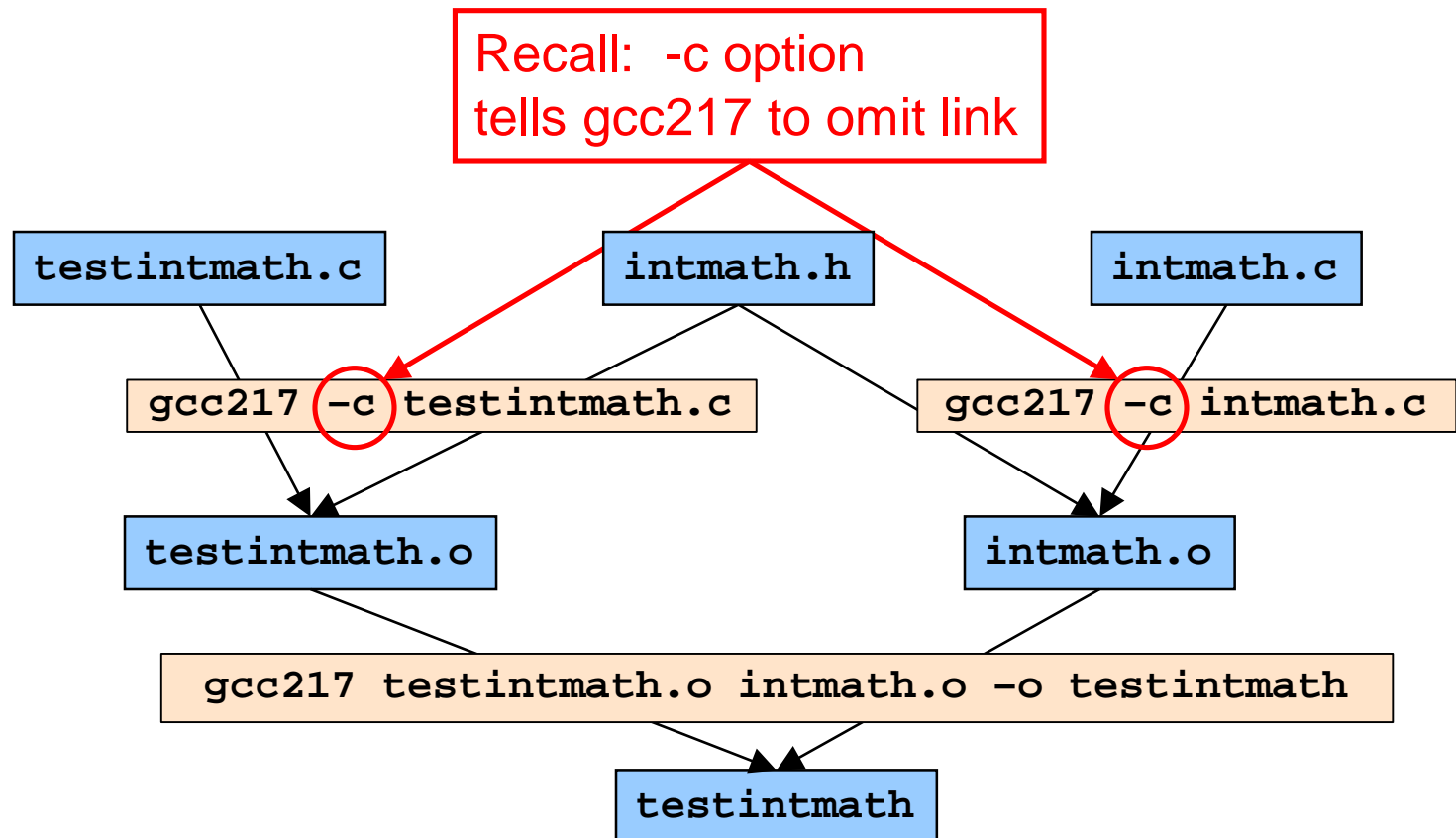




# Motivation for Make (Part 2)

## Building `testintmath`, approach 2:

- Preprocess, compile, assemble to produce `.o` files
- Link to produce executable binary file



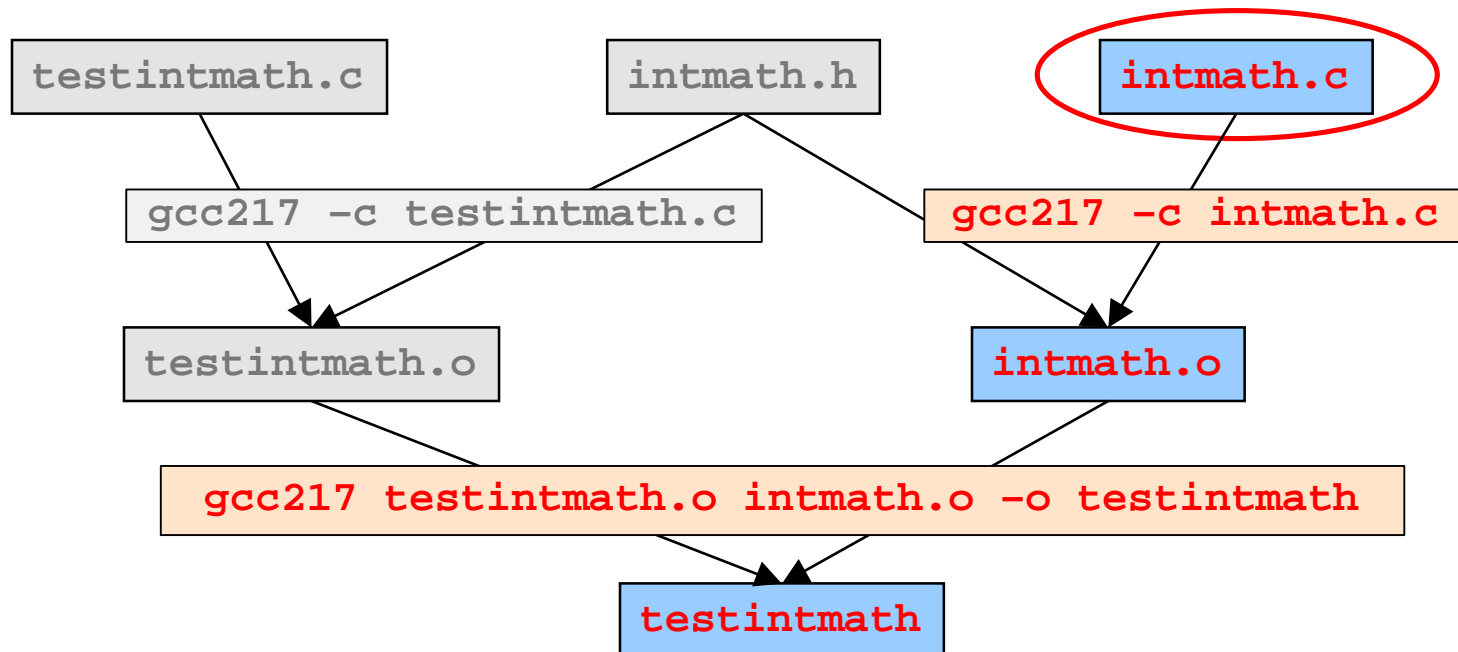


# Partial Builds

Approach 2 allows for **partial builds**

- Example: Change `intmath.c`
  - Must rebuild `intmath.o` and `testintmath`
  - Need not rebuild `testintmath.o`

If program contains many files, could save many hours of build time

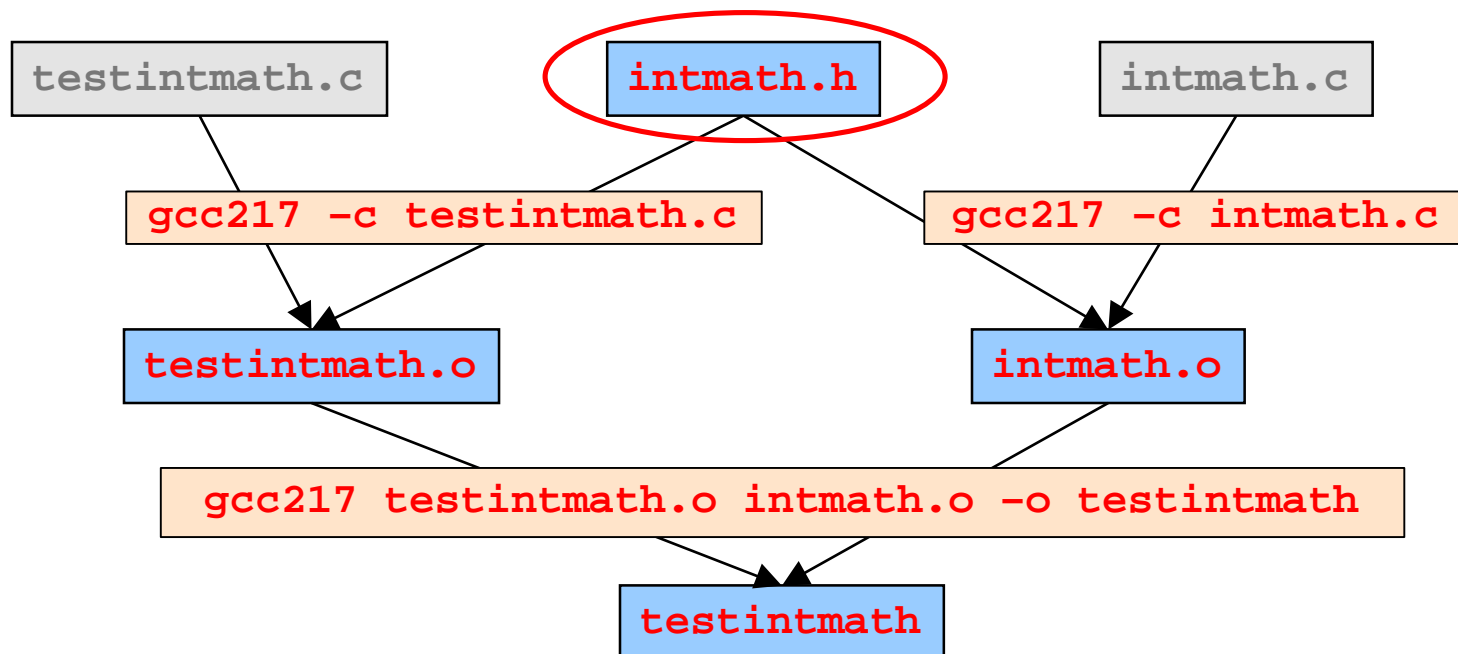




# Partial Builds

However, changing a .h file can be more dramatic

- Example: Change `intmath.h`
  - `intmath.h` is #included into `testintmath.c` and `intmath.c`
  - Must rebuild `testintmath.o`, `intmath.o`, and `testintmath`





# Wouldn't It Be Nice If...

## Observation

- Doing partial builds manually is tedious and error-prone
- Wouldn't it be nice if there were a tool...

## How would the tool work?

- Input:
  - Dependency graph (as shown previously)
    - Specifies file dependencies
    - Specifies commands to build each file from its dependents
  - Date/time stamps of files
- Algorithm:
  - ***If*** file B depends on A ***and***  
date/time stamp of A is newer than date/time stamp of B,  
***then*** rebuild B using the specified command

That's **make!**



# Agenda



Motivation for Make

**Make Fundamentals**

Non-File Targets

Macros



# Make Command Syntax

## Command syntax

```
$ man make
```

### SYNOPSIS

```
make [-f makefile] [options] [targets]
```

- *makefile*
  - Textual representation of dependency graph
  - Contains **dependency rules**
  - Default name is `makefile`, then `Makefile`
- *target*
  - What `make` should build
  - Usually: `.o` file, or an executable binary file
  - Default is first one defined in *makefile*



# Dependency Rules in Makefile

## Dependency rule syntax

```
target: dependencies  
    <tab>command
```

- *target*: the file you want to build
- *dependencies*: the files on which the target depends
- *command*: (after a TAB character) what to execute to create the target

## Dependency rule semantics

- Build *target* iff it is older than any of its *dependencies*
- Use *command* to do the build

Work recursively; examples illustrate...



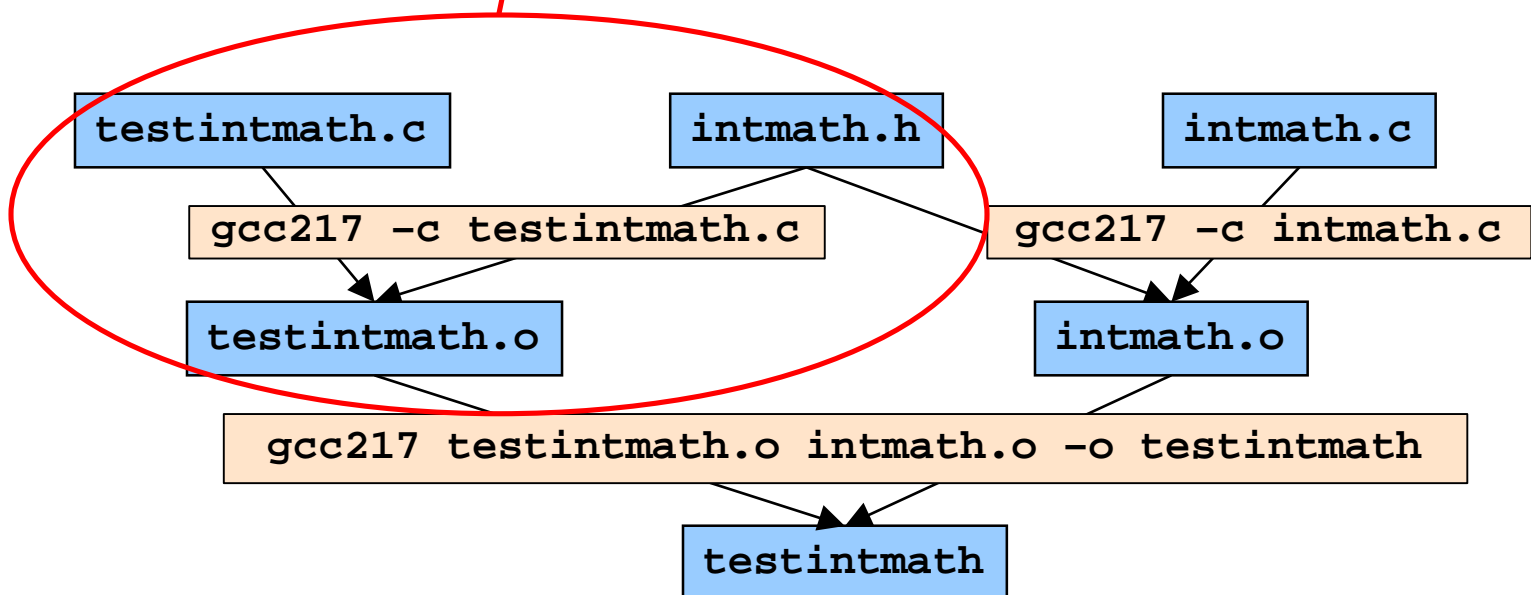
# Makefile Version 1

## Makefile:

```
testintmath: testintmath.o intmath.o
    gcc217 testintmath.o intmath.o -o testintmath

testintmath.o: testintmath.c intmath.h
    gcc217 -c testintmath.c

intmath.o: intmath.c intmath.h
    gcc217 -c intmath.c
```



# Version 1 in Action



At first, to build testintmath  
make issues all three gcc  
commands

Use the touch command to  
change the date/time stamp  
of intmath.c

```
$ make testintmath  
gcc217 -c testintmath.c  
gcc217 -c intmath.c  
gcc217 testintmath.o intmath.o -o testintmath
```

```
$ touch intmath.c
```

```
$ make testintmath  
gcc217 -c intmath.c  
gcc217 testintmath.o intmath.o -o testintmath
```

```
$ make testintmath  
make: `testintmath' is up to date.
```

```
$ make  
make: `testintmath' is up to date.
```

make does a partial build

make notes that the specified  
target is up to date

The default target is testintmath,  
the target of the first dependency rule

# ▶ iClicker Question

Q: If you were making a **Makefile** for this program, what should **a.o** depend on?

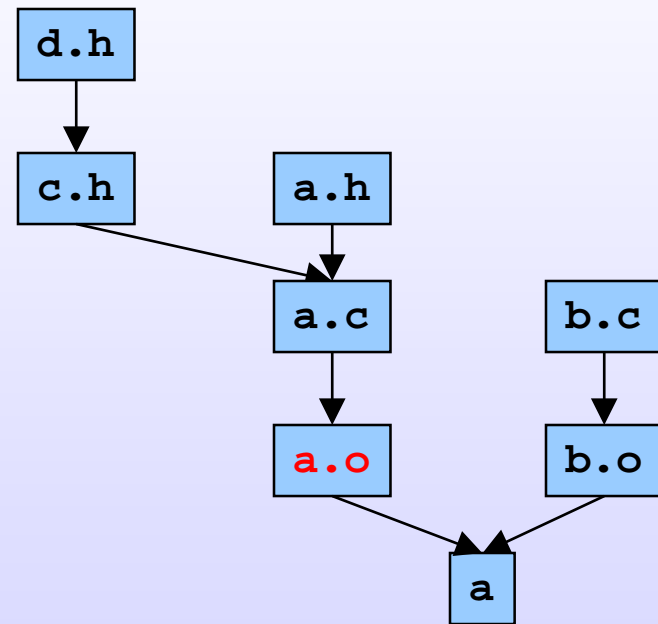
A. a

B. a.c

C. a.c a.h

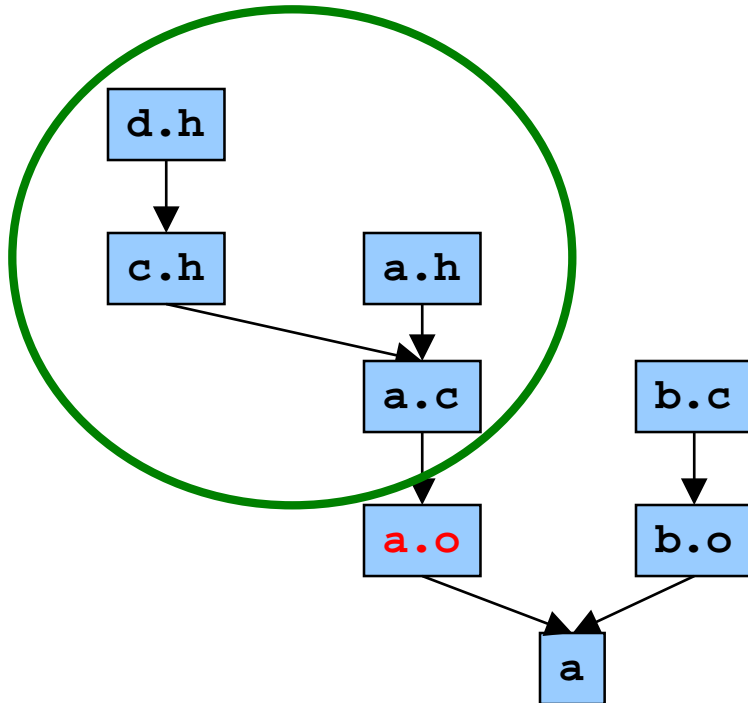
D. a.h c.h d.h

E. a.c a.h c.h d.h





# Makefile Guidelines



```
a.o: a.c a.h c.h d.h
```

In a proper Makefile, each object file:

- Depends upon its .c file
  - Does not depend upon any other .c file
  - Does not depend upon any .o file
- Depends upon any .h files that are #included **directly or indirectly**

# ▶ iClicker Question

Q: If you were making a **Makefile** for this program, what should **a** depend on?

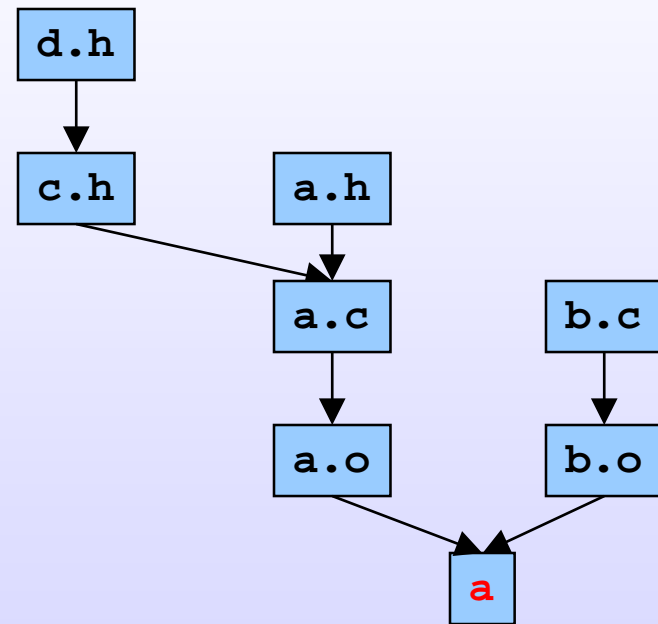
A. a.o b.o

B. a.o b.o a.c b.c

C. a.o b.o a.h c.h d.h

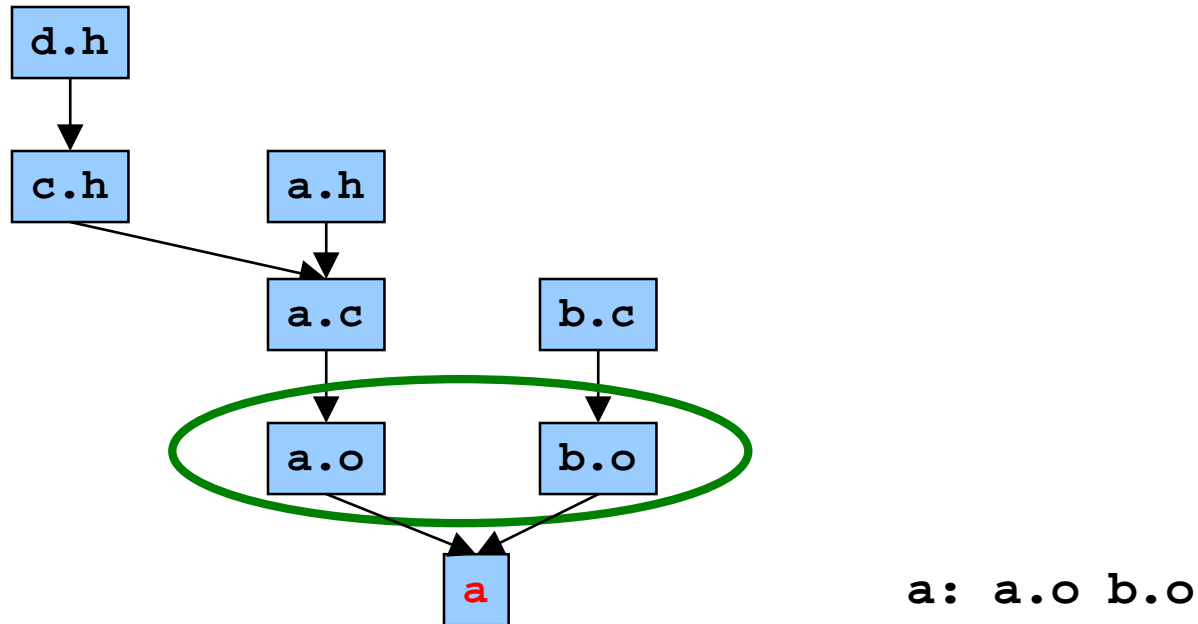
D. a.c b.c a.h c.h d.h

E. a.o b.o a.c b.c a.h c.h d.h





# Makefile Guidelines



In a proper Makefile, each executable:

- Depends upon the .o files that comprise it
- Does not depend upon any .c files
- Does not depend upon any .h files

# Agenda



Motivation for Make

Make Fundamentals

**Non-File Targets**

Macros



# Non-File Targets

## Adding useful shortcuts for the programmer

- **make all**: create the final executable binary file
- **make clean**: delete all .o files, executable binary file
- **make clobber**: delete all Emacs backup files, all .o files, executable

## Commands in the example

- **rm -f**: remove files without querying the user
- Files ending in '~' and starting/ending in '#' are Emacs backup files

```
all: testintmath  
  
clobber: clean  
    rm -f *~ \#\*\#  
  
clean:  
    rm -f testintmath *.o
```

# Makefile Version 2



```
# Dependency rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#\#
clean:
    rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
    gcc217 testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
    gcc217 -c testintmath.c
intmath.o: intmath.c intmath.h
    gcc217 -c intmath.c
```



# Version 2 in Action

make observes that “clean” target doesn’t exist; attempts to build it by issuing “rm” command

```
$ make clean  
rm -f testintmath *.o
```

```
$ make clobber  
rm -f testintmath *.o  
rm -f *~ \#*\#
```

```
$ make all  
gcc217 -c testintmath.c  
gcc217 -c intmath.c  
gcc217 testintmath.o intmath.o -o testintmath
```

```
$ make  
make: Nothing to be done for `all'.
```

Same idea here, but “clobber” depends upon “clean”

“all” depends upon “testintmath”

“all” is the default target

# Agenda



Motivation for Make

Make Fundamentals

Non-File Targets

**Macros**



# Macros

**make** has a macro facility

- Performs textual substitution
- Similar to C preprocessor's `#define`

Macro definition syntax

*macroname* = *macrodefinition*

- **make** replaces `$(macroname)` with *macrodefinition* in remainder of Makefile

Example: Make it easy to change build commands

```
CC = gcc217
```

Example: Make it easy to change build flags

```
CFLAGS = -D NDEBUG -O
```

# Makefile Version 3



```
# Macros
CC = gcc217
# CC = gcc217m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#*\#
clean:
    rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
    $(CC) $(CFLAGS) testintmath.o intmath.o -o testintmath
testintmath.o: testintmath.c intmath.h
    $(CC) $(CFLAGS) -c testintmath.c
intmath.o: intmath.c intmath.h
    $(CC) $(CFLAGS) -c intmath.c
```



# Version 3 in Action



Same as Version 2

# Makefile Gotchas



## Beware:

- Each command (i.e., second line of each dependency rule) must begin with a tab character, not spaces
- Use the `rm -f` command with caution



# Making Makefiles

## In this course

- Create Makefiles manually

## Beyond this course

- Can use tools to generate Makefiles
  - See **mkmf**, others

# Advanced: Implicit Rules



**make** has implicit rules for compiling and linking C programs

- **make** knows how to build x.o from x.c
  - Automatically uses \$(CC) and \$(CFLAGS)
- **make** knows how to build an executable from .o files
  - Automatically uses \$(CC)

**make** has implicit rules for inferring dependencies

- **make** will assume that x.o depends upon x.c

Not required (and potentially confusing):  
see appendix of these slides for details!

# Make Resources



*C Programming: A Modern Approach (King) Section 15.4*

*GNU make*

- <http://www.gnu.org/software/make/manual/make.html>

# Summary



## Motivation for Make

- Automation of partial builds

## Make fundamentals (Makefile version 1)

- Dependency rules, targets, dependencies, commands

## Non-file targets (Makefile version 2)

## Macros (Makefile version 3)



## Debugging (Part 1)



The material for this lecture is drawn, in part, from  
*The Practice of Programming* (Kernighan & Pike) Chapter 5

# Goals of this Lecture



## Help you learn about:

- Strategies and tools for debugging your code

## Why?

- Debugging large programs can be difficult
- A power programmer knows a wide variety of debugging **strategies**
- A power programmer knows about **tools** that facilitate debugging
  - Debuggers
  - Version control systems





# Testing vs. Debugging

## Testing

- What should I do to try to **break** my program?

## Debugging

- What should I do to try to **fix** my program?

# Agenda



## **(1) Understand error messages**

(2) Think before writing

(3) Look for familiar bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes



# Understand Error Messages

Debugging at **build-time** is easier than debugging at **run-time**, if and only if you...

**Understand the error messages!**

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

# Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

Which tool  
(preprocessor,  
compiler, or  
linker) reports  
the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:1:20: error: stdio.h: No such file or
directory
hello.c:2:1: error: unterminated comment
```

# Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{   printf("hello, world\n")
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)



# Understand Error Messages

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

Which tool  
(preprocessor,  
compiler, or  
linker) reports  
the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:6:4: error: expected ';' before 'return'
hello.c:7:1: warning: control reaches end of non-void
function
```

# Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

# Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

Which tool  
(preprocessor,  
compiler, or  
linker) reports  
the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:5:1: warning: implicit declaration of function
'printf'
/tmp/ccLSPMTR.o: In function `main':
hello.c:(.text+0x10): undefined reference to `printf'
collect2: ld returned 1 exit status
```



# Understand Error Messages



```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{  enum StateType
   {  STATE_REGULAR,
     STATE_INWORD
   }
  printf("just hanging around\n");
  return EXIT_SUCCESS;
}
```

What are the errors? (No fair looking at the next slide!)

# Understand Error Messages



```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    enum StateType
    {
        STATE_REGULAR,
        STATE_INWORD
    }
    printf("just hanging around\n");
    return EXIT_SUCCESS;
}
```

What does  
this error  
message even  
mean?

```
$ gcc217 hello.c -o hello
hello.c:9:11: error: expected declaration specifiers or '...'
before string constant
```



# Understand Error Messages

## Caveats concerning error messages

- Line # in error message may be approximate
- Error message may seem nonsensical
- Compiler may not report the real error

## Tips for eliminating error messages

- Clarity facilitates debugging
  - Make sure code is indented properly
- Look for missing semicolons
  - At ends of structure and enum type definitions
  - At ends of function declarations
- Work incrementally
  - Start at first error message
  - Fix, rebuild, repeat

# Agenda



- (1) Understand error messages
- (2) Think before writing**
- (3) Look for familiar bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

# Think Before Writing



Inappropriate changes could make matters worse, so...

Think before changing your code

- Explain the code to:
  - Yourself
  - Someone else
  - A Teddy bear / plushie stuffed tiger?
- Do experiments
  - But make sure they're disciplined



# Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs**
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes



# Look for Common Bugs

Some of our favorites:

```
switch (i)
{ case 0:
  ...
  break;
  case 1:
  ...
  case 2:
  ...
}
```

```
if (i = 5)
  ...
```

```
if (5 < i < 10)
  ...
```

```
int i;
...
scanf("%d", i);
```

```
char c;
...
c = getchar();
```

```
while (c = getchar() != EOF)
  ...
```

```
if (i & j)
  ...
```

What are the errors?

# Look for Common Bugs



Some of our favorites:

```
for (i = 0; i < 10; i++)  
{ for (j = 0; j < 10; i++)  
  { ...  
  }  
}
```

```
for (i = 0; i < 10; i++)  
{ for (j = 10; j >= 0; j++)  
  { ...  
  }  
}
```

What are the errors?





# Look for Common Bugs

Some of our favorites:

```
{  int i;
  ...
  i = 5;
  if (something)
  {  int i; ←
    ...
    i = 6;
    ...
  }
  ...
  printf("%d\n", i);
  ...
}
```

What value is written if this statement is present? Absent?

# Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer**
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

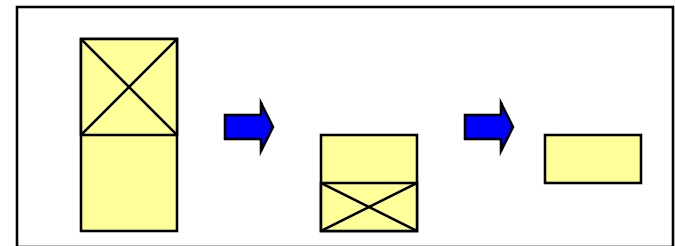
# Divide and Conquer

Divide and conquer: To debug a **program...**

- Incrementally find smallest **input file** that illustrates the bug

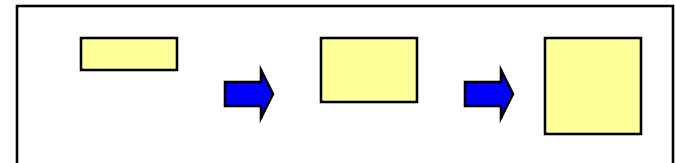
- Approach 1: **Remove** input

- Start with file
- Incrementally remove lines until bug disappears
- Examine most-recently-removed lines



- Approach 2: **Add** input

- Start with small subset of file
- Incrementally add lines until bug appears
- Examine most-recently-added lines





# Divide and Conquer

Divide and conquer: To debug a **module**...

- Incrementally find smallest **client subset** that illustrates the bug
- Approach 1: **Remove** code
  - Start with test client
  - Incrementally remove lines of code until bug disappears
  - Examine most-recently-removed lines
- Approach 2: **Add** code
  - Start with minimal client
  - Incrementally add lines of test client until bug appears
  - Examine most-recently-added lines

# Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests**
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

# Add More Internal Tests



## (5) Add more internal tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal test also can help **eliminate** bugs
  - Validating parameters & checking invariants can eliminate some functions from the bug hunt

# Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output**
- (7) Use a debugger
- (8) Focus on recent changes

# Display Output



Write values of important variables at critical spots

- Poor:

```
printf("%d", keyvariable);
```

`stdout` is buffered;  
program may crash  
before output appears

- Maybe better:

```
printf("%d\n", keyvariable);
```

Printing '`\n`' flushes  
the `stdout` buffer, but  
not if `stdout` is  
redirected to a file

- Better:

```
printf("%d", keyvariable);  
fflush(stdout);
```

Call `fflush()` to flush  
`stdout` buffer  
explicitly





# Display Output

- Maybe even better:

```
fprintf(stderr, "%d", keyvariable);
```

Write debugging output to `stderr`; debugging output can be separated from normal output via redirection

- Maybe better still:

```
FILE *fp = fopen("logfile", "w");  
...  
fprintf(fp, "%d", keyvariable);  
fflush(fp);
```

Bonus: `stderr` is unbuffered

Write to a log file

# Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger**
- (8) Focus on recent changes

# Use a Debugger



## Use a debugger

- Alternative to displaying output

# The GDB Debugger



## GNU Debugger

- Part of the GNU development environment
- Integrated with Emacs editor
- Allows user to:
  - Run program
  - Set breakpoints
  - Step through code one line at a time
  - Examine values of variables during run
  - Etc.

For details see precept tutorial, precept reference sheet, Appendix 2 of these slides



# Agenda

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes**



# Focus on Recent Changes

## Focus on recent changes

- Corollary: Debug now, not later

### Difficult:

- (1) Compose entire program
- (2) Test entire program
- (3) Debug entire program

### Easier:

- (1) Compose a little
- (2) Test a little
- (3) Debug a little
- (4) Compose a little
- (5) Test a little
- (6) Debug a little
- ...

# Focus on Recent Changes



## Focus on recent change (cont.)

- Corollary: Maintain old versions

### Difficult:

- (1) Change code
- (2) Note new bug
- (3) Try to remember what changed since last version

### Easier:

- (1) Backup current version
- (2) Change code
- (3) Note new bug
- (4) Compare code with last version to determine what changed

# Maintaining Old Versions



To maintain old versions...

Approach 1: Manually copy project directory

```
...  
$ mkdir myproject  
$ cd myproject
```

*Create project files here.*

```
$ cd ..  
$ cp -r myproject myprojectDateTime  
$ cd myproject
```

*Continue creating project files here.*

```
...
```





# Maintaining Old Versions

Approach 2: Use a **Revision Control System** such as subversion or git

- Allows programmer to:
  - **Check-in** source code files from **working copy** to **repository**
  - **Commit** revisions from **working copy** to **repository**
    - saves all old versions
  - **Update** source code files from **repository** to **working copy**
    - Can retrieve old versions
- Appropriate for one-developer projects
- Extremely useful, almost *necessary* for multideveloper projects!

Not required for COS 217, but good to know!

Google “subversion svn” or “git” for more information.

# Summary



## General debugging strategies and tools:

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
  - Use GDB!!!
- (8) Focus on recent changes
  - Consider using git, etc.

# Appendix 1: Implicit Rules



**make** has implicit rules for compiling and linking C programs

- **make** knows how to build `x.o` from `x.c`
  - Automatically uses `$(CC)` and `$(CFLAGS)`
- **make** knows how to build an executable from `.o` files
  - Automatically uses `$(CC)`

```
intmath.o: intmath.c intmath.h
$(CC) $(CFLAGS) -c intmath.c
```



```
intmath.o: intmath.c intmath.h
```

```
testintmath: testintmath.o intmath.o
$(CC) testintmath.o intmath.o -o testintmath
```



```
testintmath: testintmath.o intmath.o
```

# Makefile Version 4



```
# Macros
CC = gcc217
# CC = gcc217m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#*\#
clean:
    rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
testintmath.o: testintmath.c intmath.h
intmath.o: intmath.c intmath.h
```

# Version 4 in Action



Same as Version 2

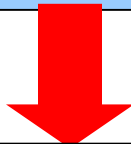


# Implicit Dependencies

`make` has implicit rules for inferring dependencies

- `make` will assume that `x.o` depends upon `x.c`

```
intmath.o: intmath.c intmath.h
```



```
intmath.o: intmath.h
```

# Makefile Version 5



```
# Macros
CC = gcc217
# CC = gcc217m
CFLAGS =
# CFLAGS = -g
# CFLAGS = -D NDEBUG
# CFLAGS = -D NDEBUG -O

# Dependency rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#*\#
clean:
    rm -f testintmath *.o

# Dependency rules for file targets
testintmath: testintmath.o intmath.o
testintmath.o: intmath.h
intmath.o: intmath.h
```

# Version 5 in Action



Same as Version 2



# Makefile Gotchas



## Beware:

- To use an implicit rule to make an *executable*, the executable must have the same name as one of the `.o` files

Correct:

```
myprog: myprog.o someotherfile.o
```



Won't work:

```
myprog: somefile.o someotherfile.o
```



# Appendix 2: Using GDB



An example program

File testintmath.c:

Euclid's algorithm;  
Don't be concerned  
with details

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

```
int gcd(int i, int j)
{
    int temp;
    while (j != 0)
    {
        temp = i % j;
        i = j;
        j = temp;
    }
    return i;
}
```

```
int lcm(int i, int j)
{
    return (i / gcd(i, j)) * j;
}
...
```

```
...
int main(void)
{
    int iGcd;
    int iLcm;
    iGcd = gcd(8, 12);
    iLcm = lcm(8, 12);
    printf("%d %d\n", iGcd, iLcm);
    return 0;
}
```

The program is correct

But let's pretend it has a  
runtime error in **gcd()**...

# Using GDB



## General GDB strategy:

- Execute the program to the point of interest
  - Use breakpoints and stepping to do that
- Examine the values of variables at that point

# Using GDB



## Typical steps for using GDB:

(a) Build with `-g`

```
gcc217 -g testintmath.c -o testintmath
```

- Adds extra information to executable file that GDB uses

(b) Run Emacs, with no arguments

```
emacs
```

(c) Run GDB on executable file from within Emacs

```
<Esc key> x gdb <Enter key> testintmath <Enter key>
```

(d) Set breakpoints, as desired

```
break main
```

- GDB sets a breakpoint at the first executable line of `main()`

```
break gcd
```

- GDB sets a breakpoint at the first executable line of `gcd()`

# Using GDB



## Typical steps for using GDB (cont.):

(e) Run the program

**run**

- GDB stops at the breakpoint in main()
- Emacs opens window showing source code
- Emacs highlights line that is to be executed next

**continue**

- GDB stops at the breakpoint in gcd()
- Emacs highlights line that is to be executed next

(f) Step through the program, as desired

**step** (repeatedly)

- GDB executes the next line (repeatedly)

• Note: When next line is a call of one of your functions:

- **step** command *steps into* the function
- **next** command *steps over* the function, that is, executes the next line without stepping into the function

# Using GDB



## Typical steps for using GDB (cont.):

(g) Examine variables, as desired

```
print i  
print j  
print temp
```

- GDB prints the value of each variable

(h) Examine the function call stack, if desired

```
where
```

- GDB prints the function call stack
- Useful for diagnosing crash in large program

(i) Exit gdb

```
quit
```

(j) Exit Emacs

```
<Ctrl-x key> <Ctrl-c key>
```

# Using GDB



## GDB can do much more:

- Handle command-line arguments  
`run arg1 arg2`
- Handle redirection of stdin, stdout, stderr  
`run < somefile > someotherfile`
- Print values of expressions
- Break conditionally
- Etc.