Systems and Internet
Infrastructure Security

Institute for Networking and Security Research
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA

Build Processing

Devin J. Pohly <djpohly@cse.psu.edu>

Unix shell redirections

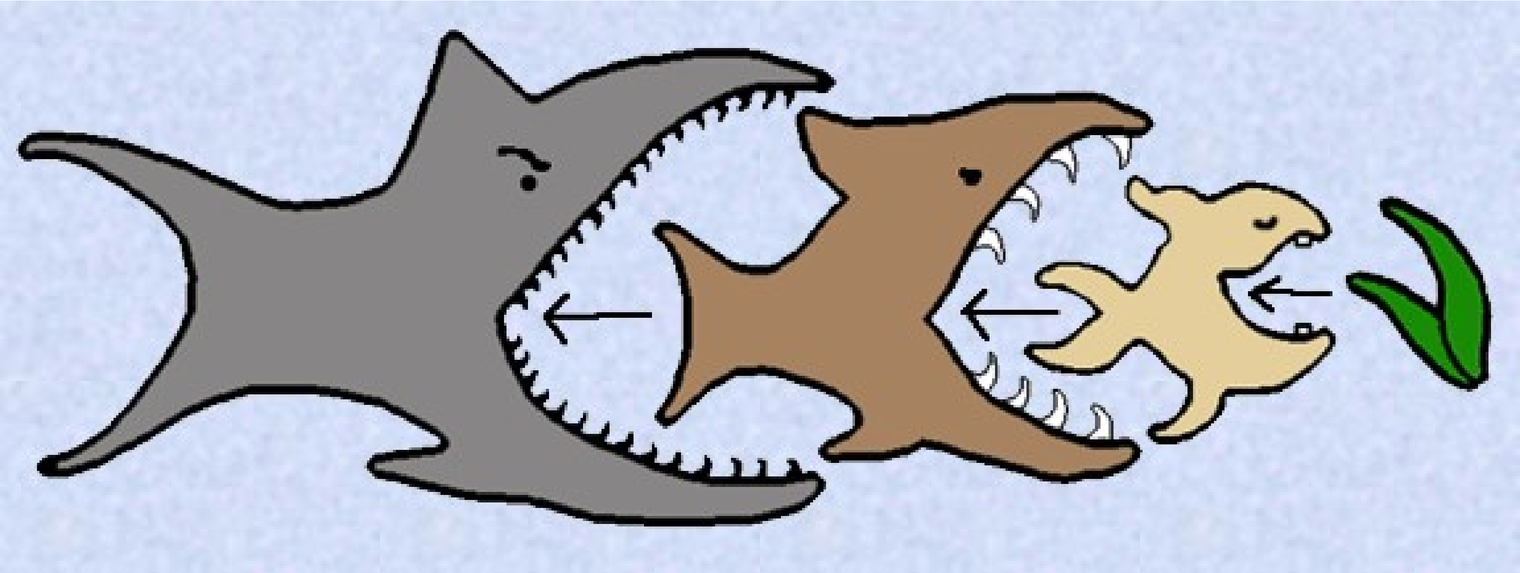
- You can assign a file to stdin, stdout, or stderr
    - Known as “redirection”
    - printf and scanf use the file instead of keyboard/display
- Syntax:

```
$ ./assign2 < numbers.txt
$ ./assign2 > output.txt
$ ./assign2 < numbers.txt >
      output.txt
```

- Or use `>>` to append

Unix pipes

- You can also connect stdout to another program!
  - Chain of programs called a "pipeline"
  - Separate the commands with a | (vertical bar) character
  - Programs designed as filters, from stdin to stdout
- Useful utilities: `cat` prints a file to stdout, `sort` reads stdin and sorts it line by line to stdout (use the `-n` option to sort numbers)

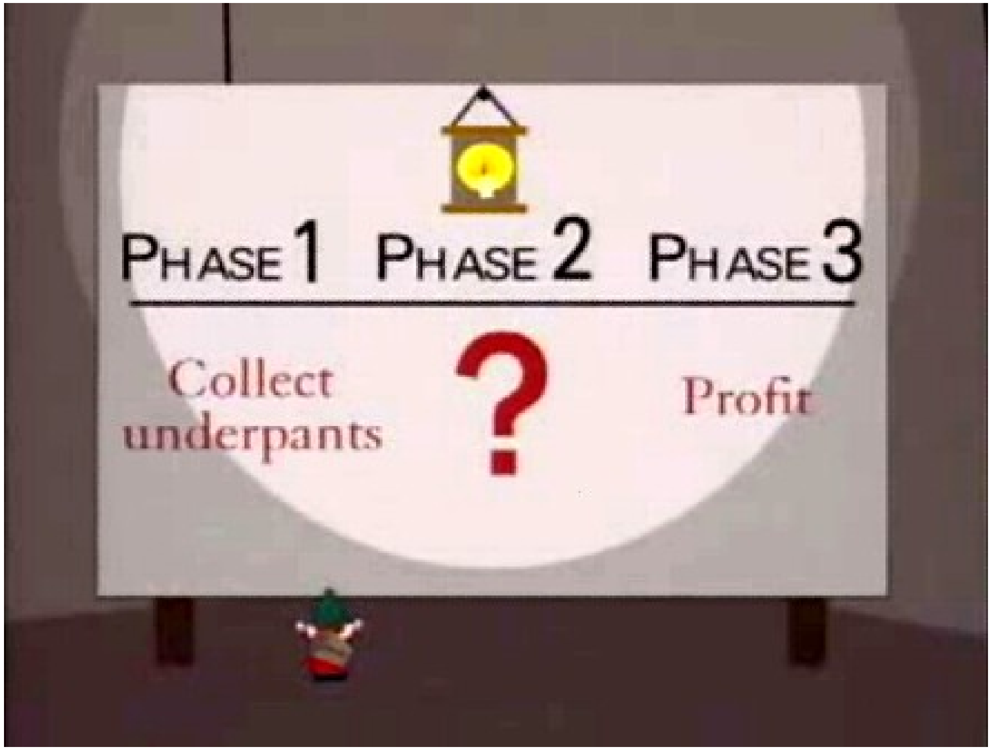


Pipeline examples

```
djpohly@chiri$ cat numbers.txt
313.11
45.64
9.50
113.89
djpohly@chiri$ cat numbers.txt | sort -n
9.50
45.64
113.89
313.11
djpohly@chiri$ echo hello world | rev | xxd -c8
0000000: 646c 726f 7720 6f6c  dlrow ol
0000008: 6c65 680a            leh.
djpohly@chiri$ echo cmpsc133 | sed 's/133/311/g'
cmpsc311
```

Building a program 101

- Two major phases of building a program are *compiling* and *linking*
  - gcc is used to compile the program
  - ld is used to link the program
  - gcc can also be used to link (it just executes ld)



Compiling sources

- Run gcc to compile

```
gcc [options] sourcefile(s)
```

- Interesting options
  - `-c`: stop after compiling (object files), don't link
  - `-Wall`: show all standard warnings (`-Wextra` for more)
  - `-g`: generate debug information
  - `-o filename.o`: write output to given file
- For example:

```
gcc -c -Wall -g -o hello.o hello.c
```

Linking object files

- Run gcc or ld to link

```
gcc [options] objfile(s)
```

- Interesting options
  - `-g`: generate debug information
  - `-o` *`filename`*: write output to given file
  - `-l`*`NAME`*: link with the library lib*NAME*
- For example:

```
gcc -g -o hello -lpng hello.o goodbye.o
```

Building a static library

- A statically linked library produces object code that is inserted into a program at *link* time.
  - This is an "archive" of object files which the linker uses to search for and transfer code into your program.
  - To create a static library, use

```
ar rcs library objfile(s)
```

- Library naming: static libraries are virtually always named `libsomething.a`, e.g.:

```
ar rcs libdoge.a such.o very.o amaze.o
```

- Later, to link a program with this library, link against the name of the *library* (`-ldoge`), not the name of the *file*

Building a static library

- A statically linked library produces object code that is inserted into a program at *link* time.
  - This is an "archive" of object files which the linker uses to search for and transfer code into your program.
  - To create a static library, use

```
ar rcs library objfile(s)
```

r – replace files in the archive
c – create the archive if it doesn't exist
s – create an index for "relocatable code"

- Library na c libraries are name

```
ar rcs libdoge.a such.o very.o amaze.o
```

- Later, to link a program with this library, link against the name of the *library* (`-ldoge`), not the name of the *file*

Building a dynamic library

- A dynamically linked library produces object code that is inserted into the program at *execution (load)* time.
  - This is a loadable version of the library which the loader uses to launch the application
  - To create a dynamic library, pass `-shared` to gcc:

```
gcc -shared -o libBLT.so bacon.o lettuce.o tomato.o
```

- Naming: `libsomething.so` ("shared object" file)
- Important: all object files in a shared library must have been compiled to *position-independent code* (PIC)
  - PIC can be placed anywhere in memory (virtual memory)
  - E.g., it only uses relative jump/branch instructions

Building a dynamic library

- A dynamically linked library produces object code that is inserted into the program at *execution (load)* time.
  - This is a loadable version of the library which the loader uses to launch the application
  - To create a dynamic library, pass `-shared` to gcc:

```
gcc -shared -o libBLT.so bacon.o lettuce.o tomato.o
```

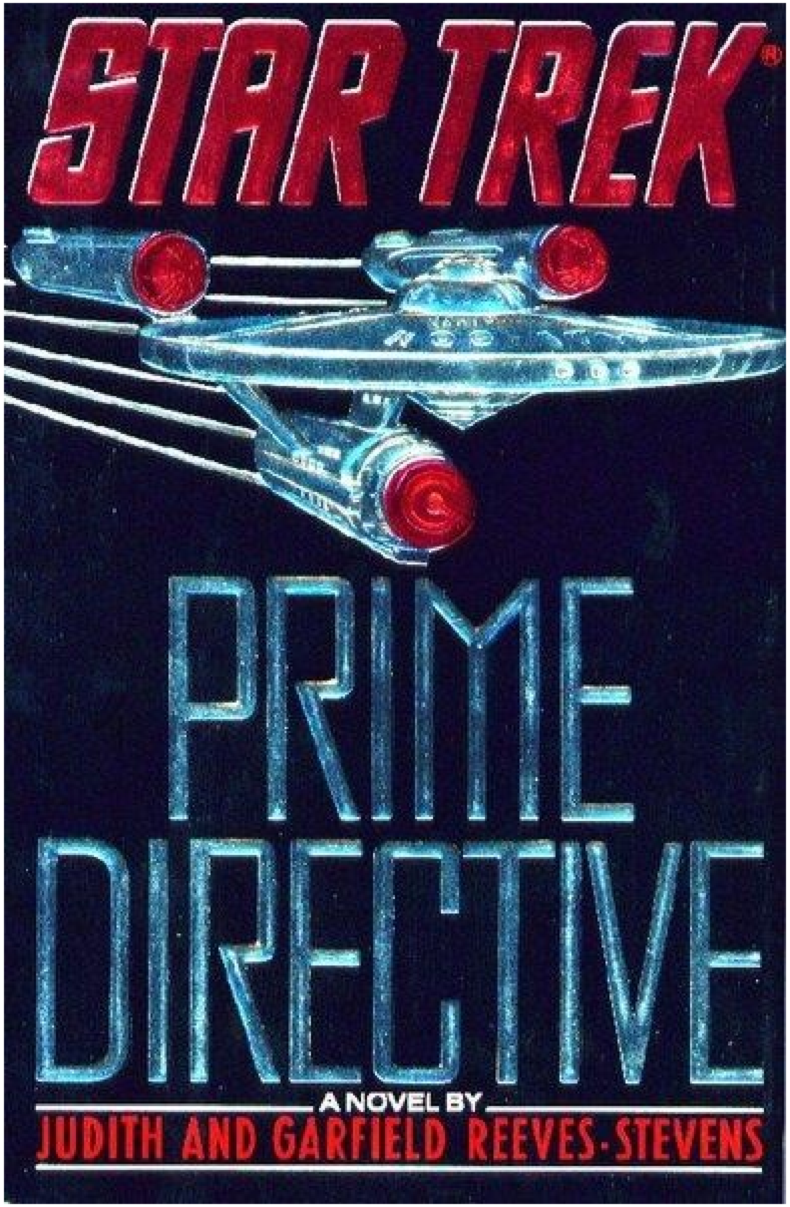
- Naming: l

```
gcc -fpic -c -o bacon.o bacon.c
```

- Important: all object files in a shared library must have been compiled to *position-independent code* (PIC)
  - PIC can be placed anywhere in memory (virtual memory)
  - E.g., it only uses relative jump/branch instructions

The C preprocessor

- Preprocessor takes input source code files and “fills them out” before they get to the compiler
    - Reads “directives”: commands that start with the # character.
    - We have already seen include directives.
    - There are more!



#include

- The `#include` directive tells the preprocessor to insert the contents of an entire file
  - `#include "foo.h"`: include file is in the local directory
  - `#include <foo.h>`: include file is in the default system directories or one provided on the command line with `-I`
- The gcc `-I`*`path`* option
  - Tells the preprocessor to look in a specific *path* for include files (when using the < > style)
  - Can be repeated to specify multiple paths

```
gcc -I/usr/include/SDL -I/usr/include/X11 -o prog prog.c
```

#define

- The #define directive allows the user to define a macro that is used throughout the program
  - Simply replaced with the value any time it is used
  - Often a simple constant that might be changed later, e.g., the size of arrays/buffers

```
#define NUMBER_ENTRIES 15

int main(int argc, char *argv[])
{
    // Declare your variables here
    double inputs[NUMBER_ENTRIES];

    // Read input values
    for (i = 0; i < NUMBER_ENTRIES; i++) {
        scanf("%lf", &inputs[i]);
    }

    // ...
```

#define

- #define macros can also take arguments
  - These are not functions – still just simple replacement, but with parameters.
  - No function call overhead... but can create tricky or hard-to-find errors if you aren’t careful!

```
#define SWAP(x, y) {int temp = x; x = y; y = temp;}

int main(int argc, char *argv[])
{
    // Declare your variables here
    int i = 1, j = 2;

    // ...
    SWAP(i, j);

    // ...
```

Conditional compilation

```
#define CAT_ALIVE

#ifdef TOTALLY_NOT_DEFINED
/* This isn't compiled */
#else
/* but this is.
#endif

#ifndef CAT_ALIVE
/* Poor Schroedinger... */
#else
/* It's OK, this part will be compiled */
#endif
```

```
int main(int argc, char *argv[])
{
    // Declare your variables here
    double inputs[NUMBER_ENTRIES];

#if 0
    // Parts I haven't implemented yet
    // ...
#endif
    return 0;
}
```

- You can conditionally compile parts of a program using the `#if`, `#ifdef`, and `#ifndef` directives
  - `#if 0` can be used to temporarily “remove” code from the compile – like a super-comment

Make

- `make` is a utility for automating any complex build process
  - Figures out which parts of the build are out of date
  - Figures out the dependencies between objects
  - Issues commands to rebuild anything that is outdated

Note: being an efficient systems programmer requires mastering this tool!

Make basics

- Each system you want to build has one or more "Makefiles" which define how to build it:
  - What to build
  - How to build it
  - And when it needs to be rebuilt
- Terminology
  - Target: anything that can be built
  - Prerequisites: things you need to build the target
  - Recipe: commands that build the target from the prerequisites
  - Rule: statement of targets, prerequisites, and a recipe

Makefile rules

- Rules define how targets are built. The syntax is:

```
target: prereq1 prereq2 prereq3 ...
        command1
        command2
        ...
```

- Where
    - `target` is the thing to be built
    - Each `prereq` is something needed to build the target
    - `commands` are the list of Unix commands to run to build it
- *Key idea*: run the commands to build the target if any prerequisite has been changed since the last build.
    - The target is said to be "out of date" if this is the case.

Makefile rules

- Rules define how targets are built. The syntax is:

```
target: prereq1 prereq2 prereq3 ...
        command1
        command2
        ...
```

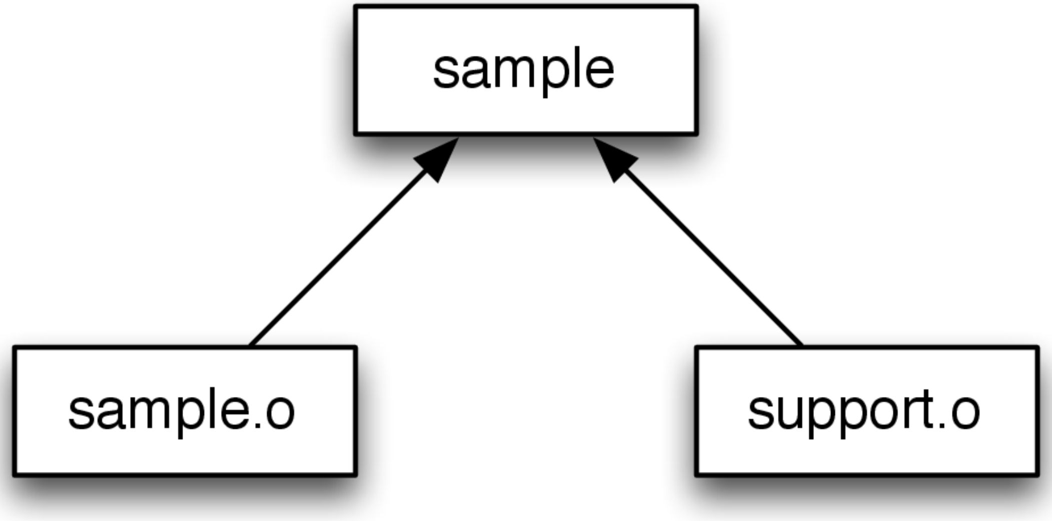
- Where
  - `target` is t

Recipe MUST BE TABBED OVER!
Not spaces!

  - Each `prereq` is something needed to build the target
  - `commands` are the list of Unix commands to run to build it
- *Key idea*: run the commands to build the target if any prerequisite has been changed since the last build.
  - The target is said to be “out of date” if this is the case.

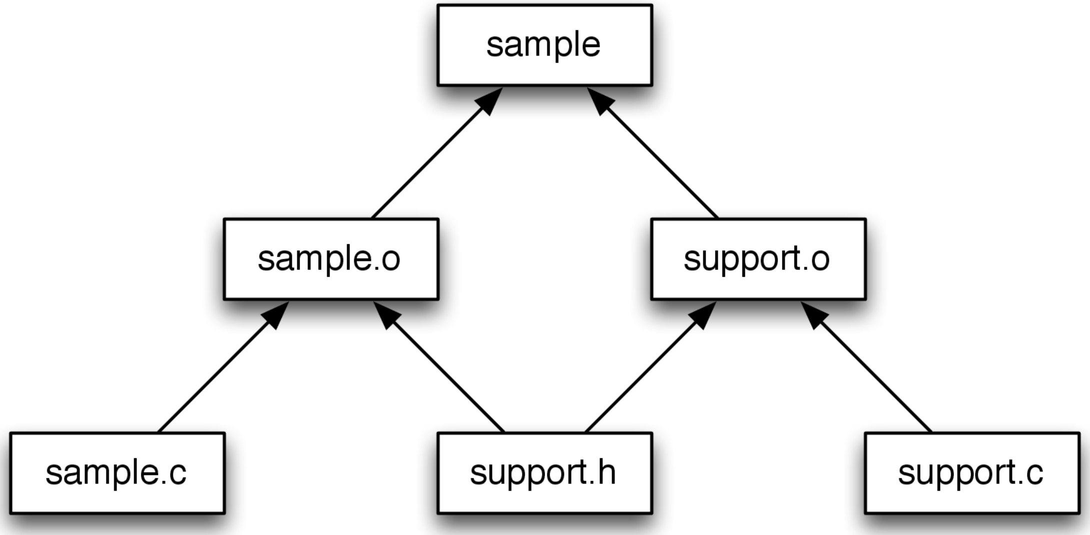
Dependencies

```
sample: sample.o support.o
        gcc -o sample sample.o support.o
```



More dependencies!

```
sample: sample.o support.o
        gcc -o sample sample.o support.o

sample.o: sample.c support.h
        gcc -c -Wall -I. -o sample.o sample.c

support.o: support.c support.h
        gcc -c -Wall -I. -o support.o support.c
```



Download in-class code

```
$ wget -U mozilla tiny.cc/311make
$ tar -xvzf 311make
$ cd make-demo
$ ls
$ vim Makefile
```

Makefile practice

```
# Compile our program with strict warning settings
addit:
        gcc -Werror -Wall -Wextra -o addit addit.c
```

- Save, exit, and run `make addit`. Run `./addit`.
- Edit addit.h and change the value.
- Run make addit again. What happens?
- Run addit. What happens?

Makefile practice

```
# Recompile if something changes
addit: addit.c addit.h
        gcc -Werror -Wall -Wextra -o addit addit.c
```

- Now run make addit again. Run addit to see your changes.
- Edit addit.h, change the value, save and exit.
- Make addit, run addit. What happens this time?

Variables

- Used to keep track of things you may want to tweak
  - Lists of files or options
  - Alternate programs (compiler, linker, etc.)
- Common variables:
  - CC: compiler (gcc -c)
  - LD: linker (ld)
  - CFLAGS: compiler options
  - LDFLAGS: linker options

Makefile practice

```
# Specify compiler and settings
CC = gcc
CFLAGS = -Werror -Wall -Wextra

# Recompile if something changes
addit: addit.c addit.h
        $(CC) $(CFLAGS) -o addit addit.c
```

- Note that all the flags still show up when Make prints the command.
- What if you edit addit.h and just run `make` with no target?
    - Default target: first target defined in the file

Phony targets

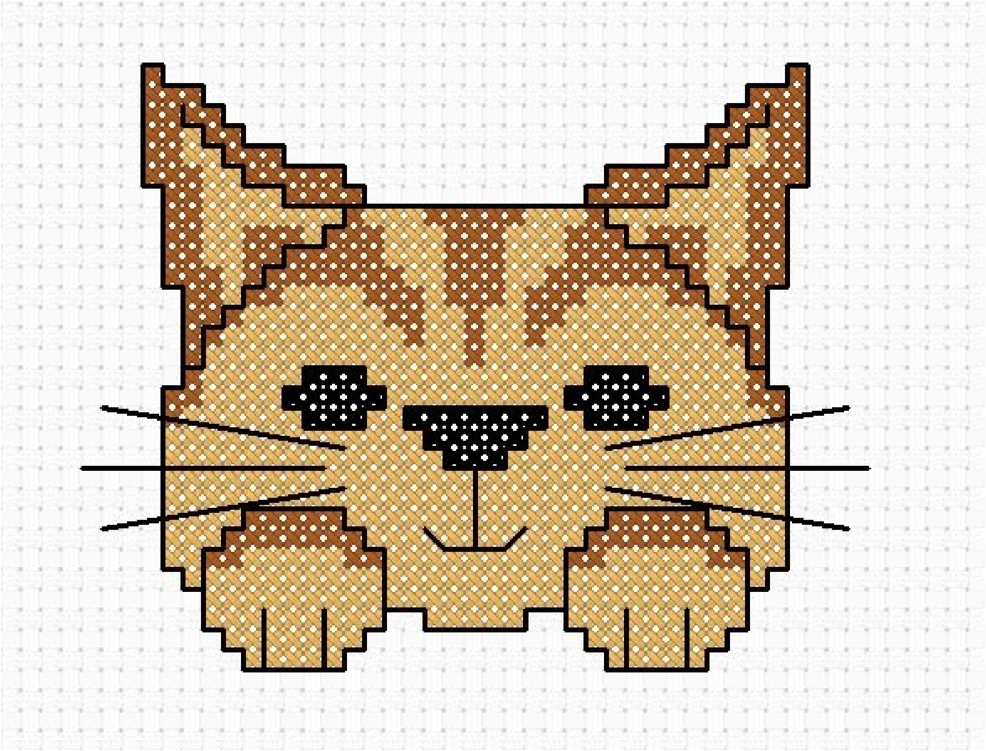
- Convenient names for build actions
  - make all: build everything
  - make clean: remove anything that isn't source code
  - make install: install the built files in the right place
- Not the name of a file
- Always out-of-date – always build if you ask for them

Makefile practice

```
# Specify compiler and settings
CC = gcc
CFLAGS = -Werror -Wall -Wextra

# Set up a phony target to build everything
.PHONY: all

all: addit

# Recompile if something changes
addit: addit.c addit.h
        $(CC) $(CFLAGS) -o addit addit.c
```

Pattern rules

- Rules based on filename patterns rather than specific filenames
    - Use `%` for a wildcard
    - Example: `%.o: %.c`
- Builds *any* file matching the target pattern by using the corresponding prerequisites



Automatic variables

- Don’t want specific filenames in the recipe for a pattern rule!
- “Automatic variables” to use instead of filenames
    - `$@`: target filename
    - `$^`: the whole list of prerequisites
    - `$<`: just the first prerequisite
    - and more

Makefile practice

```
# Specify compiler and settings
CC = gcc
CFLAGS = -Werror -Wall -Wextra

# Set up a phony target to build everything
.PHONY: all

all: addit

# Compile any simple program
%: %.c
        $(CC) $(CFLAGS) -o $@ $^
```

- Try both `make addit` and `make hello` now.
- Edit addit.h and re-make. What's wrong?

Makefile practice

```
# Specify compiler and settings
CC = gcc
CFLAGS = -Werror -Wall -Wextra

# Set up a phony target to build everything
.PHONY: all

all: addit

# Add an additional dependency without affecting
# the recipe
addit: addit.h

# Compile any simple program
%: %.c
        $(CC) $(CFLAGS) -o $@ $^
```