CSSE2310

•••

Modularity and Makefiles

Week 7, Sem 1, 2020

Assignment 1 Feedback

- Look through your specific feedback - available through `2310tool`
- Don't leave style to the end. Good style (and structure) from the start helps you do a better job.
- Ask tutors in quiet pracs about how you could do better in A1, plan for A3
- The 50 lines per function limit is very doable:
    - Split functionality into small parts: each function does one small thing
    - Review code you have written, consider if it can or should be re-written better.
    - Don't repeat functionality!

Assignment 1 Feedback

- Run repmarka1.sh
- People didn’t have (correct) makefiles committed.
- Test each small part of functionality as you write it. Test often, fail often, learn fast, develop fast, achieve more.
- Commit every time you get something working

The past 7 weeks… we're halfway!

In **lectures** we have covered:

- C + SVN
- OS
- Shell
- Processes
- Virtual Memory
- Files & Pipes

In **tutorials** we have covered:

- C + Linux
- Makefile basics
- Testing system
- Debugging
- Some theory questions

In **assignments** we have covered:

- C + SVN
- Makefile basics
- Debugging

Why are we coming back to makefiles?

- Assignment 1 was small, we only needed a simple makefile
- Assignments 3 & 4 (and many real-world projects) aren’t small and simple

In most cases you will have:

- Multiple source files which need to be compiled together.
- Multiple programs to compile.

Linker Modularity vs Function Pointers

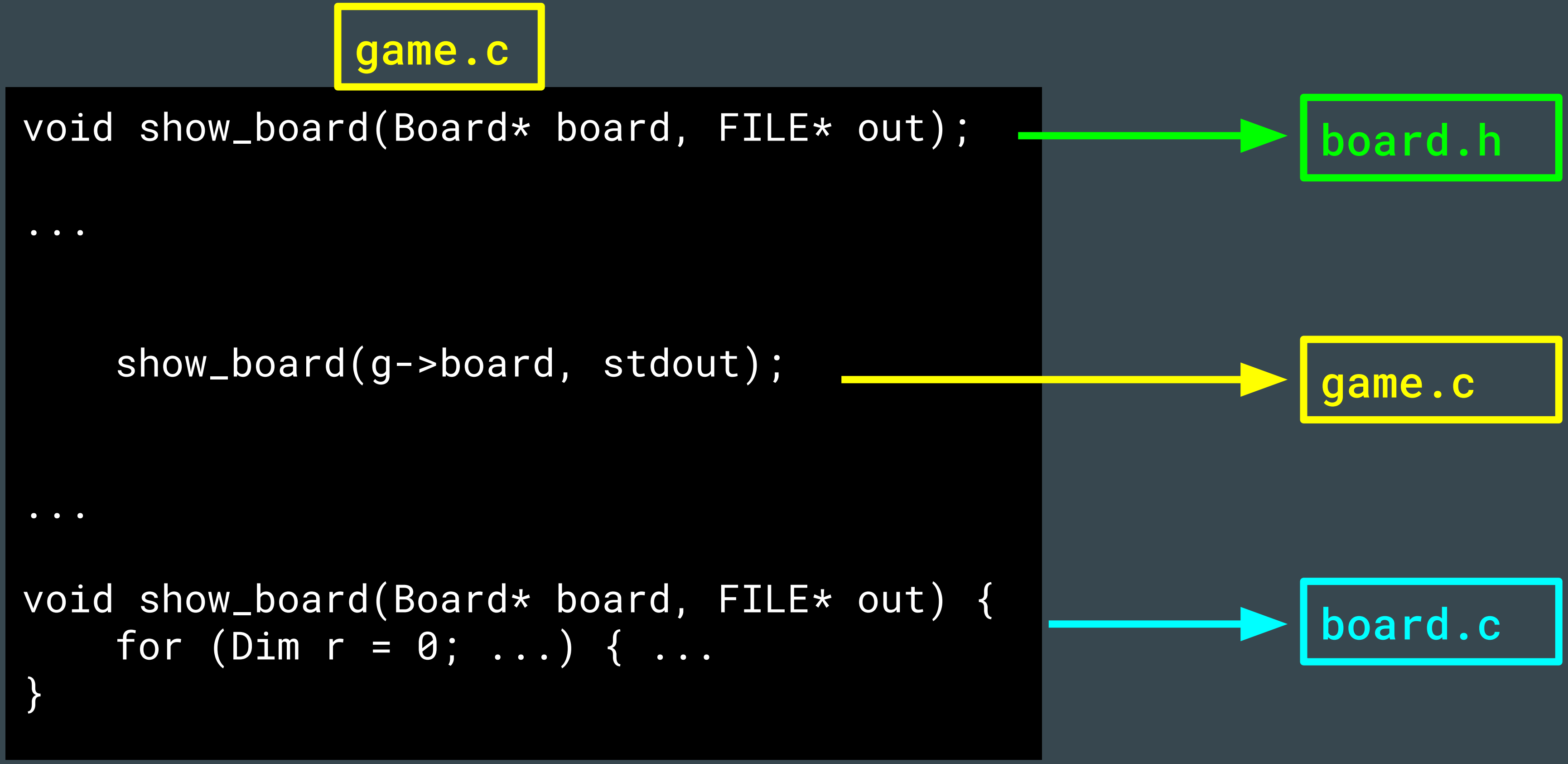
Linker Modularity

- Multiple source files.
- Function is implemented differently in different files.
- Linker (part of gcc) will choose the functionality based on input files at **compile time**.
- Good use case from assignment 1:
  - Move scoring functions to a separate file, so variations could be made with different scoring rules.

Function Pointers

- Multiple different functions, but all have same type signature.
  - Same parameter types & ordering.
  - Same return type.
- Function selected at **run time**.
- Good use case from assignment 1:
  - Player struct uses function pointer to make a move.
  - Function pointer is either set to the human player function or to one of the automated player functions.

Structuring Multiple Source Files



game.c

```
void show_board(Board* board, FILE* out);

...


    show_board(g->board, stdout);


...

void show_board(Board* board, FILE* out) {
    for (Dim r = 0; ...) { ...
}
```

board.h

game.c

board.c

Structuring Multiple Source Files

game.c

```
#include “board.h”
...

    show_board(g->board, stdout);

...
```

board.h

```
...

void show_board(Board* board,
        FILE* out);
...
```

board.c

```
...

void show_board(Board* board,
        FILE* out) {
    for (Dim r = 0; ...) { ...
}
...
```

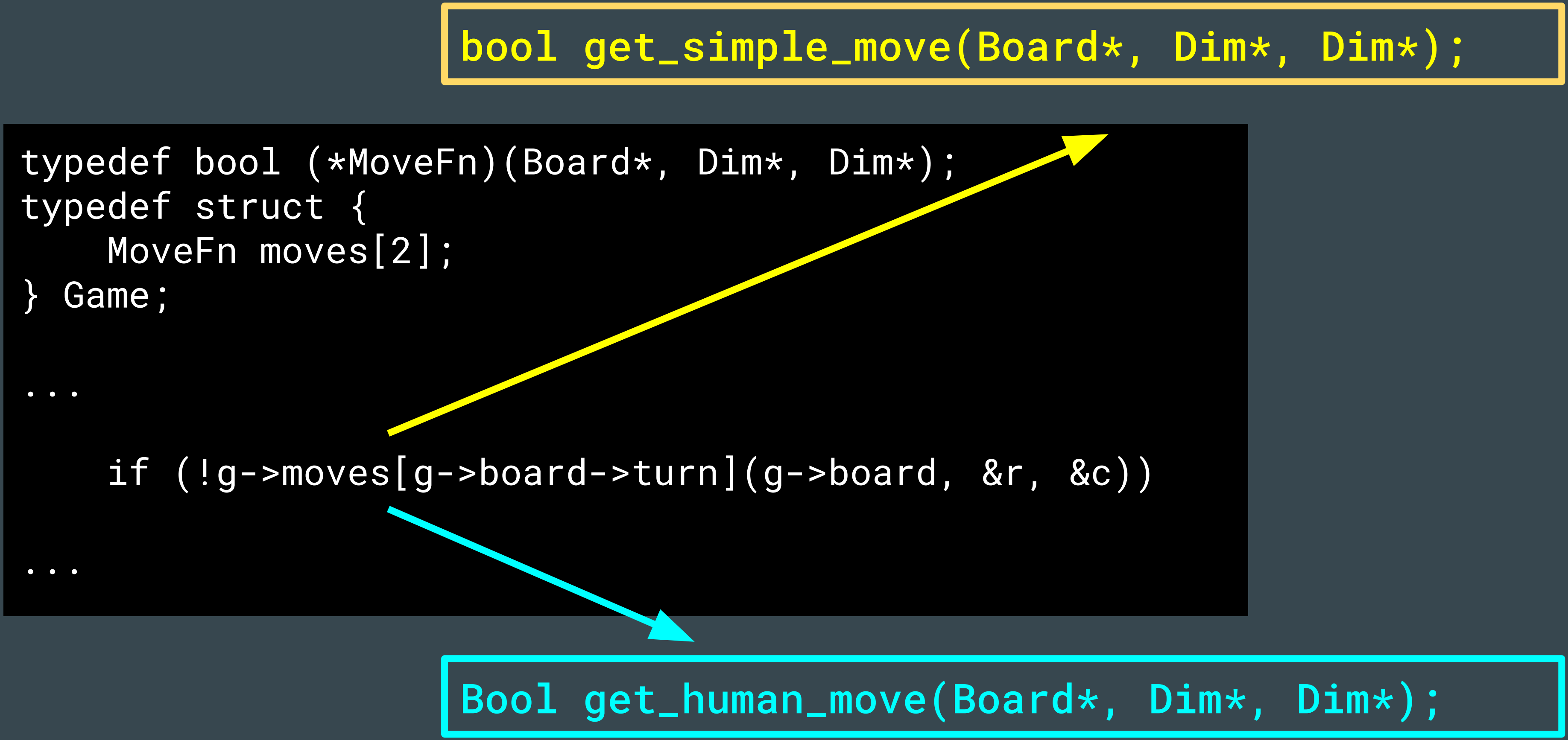
Header guards

```
#ifndef __NAME_H__
#define __NAME_H__

    Function declarations

#endif
```

Header guards are pieces of code that protect the contents of a **header** file from being included more than once.

Header guards should be unique

Function Pointer Modularity: Player Type



```
bool get_simple_move(Board*, Dim*, Dim*);
```

```
typedef bool (*MoveFn)(Board*, Dim*, Dim*);
typedef struct {
    MoveFn moves[2];
} Game;

...

    if (!g->moves[g->board->turn](g->board, &r, &c))

...
```

```
Bool get_human_move(Board*, Dim*, Dim*);
```

Linker modularity for multiple auto players

player.h

```
...

bool get_auto_move(Board*, Dim*, Dim*);
...
```

autoType0.c

```
...

bool get_auto_move(
        Board* b, Dim* r, Dim* c)
{
    // logic for type 0

...
```

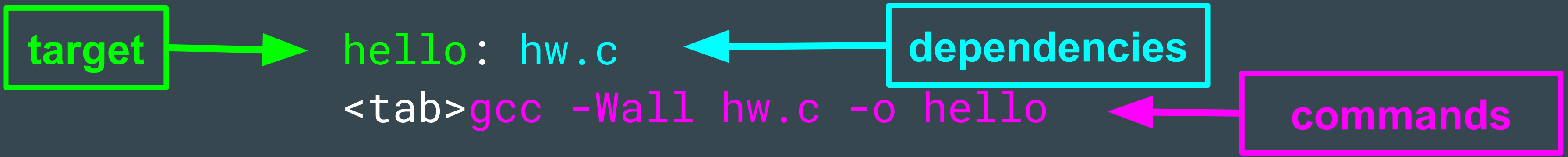
autoType1.c

```
...

Move get_auto_move(
        Board* b, Dim* r, Dim* c)
{
    // logic for type 1

...
```

Basic anatomy of a makefile: Rules

Makefiles are plain text files called “`makefile`” or “`Makefile`”.

They are made up of basic building blocks known as “Rules”



target

```
hello: hw.c
<tab>gcc -Wall hw.c -o hello
```

dependencies

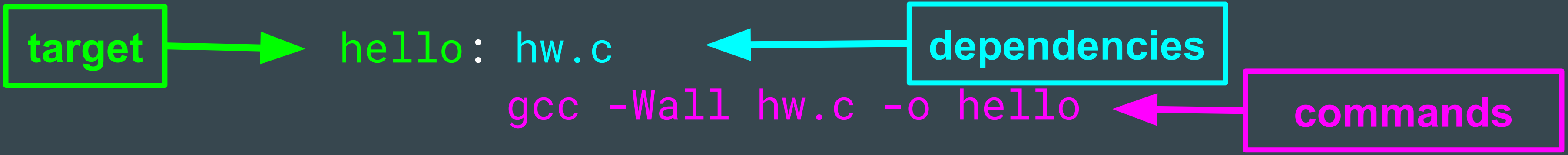
commands

The **Target** is the name of the rule. Often named for what the rule creates.

Dependencies are the required input files to the rule. A rule is run if dependencies are newer than target

Commands will be run when the target is triggered. Can have multiple per rule. Must be indented by a single **tab** character.

Basic anatomy of a makefile: Running a rule



target

dependencies

commands

```
hello: hw.c
	gcc -Wall hw.c -o hello
```

To compile `hw.c` to `hello`, we use the desired target as a command line argument:

```
$ make hello
```

Alternatively running `make` with no arguments will run the default rule if it exists. If not, it will run the first rule in the makefile.

Default Targets, Phony Goals and Comments

When `make` is run without any target arguments, it will run the first rule by default. You can override this using `.DEFAULT_GOAL`.

Phony targets are targets which have *no associated output file*. Here, **`all`** doesn't create any output files, instead it runs its set of dependencies which perform the needed tasks instead.

Due to this we need to mark it as 'phony' to prevent 'Nothing to do' errors.

Comments can be added using #

```
.PHONY: all
.DEFAULT_GOAL := all

all: hello

# build hello from hw.c
hello: hw.c
	gcc -Wall hw.c -o hello
```

Variables

Variables are awesome! Use them! They reduce duplication.

You can create a variable using the `<NAME>=<Value>` syntax

Variables are accessed using $()

We can use targets to modify variables. Here, the `careful` rule is used to change the compile flags variable in order to to convert warnings to errors.

```
CFLAGS= -Wall -pedantic
.PHONY: all clean careful
.DEFAULT_GOAL := all

all: hello


careful: CFLAGS += -Werror
careful: all


# build hello from hw.c
hello: hw.c
        gcc $(CFLAGS) hw.c -o hello
clean:
        rm hello
```

Some Best Practices

Use variables to reduce duplication, especially for compiler flags.

Have a clean rule to remove executables and .o files

Set a DEFAULT_GOAL and all rule to help control makefile execution

A debug rule is also handy

```
CFLAGS= -Wall -pedantic
.PHONY: all clean careful
.DEFAULT_GOAL := all

all: hello

careful: CFLAGS += -Werror
careful: all

# build hello from hw.c
hello: hw.c
	gcc $(CFLAGS) hw.c -o hello
clean:
	rm hello
```

Compile and link separately

Particularly when you have many source files, you don’t want to compile all of them every time.

Separate out compiling each source file by creating rules for each source and object file pair.

We then have a single rule per program to link the object files together. Note the dependencies.

```
CFLAGS = -Wall
.PHONY: all clean careful
.DEFAULT_GOAL := all

all: hello
careful: CFLAGS += -Werror
careful: all

# link hello from object files
hello: hw.o
        gcc hw.o -o hello

# compile hw.c to an object file
hw.o: hw.c
        gcc $(CFLAGS) -c hw.c
clean:
        rm hello hw.o
```

Multiple source files

To add a second source file, we add the object files to the link rule (as a dependency and into the command).

Add another rule for the second source file.

Remember to update and check clean rule.

```
CFLAGS = -Wall
.PHONY: all careful clean
.DEFAULT_GOAL := all

all: hello
careful: CFLAGS += -Werror
careful: all

# link hello from object files
hello: hw.o hw2.o
	gcc hw.o hw2.o -o hello
hw.o: hw.c
	gcc $(CFLAGS) -c hw.c
# compile second source file
hw2.o: hw2.c
	gcc $(CFLAGS) -c hw2.c
clean:
	rm hello hw.o hw2.o
```

Now you try

Supplied files: `~s4436755/public/csse2310/pracs/makefiles/make.ex.tar.gz`

Ex1: (Covered in Tutorials)

1. Write a Makefile to build "`hi`" from hw.c
2. Modify the Makefile to compile and link separately.
3. Add a "`clean`" target.
4. Modify the Makefile to use a variable to specify compile flags and then build with debug.

Ex2: (Practice for your assignment)

1. Write a Makefile to build "`thromborax`" from the supplied files, be sure to build each .o separately

Modularity:

1. makefiletute.pdf on blackboard
2. linker.pdf on blackboard
3. function_pointers.pdf on blackboard

More resources

Long-time 2310 tutor’s website:
https://uni.joeladdison.com/csse2310/programming/makefiles

Official manual for GNU make:
https://www.gnu.org/software/make/manual/make.html