Separate Compilation

CS 211

Road map

Problem: Sharing stuff between files

Road map

Problem: Sharing stuff between files

The C compilation model

- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

Separate Compilation

Road map

Problem: Sharing stuff between files

The C compilation model

- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

The solution

Separate Compilation

Road map

Problem: Sharing stuff between files

The C compilation model

- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

The solution

Building with `make`

- `make` rules
- `make` in action

Initial code setup

The code in this course is available in your Unix shell account. You can get your own copy like this:

```
% cd cs211
% tar -xvkf ~cs211/lec/03_separate.tgz
⋮
% cd 03_separate
```

Up next

Problem: Sharing stuff between files

The C compilation model

- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

The solution

Building with `make`

- `make` rules
- `make` in action

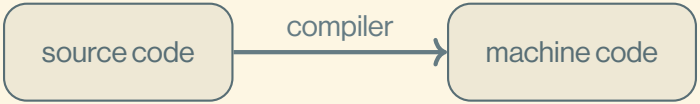
Reminder: Compilation

Your computer doesn't understand C. To run a program, you need to translate it from

- source code (human readable, *e.g.,* C or Swift)

to

- machine code (machine executable, *e.g.,* x86-64 or ARM).



The general problem

Two issues with compilation:

- Big programs take a long time to compile
- How can we reuse our functions in multiple programs?

The general problem

Two issues with compilation:

- Big programs take a long time to compile
- How can we reuse our functions in multiple programs?

Let's focus on the second issue. It would be really nice if we could:

1. Write some functions in one place.
2. Call those functions from multiple programs.

A more specific problem for today

We want to:

1. Write some functions in one place
2. Write a program that uses those functions
3. Write tests that ensure those functions are correct

A more specific problem for today

We want to:

1. Write some functions in one place
2. Write a program that uses those functions
3. Write tests that ensure those functions are correct

But: C has no facilities for testing. A test is just an ordinary program that calls some functions and checks their results.

So our goal is still one library with two (or more) clients.

A more specific problem for today

We want to:

1. Write some functions in one place: a library
2. Write a program that uses those functions: a client
3. Write tests that ensure those functions are correct: *another* client

But: C has no facilities for testing. A test is just an ordinary program that calls some functions and checks their results.

So our goal is still one library with two (or more) clients.

Making it concrete

- The code we want to share defines a `struct posn` type and three functions, `read_posn()`, `make_posn()`, and `manhattan_dist()`. (Call this the *posn* library.)

Making it concrete

- The code we want to share defines a `struct posn` type and three functions, `read_posn()`, `make_posn()`, and `manhattan_dist()`. (Call this the *`posn`* library.)
- Client 1, the *`interact`* program, uses the *`posn`* library to read positions from the standard input, calculate distances, and print the distances to the standard output.

Making it concrete

- The code we want to share defines a `struct posn` type and three functions, `read_posn()`, `make_posn()`, and `manhattan_dist()`. (Call this the *`posn`* library.)
- Client 1, the *`interact`* program, uses the *`posn`* library to read positions from the standard input, calculate distances, and print the distances to the standard output.
- Client 2, the *`posn_test`* test program, checks that the *`posn`* library's `manhattan_dist()` function gives the answers we expect.

What the *posn* library provides (highlights)

```c
// A 2-D point.
struct posn
{
    double x;
    double y;
};

// Computes the Manhattan distance between two posns.
double manhattan_dist(struct posn p, struct posn q)
{
    return fabs(p.x - q.x) + fabs(p.y - q.y);
}
```

The `interact` program

```c
// import posn library somehow?
#include <stdio.h>

int main(void)
{
    struct posn target = read_posn();

    for (;;) {
        struct posn each = read_posn();
        double dist = manhattan_dist(target, each);
        printf("%f\n", dist);
    }
}
```

The `posn_test` test program

```
// import posn library somehow?
#include <assert.h>

int main(void)
{
    struct posn p = make_posn(0, 0);
    struct posn q = make_posn(3, 4);

    assert( manhattan_dist(p, p) == 0 );
    assert( manhattan_dist(q, p) == 7 );
}
```

(The `assert()` macro does nothing if its argument is `true`, but if its argument is false then it crashes the program and prints a message. We'll have nicer ways to write tests in the future, but this week we'll stick with `assert`.)

Up next

Problem: Sharing stuff between files

The C compilation model

- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

The solution

Building with `make`

- `make` rules
- `make` in action

The C compilation model

2. Each `.c` file is its own compilation unit, translated into its own `.o` file

The C compilation model

2. Each `.c` file is its own compilation unit, translated into its own `.o` file
1. Before translating source code, the C preprocessor expands `#include`s (among other things)

The C compilation model

2. Each `.c` file is its own compilation unit, translated into its own `.o` file
1. Before translating source code, the C preprocessor expands `#include`s (among other things)
3. After translation, the linker combines multiple `.o` files with one `main()` function into an executable

Up next

- Problem: Sharing stuff between files
- The C compilation model
  - Step 2: Translation
  - Step 1: Preprocessing
  - Step 3: Linking
- The solution
- Building with `make`
  - `make` rules
  - `make` in action

The C compilation model: Translation

Each `.c` source file is its own compilation unit:

- Each is translated in isolation to an object (`.o`) file containing machine code
- The compiler only knows about one `.c` file at a time
- To translate a function call, the compiler only needs to know the function's signature, not its full definition

(Pass `cc` the `-c` flag to ask it to stop after translation, before linking.)

Declarations, definitions, and signatures

	Declaration	Definition
function	`int sqri(int);`	`int sqri(int i)` `{ return i * i; }`
structure	`struct posn;`	`struct posn` `{ double x, y; };`
variable	`extern short s;`	`short s;` or: `short s = 5;`

(A function declaration is the function's signature.)

Scope: C can only see backward

C compiler is happy:

src/min3.c

```c
double min2(double x, double y)
{
    return x < y ? x : y;
}

double min3(double x, double y, double z)
{
    return min2(x, min2(y, z));
}
```

Scope: C can only see backward

C compiler is unhappy, says that `min2` isn't declared:

src/min3.c

```c
double min3(double x, double y, double z)
{
    return min2(x, min2(y, z));
}

double min2(double x, double y)
{
    return x < y ? x : y;
}
```

Scope: C can only see backward

C compiler is happy once again:

src/min3.c

```c
double min2(double, double);

double min3(double x, double y, double z)
{
    return min2(x, min2(y, z));
}

double min2(double x, double y)
{
    return x < y ? x : y;
}
```

Up next

- Problem: Sharing stuff between files
- The C compilation model
  - Step 2: Translation
  - Step 1: Preprocessing
  - Step 3: Linking
- The solution
- Building with `make`
  - `make` rules
  - `make` in action

The C compilation model: Preprocessing

Before translating source code, the C compiler runs the C preprocessor:

- Replaces each `#include` with the file's contents
- Replaces `#define`d macros with their definitions

(You can pass `cc` the `-E` flag to ask it to stop after preprocessing.)

A preprocessing example

library.h

```c
int sqri(int);
```

client.c

```c
#include "library.h"
int main(void)
{ return sqri(5); }
```

A preprocessing example

library.h

```
int sqri(int);
```

client.c

```
#include "library.h"
int main(void)
{ return sqri(5); }
```

```
%
```

A preprocessing example

library.h

```
int sqri(int);
```

client.c

```
#include "library.h"
int main(void)
{ return sqri(5); }
```

```
% cc -E client.c
```

A preprocessing example

library.h

```
int sqri(int);
```

client.c

```
#include "library.h"
int main(void)
{ return sqri(5); }
```

```
% cc -E client.c
# 1 "client.c"
# 1 "./library.h" 1
int sqri(int);
# 2 "client.c" 2
int main(void)
{ return sqri(5); }
%
```

Up next

- Problem: Sharing stuff between files
- The C compilation model
  - Step 2: Translation
  - Step 1: Preprocessing
  - Step 3: Linking
- The solution
- Building with `make`
  - `make` rules
  - `make` in action

The C compilation model: Linking

The linker combines multiple `.o` files into an executable:

- Resolves references between files—every function that is used must be defined somewhere
- One Definition Rule (ODR): You can't have multiple definitions of the same function
- There must be exactly one definition of `main()`

Use `cc` to link (or not)

```
% cc client.o
```

Use `cc` to link (or not)

```
% cc client.o
client.o: In function 'main':
client.c:(.text+0xa): undefined reference to 'sqri'
collect2: error: ld returned 1 exit status
[1]%
```

Use `cc` to link (or not)

```
% cc client.o
client.o: In function 'main':
client.c:(.text+0xa): undefined reference to 'sqri'
collect2: error: ld returned 1 exit status
[1]% cc library.o
```

Use `cc` to link (or not)

```
% cc client.o
client.o: In function `main':
client.c:(.text+0xa): undefined reference to `sqri'
collect2: error: ld returned 1 exit status
[1]% cc library.o
/usr/lib/../lib64/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
[1]%
```

Use `cc` to link (or not)

```
% cc client.o
client.o: In function `main':
client.c:(.text+0xa): undefined reference to `sqri'
collect2: error: ld returned 1 exit status
[1]% cc library.o
/usr/lib/../lib64/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
[1]% cc library.o client.o src/min3.o
```

Use `cc` to link (or not)

```
% cc client.o
client.o: In function `main':
client.c:(.text+0xa): undefined reference to `sqri'
collect2: error: ld returned 1 exit status
[1]% cc library.o
/usr/lib/../lib64/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
[1]% cc library.o client.o src/min3.o
src/min3.o: In function `main':
min3.c:(.text+0x6c): multiple definition of `main'
client.o:client.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
[1]% cc library.o src/min3.o
```

Use `cc` to link (or not)

```
% cc client.o
client.o: In function `main':
client.c:(.text+0xa): undefined reference to `sqri'
collect2: error: ld returned 1 exit status
[1]% cc library.o
/usr/lib/../lib64/crt1.o: In function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
[1]% cc library.o client.o src/min3.o
src/min3.o: In function `main':
min3.c:(.text+0x6c): multiple definition of `main'
client.o:client.c:(.text+0x0): first defined here
collect2: error: ld returned 1 exit status
[1]% cc library.o src/min3.o
%
```

Use *nm* to inspect `.o` files

```
% nm library.o
```

Use *nm* to inspect `.o` files

```
% nm library.o
0000000000000000 T sqri
%
```

Use *nm* to inspect `.o` files

```
% nm library.o
0000000000000000 T sqri
% nm client.o
```

Use *nm* to inspect `.o` files

```
% nm library.o
0000000000000000 T sqri
% nm client.o
0000000000000000 T main
                 U sqri
%
```

Use *nm* to inspect `.o` files

```
% nm library.o
0000000000000000 T sqri
% nm client.o
0000000000000000 T main
                 U sqri
% nm src/min3.o
```

Use *nm* to inspect `.o` files

```
% nm library.o
0000000000000000 T sqri
% nm client.o
0000000000000000 T main
                 U sqri
% nm src/min3.o
                 U __isoc99_scanf
000000000000006c T main
0000000000000000 T min2
0000000000000028 T min3
                 U printf
%
```

Up next

Problem: Sharing stuff between files

The C compilation model

- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

The solution

Building with `make`

- `make` rules
- `make` in action

Using the C compilation model

The general recipe:

1. Put implementations of functions in `.c` files

Using the C compilation model

The general recipe:

1. Put implementations of functions in `.c` files
2. A `.c` file cannot both define `main()` and define functions called from other files! So don't put these together.

Using the C compilation model

The general recipe:

1. Put implementations of functions in `.c` files
2. A `.c` file cannot both define `main()` and define functions called from other files! So don't put these together.
3. For each `.c` file that doesn't define `main()`, create a matching `.h` file describing the interface of the `.c` file, namely `struct` definitions and function signatures

Using the C compilation model

The general recipe:

1. Put implementations of functions in `.c` files
2. A `.c` file cannot both define `main()` and define functions called from other files! So don't put these together.
3. For each `.c` file that doesn't define `main()`, create a matching `.h` file describing the interface of the `.c` file, namely `struct` definitions and function signatures
4. Each `.c` file that wants to call functions from another `.c` file must `#include` the corresponding `.h` file (otherwise it won't know about them)

(Even more fiddly details)

- Every `.h` file should start with a guard,

  ```
  #pragma once
  ```

  to prevent it from being included more than once
- Use `#include <…>` for system headers and `#include "…"` for user headers (yours)
- Never `#include` a `.c` file. Ever.

The solution, applied

- `src/posn.h` contains
  - Definition of `struct posn` type
  - Signatures for shared functions (`read_posn()`, `make_posn()`, and `manhattan_dist()`)

The solution, applied

- `src/posn.h` contains
  - Definition of `struct posn` type
  - Signatures for shared functions (`read_posn()`, `make_posn()`, and `manhattan_dist()`)
- `src/posn.c` `#include`s `src/posn.h` and contains definitions of the same shared functions

The solution, applied

- `src/posn.h` contains
  - Definition of `struct posn` type
  - Signatures for shared functions (`read_posn()`, `make_posn()`, and `manhattan_dist()`)
- `src/posn.c` `#include`s `src/posn.h` and contains definitions of the same shared functions
- `src/interact.c` `#include`s `src/posn.h` and contains the `main()` function for the `interact` program

The solution, applied

- `src/posn.h` contains
  - Definition of `struct posn` type
  - Signatures for shared functions (`read_posn()`, `make_posn()`, and `manhattan_dist()`)
- `src/posn.c` `#include`s `src/posn.h` and contains definitions of the same shared functions
- `src/interact.c` `#include`s `src/posn.h` and contains the `main()` function for the `interact` program
- `test/posn_test.c` `#include`s `src/posn.h` and contains a `main()` function that tests the functions defined in `src/posn.c`

The solution, applied

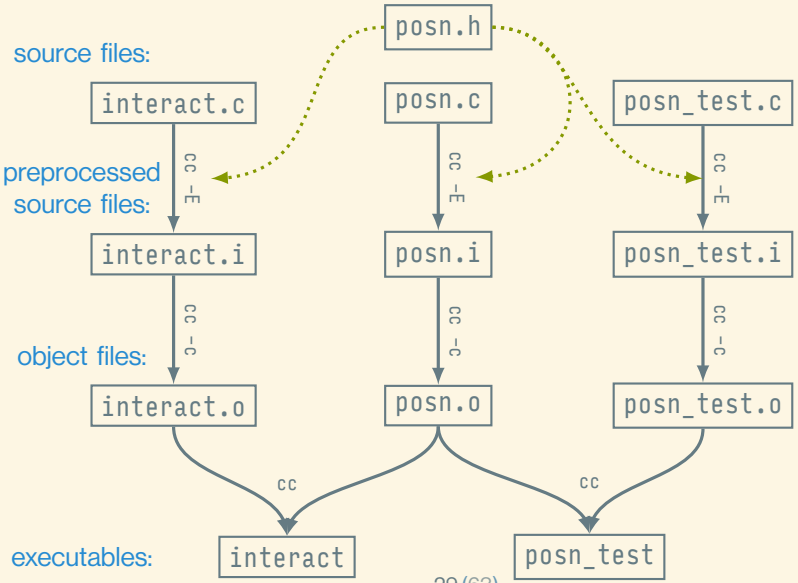
- `src/posn.h` contains
  - Definition of `struct posn` type
  - Signatures for shared functions (`read_posn()`, `make_posn()`, and `manhattan_dist()`)
- `src/posn.c` `#include`s `src/posn.h` and contains definitions of the same shared functions
- `src/interact.c` `#include`s `src/posn.h` and contains the `main()` function for the `interact` program
- `test/posn_test.c` `#include`s `src/posn.h` and contains a `main()` function that tests the functions defined in `src/posn.c`

The solution, applied

- `src/posn.h` contains
  - Definition of `struct posn` type
  - Signatures for shared functions (`read_posn()`, `make_posn()`, and `manhattan_dist()`)
- `src/posn.c` `#include`s `src/posn.h` and contains definitions of the same shared functions
- `src/interact.c` `#include`s `src/posn.h` and contains the `main()` function for the `interact` program
- `test/posn_test.c` `#include`s `src/posn.h` and contains a `main()` function that tests the functions defined in `src/posn.c`

(ODR: You cannot have more than one definition of the same symbol (variable, constant, or function) in the same program. This means that attempting to link `interact.o` and `posn_test.o` together will result in an error.)

Build dependencies



Up next

Separate Compilation

Problem: Sharing stuff between files

The C compilation model

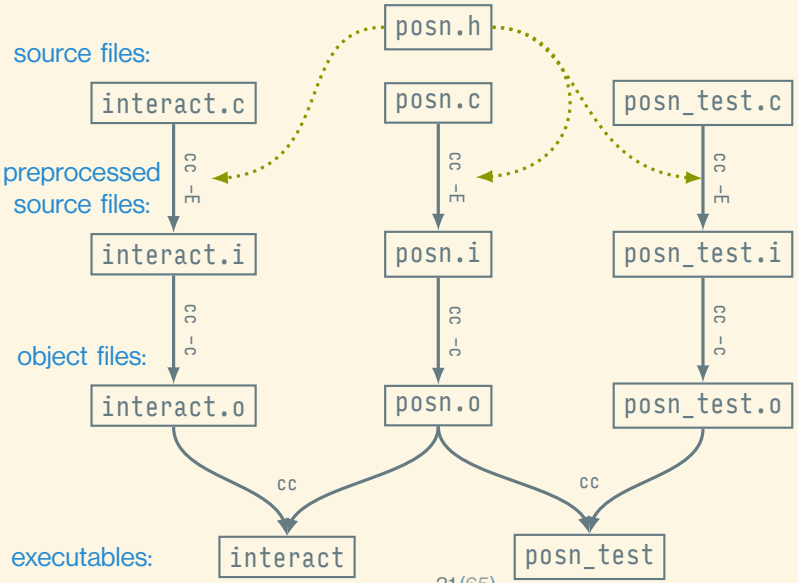
- Step 2: Translation
- Step 1: Preprocessing
- Step 3: Linking

The solution

Building with `make`

- `make` rules
- `make` in action

Build dependencies



This is complicated! Let's use `make`

`make`(1) is a tool for building programs out of multiple source files.

This is complicated! Let's use `make`

`make`(1) is a tool for building programs out of multiple source files.

To build a file named ⟨*goal*⟩ using `make`, you just run

```
% make ⟨goal⟩
```

(don't type the %)

This is complicated! Let's use `make`

`make`(1) is a tool for building programs out of multiple source files.

To build a file named ⟨*goal*⟩ using `make`, you just run

```
% make ⟨goal⟩
```
(don't type the %)

Then `make` looks in the current directory for a file named `Makefile` to find a rule for building ⟨*goal*⟩.

Up next

- Problem: Sharing stuff between files
- The C compilation model
  - Step 2: Translation
  - Step 1: Preprocessing
  - Step 3: Linking
- The solution
- Building with `make`
  - `make` rules
  - `make` in action

What does a `make` rule look like?

A rule tells `make` how to build a particular goal file from some prerequisite file(s) by running some shell commands:

⟨*goal*⟩ **:** ⟨*prereqs*⟩ . . .
⟨*commands*⟩
⋮

What does a `make` rule look like?

A rule tells `make` how to build a particular goal file from some prerequisite file(s) by running some shell commands:

⟨*goal*⟩`:` ⟨*prereqs*⟩...
⟨*commands*⟩
⋮

For example, here's a rule that specifies how to build `hello` from `hello.c` by running the command `cc -o hello hello.c`:

```
hello: hello.c
	cc -o hello hello.c
```

A `Makefile` for building `interact` and `posn_test`

These rules encode the dependency diagram from a few slides back (but with preprocessing and translation done together):

```
interact: interact.o posn.o
        cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
        cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
        cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
        cc -c -o posn.o posn.c
```

A `Makefile` for building `interact` and `posn_test`

Good progammers are lazy and hate repetition. So much repetition here!

```
interact: interact.o posn.o
        cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
        cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
        cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
        cc -c -o posn.o posn.c
```

A `Makefile` for building *`interact`* and *`posn_test`*

You don't have to repeat the goal in the recipe; it's better use the special variable `$@` instead...

```
interact: interact.o posn.o
        cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
        cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
        cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
        cc -c -o posn.o posn.c
```

A `Makefile` for building `interact` and `posn_test`

You don't have to repeat the goal in the recipe; it's better use the special variable `$@` instead:

```
interact: interact.o posn.o
        cc -o $@ interact.o posn.o

posn_test: posn_test.o posn.o
        cc -o $@ posn_test.o posn.o

interact.o: interact.c posn.h
        cc -c -o $@ interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o $@ posn_test.c

posn.o: posn.c posn.h
        cc -c -o $@ posn.c
```

A `Makefile` for building `interact` and `posn_test`

Similarly, use `$^` to stand for all the prerequisites...

```
interact: interact.o posn.o
        cc -o $@ interact.o posn.o

posn_test: posn_test.o posn.o
        cc -o $@ posn_test.o posn.o

interact.o: interact.c posn.h
        cc -c -o $@ interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o $@ posn_test.c

posn.o: posn.c posn.h
        cc -c -o $@ posn.c
```

A `Makefile` for building `interact` and `posn_test`

Similarly, use `$^` to stand for all the prerequisites:

```
interact: interact.o posn.o
        cc -o $@ $^

posn_test: posn_test.o posn.o
        cc -o $@ $^

interact.o: interact.c posn.h
        cc -c -o $@ interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o $@ posn_test.c

posn.o: posn.c posn.h
        cc -c -o $@ posn.c
```

A `Makefile` for building *`interact`* and *`posn_test`*

Similarly, use `$^` to stand for all the prerequisites, or use `$<` when you need just the first prerequisite...

```
interact: interact.o posn.o
        cc -o $@ $^

posn_test: posn_test.o posn.o
        cc -o $@ $^

interact.o: interact.c posn.h
        cc -c -o $@ interact.c

posn_test.o: posn_test.c posn.h
        cc -c -o $@ posn_test.c

posn.o: posn.c posn.h
        cc -c -o $@ posn.c
```

A `Makefile` for building `interact` and `posn_test`

Similarly, use `$^` to stand for all the prerequisites, or use `$<` when you need just the first prerequisite:

```
interact: interact.o posn.o
        cc -o $@ $^

posn_test: posn_test.o posn.o
        cc -o $@ $^

interact.o: interact.c posn.h
        cc -c -o $@ $<

posn_test.o: posn_test.c posn.h
        cc -c -o $@ $<

posn.o: posn.c posn.h
        cc -c -o $@ $<
```

A `Makefile` for building `interact` and `posn_test`

Now the three compilation rules are all *the same* except for the filename, so we can replace all three with a single pattern rule...

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

interact.o: interact.c posn.h
	cc -c -o $@ $<

posn_test.o: posn_test.c posn.h
	cc -c -o $@ $<

posn.o: posn.c posn.h
	cc -c -o $@ $<
```

A `Makefile` for building `interact` and `posn_test`

Now the three compilation rules are all *the same* except for the filename, so we can replace all three with a single pattern rule:

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c posn.h
	cc -c -o $@ $<
```

A `Makefile` for building `interact` and `posn_test`

Now the three compilation rules are all *the same* except for the filename, so we can replace all three with a single pattern rule:

```
interact: interact.o posn.o
        cc -o $@ $^

posn_test: posn_test.o posn.o
        cc -o $@ $^

%.o: %.c posn.h
        cc -c -o $@ $<
```

(This pattern rule says we can build *anything*`.o` from a matching `.c`, and each depends `posn.h` as well.)

A `Makefile` for building `interact` and `posn_test`

For *our* project every `.o` depends on `posn.h`, but this isn't true of *all* projects. Let's split out the particulars of our project from the general rule...

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c posn.h
	cc -c -o $@ $<
```

(This pattern rule says we can build *anything*`.o` from a matching `.c`, and each depends `posn.h` as well.)

A `Makefile` for building `interact` and `posn_test`

For *our* project every `.o` depends on `posn.h`, but this isn't true of *all* projects. Let's split out the particulars of our project from the general rule:

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c
	cc -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

A `Makefile` for building `interact` and `posn_test`

Now we're going to parameterize a little. It will be easier to use different C compilers if we refer to it via a variable `$(CC)` instead of the literal `cc`...

```
interact: interact.o posn.o
        cc -o $@ $^

posn_test: posn_test.o posn.o
        cc -o $@ $^

%.o: %.c
        cc -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

A `Makefile` for building `interact` and `posn_test`

Now we're going to parameterize a little. It will be easier to use different C compilers if we refer to it via a variable `$(CC)` instead of the literal `cc`:

```
interact: interact.o posn.o
	$(CC) -o $@ $^

posn_test: posn_test.o posn.o
	$(CC) -o $@ $^

%.o: %.c
	$(CC) -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

A `Makefile` for building `interact` and `posn_test`

And finally, usually there are options we want to pass to `cc`. We'll use the standard variables to hold the options for each...

```
interact: interact.o posn.o
        $(CC) -o $@ $^

posn_test: posn_test.o posn.o
        $(CC) -o $@ $^

%.o: %.c
        $(CC) -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

A `Makefile` for building `interact` and `posn_test`

And finally, usually there are options we want to pass to `cc`. We'll use the standard variables to hold the options for each:

```
interact: interact.o posn.o
	$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

posn_test: posn_test.o posn.o
	$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

%.o: %.c
	$(CC) -c -o $@ $< $(CPPFLAGS) $(CFLAGS)

interact.o posn_test.o posn.o: posn.h
```

Up next

- Problem: Sharing stuff between files
- The C compilation model
  - Step 2: Translation
  - Step 1: Preprocessing
  - Step 3: Linking
- The solution
- Building with `make`
  - `make` rules
  - `make` in action

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
%
```

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
% make interact
```

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
% make interact
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -c -o src/posn.o src/posn.c -std=c11 -pedantic -W…
cc -o interact src/interact.o src/posn.o -std=c11 -p…
%
```

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
% make interact
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -c -o src/posn.o src/posn.c -std=c11 -pedantic -W…
cc -o interact src/interact.o src/posn.o -std=c11 -p…
% make posn_test
```

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
% make interact
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -c -o src/posn.o src/posn.c -std=c11 -pedantic -W…
cc -o interact src/interact.o src/posn.o -std=c11 -p…
% make posn_test
cc -c -o test/posn_test.o test/posn_test.c -std=c11 …
cc -o posn_test test/posn_test.o src/posn.o -std=c11…
%
```

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
% make interact
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -c -o src/posn.o src/posn.c -std=c11 -pedantic -W…
cc -o interact src/interact.o src/posn.o -std=c11 -p…
% make posn_test
cc -c -o test/posn_test.o test/posn_test.c -std=c11 …
cc -o posn_test test/posn_test.o src/posn.o -std=c11…
% make posn_test
```

make understands dependencies

Notice that when we build *posn_test*, Make does not recompile `src/posn.c` to `src/posn.o`, because it already did that to build *interact*.

```
% make clean
rm -f client $(EXES) *.o */*.o
% make interact
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -c -o src/posn.o src/posn.c -std=c11 -pedantic -W…
cc -o interact src/interact.o src/posn.o -std=c11 -p…
% make posn_test
cc -c -o test/posn_test.o test/posn_test.c -std=c11 …
cc -o posn_test test/posn_test.o src/posn.o -std=c11…
% make posn_test
make: `posn_test' is up to date.
%
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

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

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
%
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
% make
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
% make
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -o src/interact src/interact.o src/posn.c -std=c1…
%
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
% make
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -o src/interact src/interact.o src/posn.c -std=c1…
% touch src/posn.h
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
% make
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -o src/interact src/interact.o src/posn.c -std=c1…
% touch src/posn.h
%
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
% make
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -o src/interact src/interact.o src/posn.c -std=c1…
% touch src/posn.h
% make
```

`make` performs minimal rebuilds

The `touch`(1) command updates a file's modification time. This lets us see how `make` deals with files changing:

```
% make
make: Nothing to be done for 'all'.
% touch src/interact.c
% make
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -o src/interact src/interact.o src/posn.c -std=c1…
% touch src/posn.h
% make
cc -c -o src/interact.o src/interact.c -std=c11 -ped…
cc -c -o src/posn.o src/posn.c -std=c11 -pedantic -W…
cc -o interact src/interact.o src/posn.c -std=c11 -p…
cc -c -o test/posn_test.o test/posn_test.c -std=c11 …
cc -o posn_test test/posn_test.o src/posn.o -std=c11…
```

– Next time: Arrays, pointers, and strings –