

# Separate Compilation

CS 211

Winter 2020

## Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl -k $URL211/lec/03separate.tgz | tar zxv
...
% cd 02typed_imp
```

# The general problem

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 need 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 need 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 built-in facilities for testing. Your tests are just an ordinary program that calls the functions and checks the results.

So the goal is the same: one library with two (or more) clients.

## Making it concrete

1. The `posn` library: provides the `struct posn` type and three functions, `read_posn()`, `make_posn()`, and `manhattan_dist()`.
2. 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.
3. Client 2, the `posn_test` test program: checks that the `posn` library's `manhattan_dist()` function gives the answers we expect.

## The posn library (highlights)

```
// 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

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

int main()
{
    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()
{
    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()` function crashes the program if its argument is `false`, or does nothing if its argument is `true`. We'll have nicer ways to write tests in the future, but right now we'll stick with `assert`.)

## The solution, generally

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

## The solution, generally

1. Put implementations of functions in `.c` files.
2. Describe the interface to each `.c` file (type definitions, function *signatures*) in a corresponding `.h` (header) file.

## The solution, generally

1. Put implementations of functions in `.c` files.
2. Describe the interface to each `.c` file (type definitions, function *signatures*) in a corresponding `.h` (header) file.
3. Each `.c` file that wants to call code from another `.c` file must *#include* the corresponding `.h` file.

## The solution, generally

1. Put implementations of functions in `.c` files.
2. Describe the interface to each `.c` file (type definitions, function *signatures*) in a corresponding `.h` (header) file.
3. Each `.c` file that wants to call code from another `.c` file must *#include* the corresponding `.h` file.
4. Each `.c` file is its own *compilation unit*, which means it is translated by the compiler in isolation, with no direct knowledge of the other `.c` files, into a `.o` (object) file containing machine code. All dependencies are via `.h` files that the `.c` file *#includes*.

## The solution, generally

1. Put implementations of functions in `.c` files.
2. Describe the interface to each `.c` file (type definitions, function *signatures*) in a corresponding `.h` (header) file.
3. Each `.c` file that wants to call code from another `.c` file must *#include* the corresponding `.h` file.
4. Each `.c` file is its own *compilation unit*, which means it is translated by the compiler in isolation, with no direct knowledge of the other `.c` files, into a `.o` (object) file containing machine code. All dependencies are via `.h` files that the `.c` file *#includes*.
5. Once all the `.c` files for a program have been translated into `.o` files, the *linker* combines them into a single executable, resolving the references between them.

## (And the fiddly details)

- When translating individual source files, pass `cc` the `-c` switch to suppress linking.
- Every `.h` file should start with a *guard*,  
*`#pragma once`*  
to prevent processing it more than once per compilation unit.
- Never *`#include`* a `.c` file. Ever.

## Why this works

The C compiler is pretty stupid:

- Remembers nothing from one .c file to the next
- Reads strictly downward (so it doesn't know about things at the bottom of a file when it's processing the top of that file)



## Why this works

The C compiler is pretty stupid:

- Remembers nothing from one .c file to the next
- Reads strictly downward (so it doesn't know about things at the bottom of a file when it's processing the top of that file)

But:

- To compile a function call, it only needs to know the signature (type) of the function, not its whole definition.
- *A function declaration* specifies a function signature without the definition, like so:

```
double manhattan_dist(struct posn, struct posn);
```

(The parameter names are optional, so it makes sense to omit them from signatures when they aren't informative.)

## Example of C scope

C compiler is happy:

```
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));
}
```

## Example of C scope

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

```
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;
}
```

## Example of C scope

C compiler is happy once again:

```
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;
}
```

## 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` *#includes* `src/posn.h` and contains definitions of the same shared functions
- `src/interact.c` *#includes* `src/posn.h` and contains the `main` function for the `interact` program
- `test/posn_test.c` *#includes* `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` `#includes` `src/posn.h` and contains definitions of the same shared functions
- `src/interact.c` `#includes` `src/posn.h` and contains the `main` function for the `interact` program
- `test/posn_test.c` `#includes` `src/posn.h` and contains a `main` function that tests the functions defined in `src/posn.c`.

Important C rule: 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

header files:

posn.h

source files:

interact.c

posn.c

posn\_test.c

object files:

interact.o

posn.o

posn\_test.o

executable files:

interact

posn\_test

cc

cc

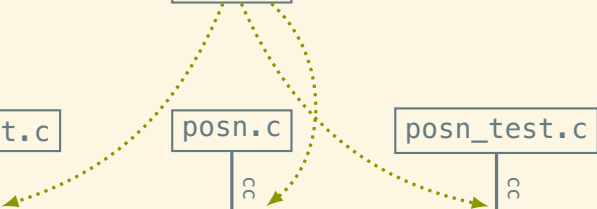
cc

cc

cc

cc

cc



## A bit more Make

To implement the previous slide in Make, we define *pattern rules* for particular types of files. Here's the rule for translating any .c file into a .o file:

```
build/%.o: src/%.c | build/  
    cc -c -o $@ $< $(CFLAGS)
```



## A bit more Make

To implement the previous slide in Make, we define *pattern rules* for particular types of files. Here's the rule for translating any `.c` file into a `.o` file:

```
build/%.o: src/%.c | build/  
    cc -c -o $@ $< $(CFLAGS)
```

We also need to let Make know which object files depend on which header files. These dependency specifications say that if `src/posn.h` changes then each of the three object files dependent on it needs to be rebuilt:

```
build/interact.o: src/posn.h  
build/posn.o: src/posn.h  
build/posn_test.o: src/posn.h
```

## Make understands dependencies

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

%

## Make understands dependencies

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

```
% make clean
```

## Make understands dependencies

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

```
% make clean  
rm -Rf build  
%
```

## Make understands dependencies

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

```
% make clean  
rm -Rf build  
% make build/interact
```

## Make understands dependencies

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

```
% make clean
rm -Rf build
% make build/interact
mkdir -p build
cc -c -o build/interact.o src/interact.c -std=c11...
cc -c -o build/posn.o src/posn.c -std=c11 -pedant...
cc -o build/interact build/interact.o build/posn...
%
```

## Make understands dependencies

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

```
% make clean
rm -Rf build
% make build/interact
mkdir -p build
cc -c -o build/interact.o src/interact.c -std=c11...
cc -c -o build/posn.o src/posn.c -std=c11 -pedant...
cc -o build/interact build/interact.o build/posn...
% make build/posn_test
```

## Make understands dependencies

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

```
% make clean
rm -Rf build
% make build/interact
mkdir -p build
cc -c -o build/interact.o src/interact.c -std=c11...
cc -c -o build/posn.o src/posn.c -std=c11 -pedant...
cc -o build/interact build/interact.o build/posn...
% make build/posn_test
cc -c -o build/posn_test.o test/posn_test.c -std=...
cc -o build/posn_test build/posn_test.o build/pos...
%
```



## Make understands dependencies

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

```
% make clean
rm -Rf build
% make build/interact
mkdir -p build
cc -c -o build/interact.o src/interact.c -std=c11...
cc -c -o build/posn.o src/posn.c -std=c11 -pedant...
cc -o build/interact build/interact.o build/posn...
% make build/posn_test
cc -c -o build/posn_test.o test/posn_test.c -std=...
cc -o build/posn_test build/posn_test.o build/pos...
% make build/posn_test
```

## Make understands dependencies

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

```
% make clean
rm -Rf build
% make build/interact
mkdir -p build
cc -c -o build/interact.o src/interact.c -std=c11...
cc -c -o build/posn.o src/posn.c -std=c11 -pedant...
cc -o build/interact build/interact.o build/posn...
% make build/posn_test
cc -c -o build/posn_test.o test/posn_test.c -std=...
cc -o build/posn_test build/posn_test.o build/pos...
% make build/posn_test
make: `build/posn_test' is up to date.
```

## Make performs minimal rebuilds

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

%

## Make performs minimal rebuilds

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

```
% make
```

## Make performs minimal rebuilds

The touch 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 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 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 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 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 build/interact.o src/interact.c -std=c11...
cc -o build/interact build/interact.o build/posn...
%
```

## Make performs minimal rebuilds

The touch 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 build/interact.o src/interact.c -std=c11...
cc -o build/interact build/interact.o build/posn...
% touch src/posn.h
```

## Make performs minimal rebuilds

The touch 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 build/interact.o src/interact.c -std=c11...
cc -o build/interact build/interact.o build/posn...
% touch src/posn.h
%
```

## Make performs minimal rebuilds

The touch 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 build/interact.o src/interact.c -std=c11...
cc -o build/interact build/interact.o build/posn...
% touch src/posn.h
% make
```

## Make performs minimal rebuilds

The touch 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 build/interact.o src/interact.c -std=c11...
cc -o build/interact build/interact.o build/posn...
% touch src/posn.h
% make
cc -c -o build/interact.o src/interact.c -std=c11...
cc -c -o build/posn.o src/posn.c -std=c11 -pedant...
cc -o build/interact build/interact.o build/posn...
cc -c -o build/posn_test.o test/posn_test.c -std=...
cc -o build/posn_test build/posn_test.o build/pos...
```