Week 1 Part II

Kyle Dewey

Overview

- Lab
- The art of programming
- Basic data types
- Variable declaration and initialization
- Data representation

Lab 1

The Art of Programming

General Advice

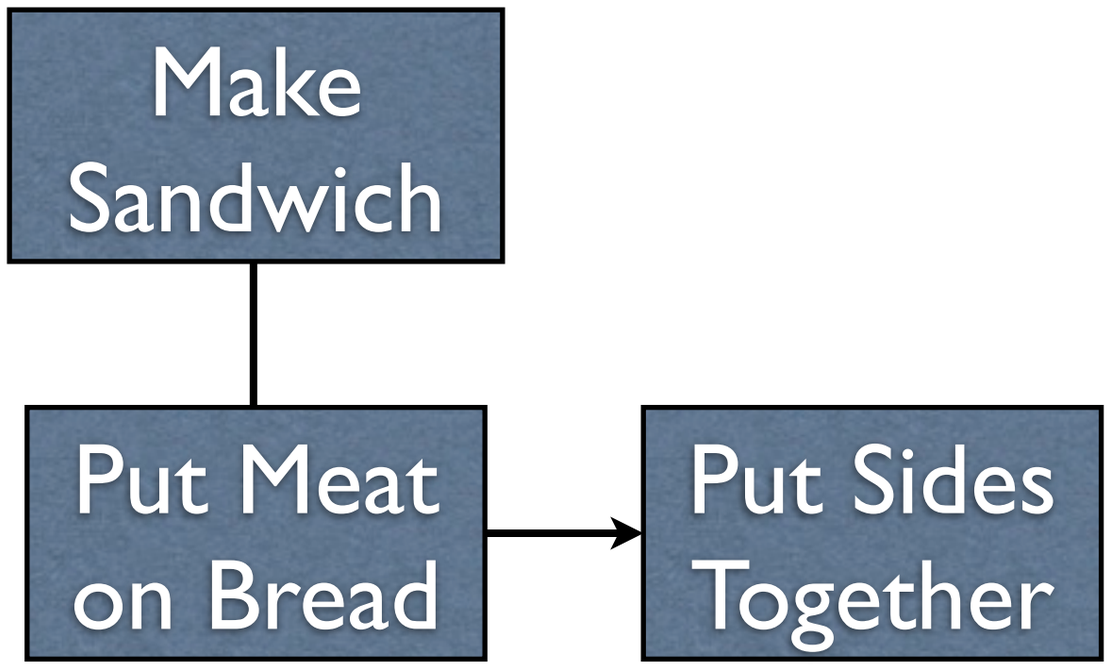
- Top-down programming: Start with the big picture and get more specific
- Bottom-up programming: work up to the big picture by glueing together smaller, existing pieces
- Divide and conquer: break a problem up into individual piecs

Top-Down Programming

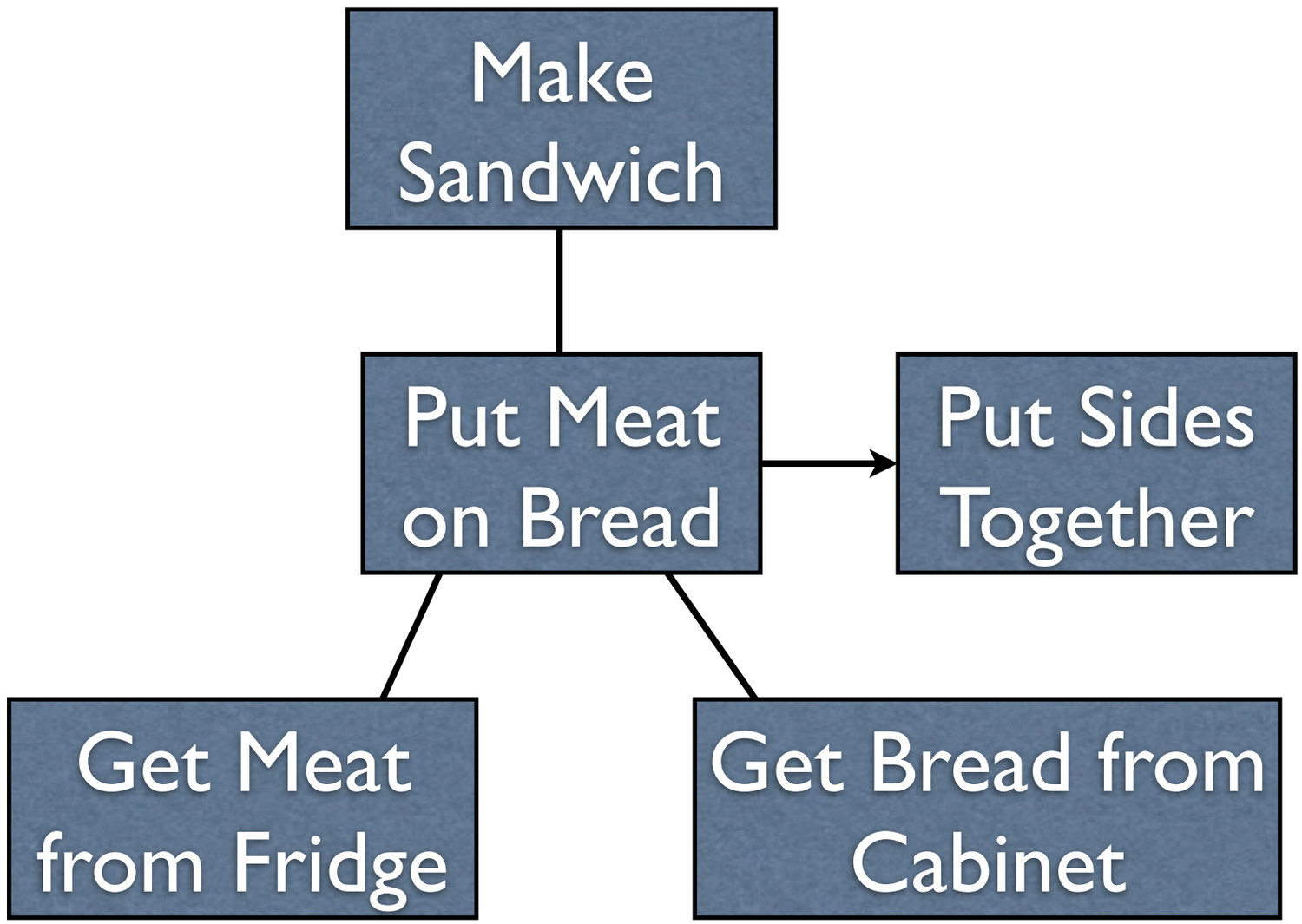
- Start general and gradually become specific, filling in the details as you go along

Top-Down

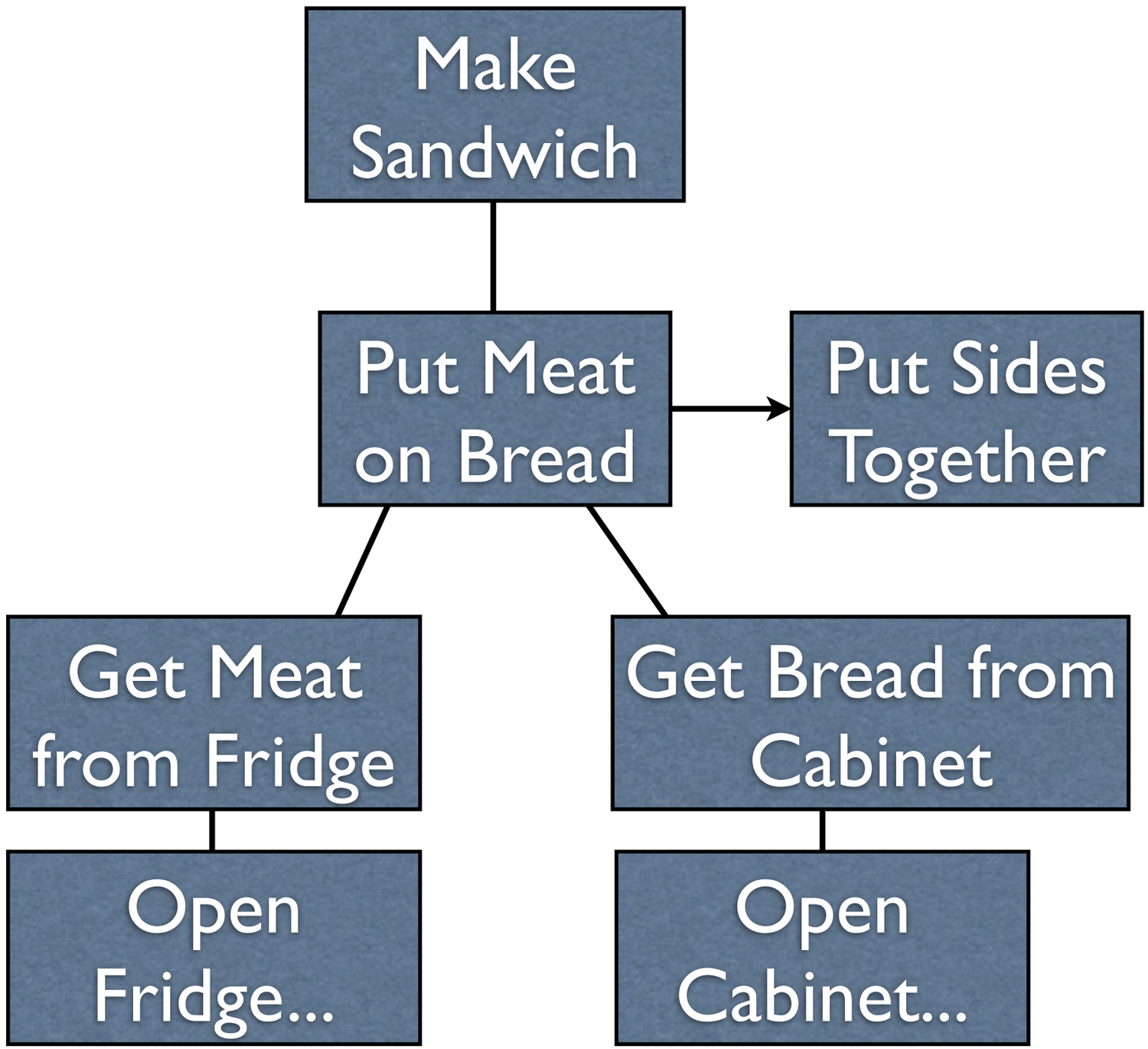
Top-Down



Top-Down


Make Sandwich
Put Meat on Bread
Put Sides Together
Get Meat from Fridge
Get Bread from Cabinet

Top-Down


Make Sandwich
Put Meat on Bread
Put Sides Together
Get Meat from Fridge
Get Bread from Cabinet
Open Fridge...
Open Cabinet...

Bottom-Up Programming

- Glue **preexisting** pieces together to derive a solution

Bottom-Up

Handlebars

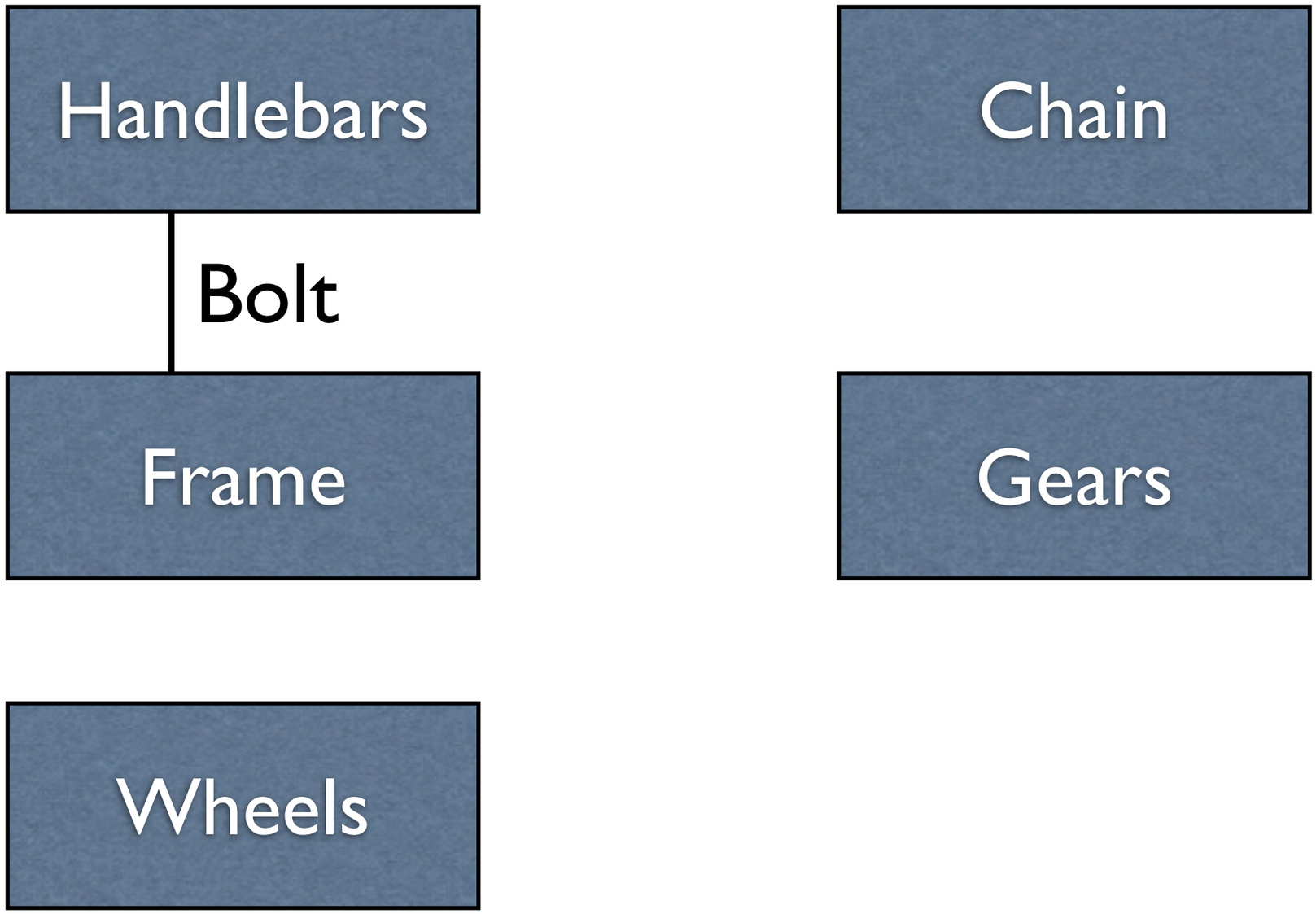
Chain

Frame

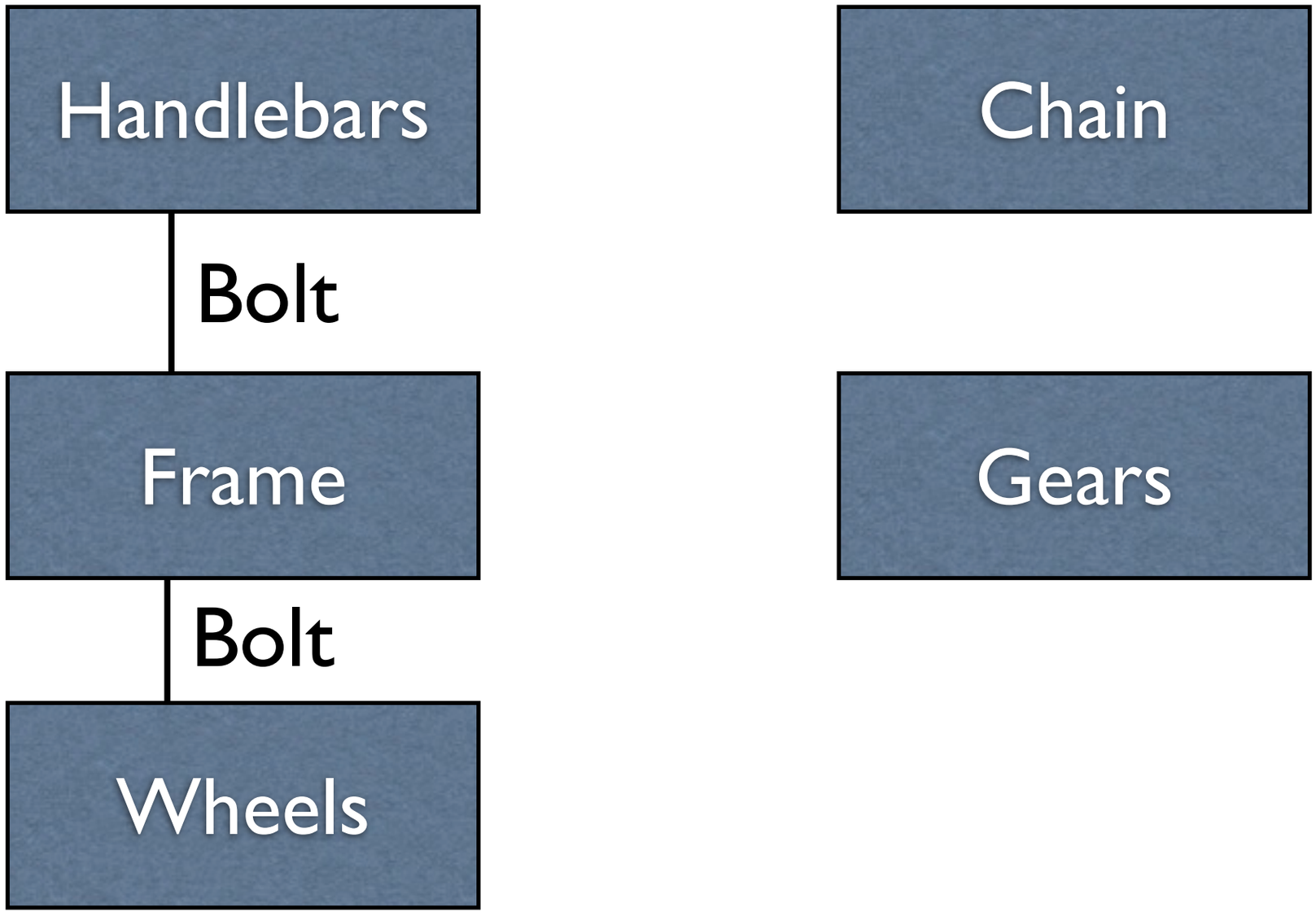
Gears

Wheels

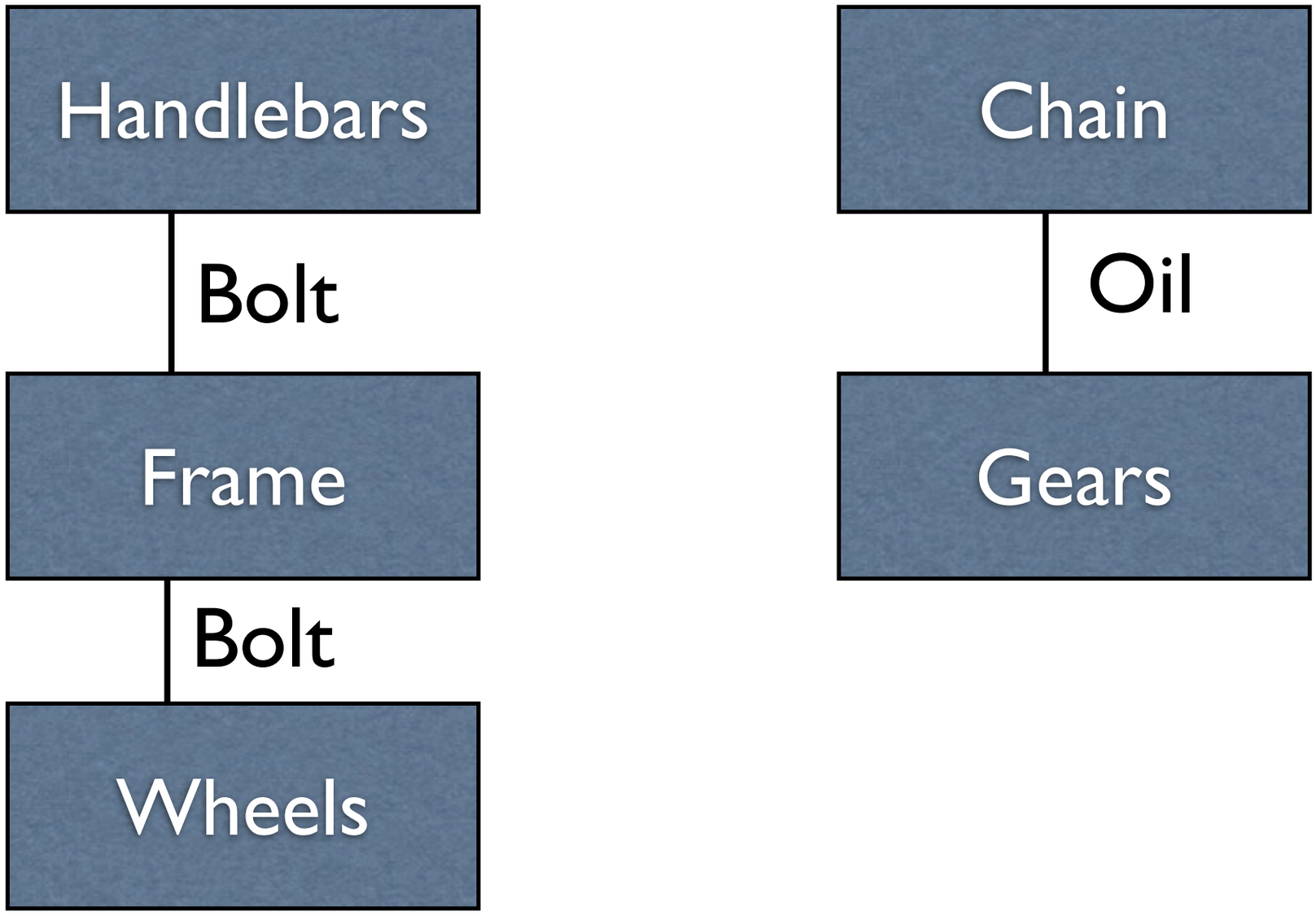
Bottom-Up



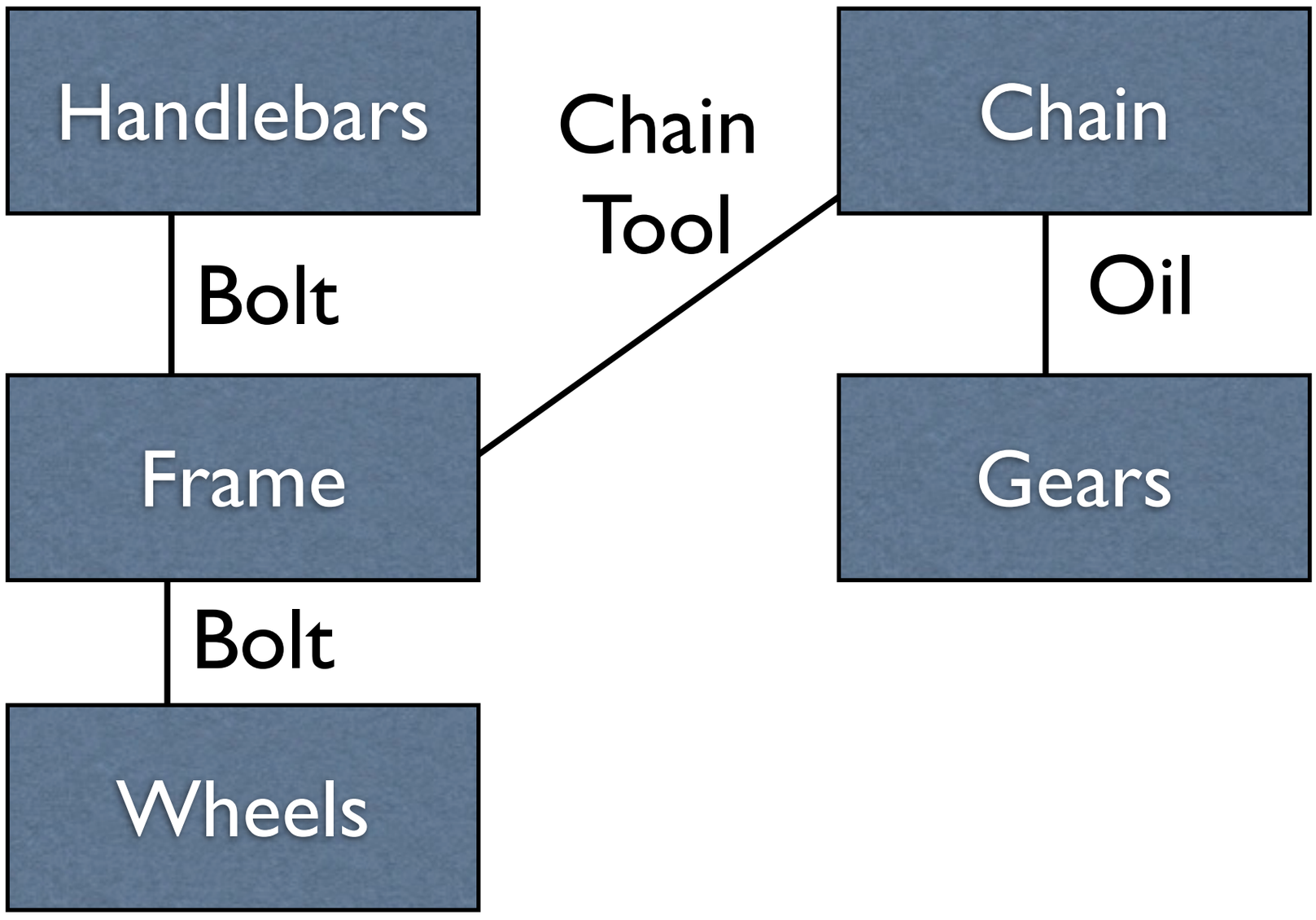
Bottom-Up



Bottom-Up



Bottom-Up


Handlebars
Chain Tool
Chain
Bolt
Oil
Frame
Gears
Bolt
Wheels

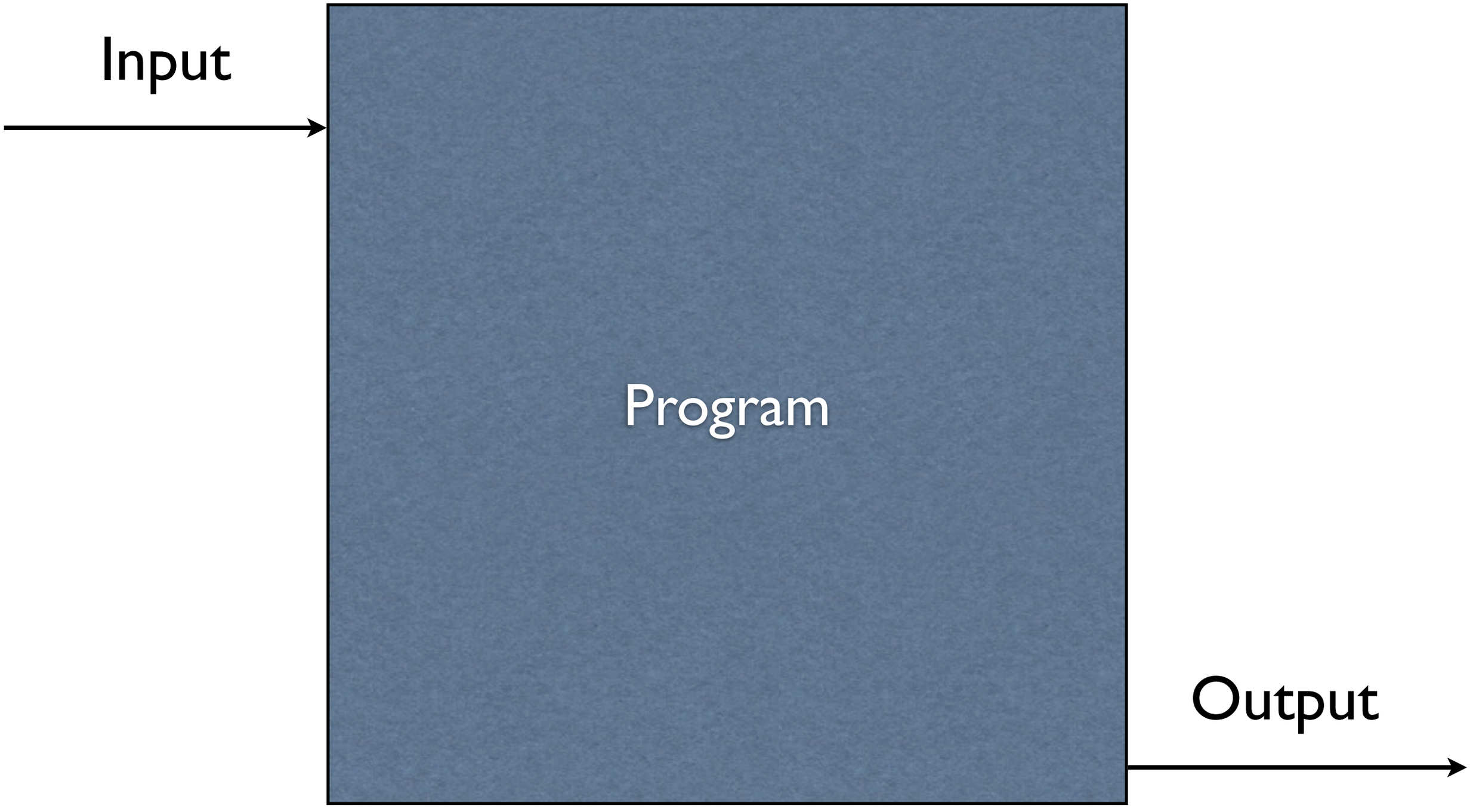
Combination

- `printf` / `scanf` are generally written once (preexisting)
- A middle ground

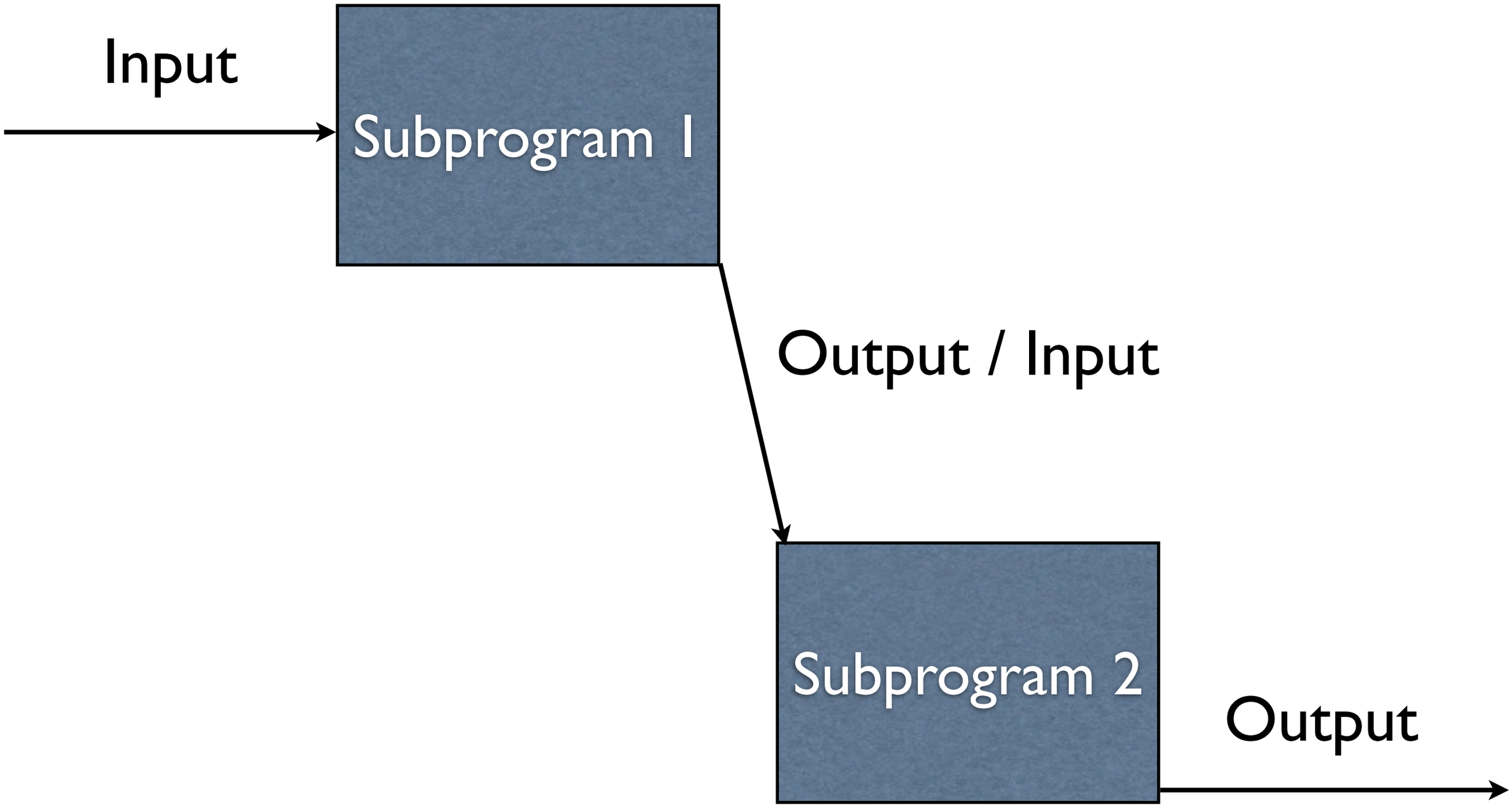
Divide and Conquer

- Really part of top-down and bottom-up
- Break a problem down into distinct subproblems, solve them individually, and finally combine them

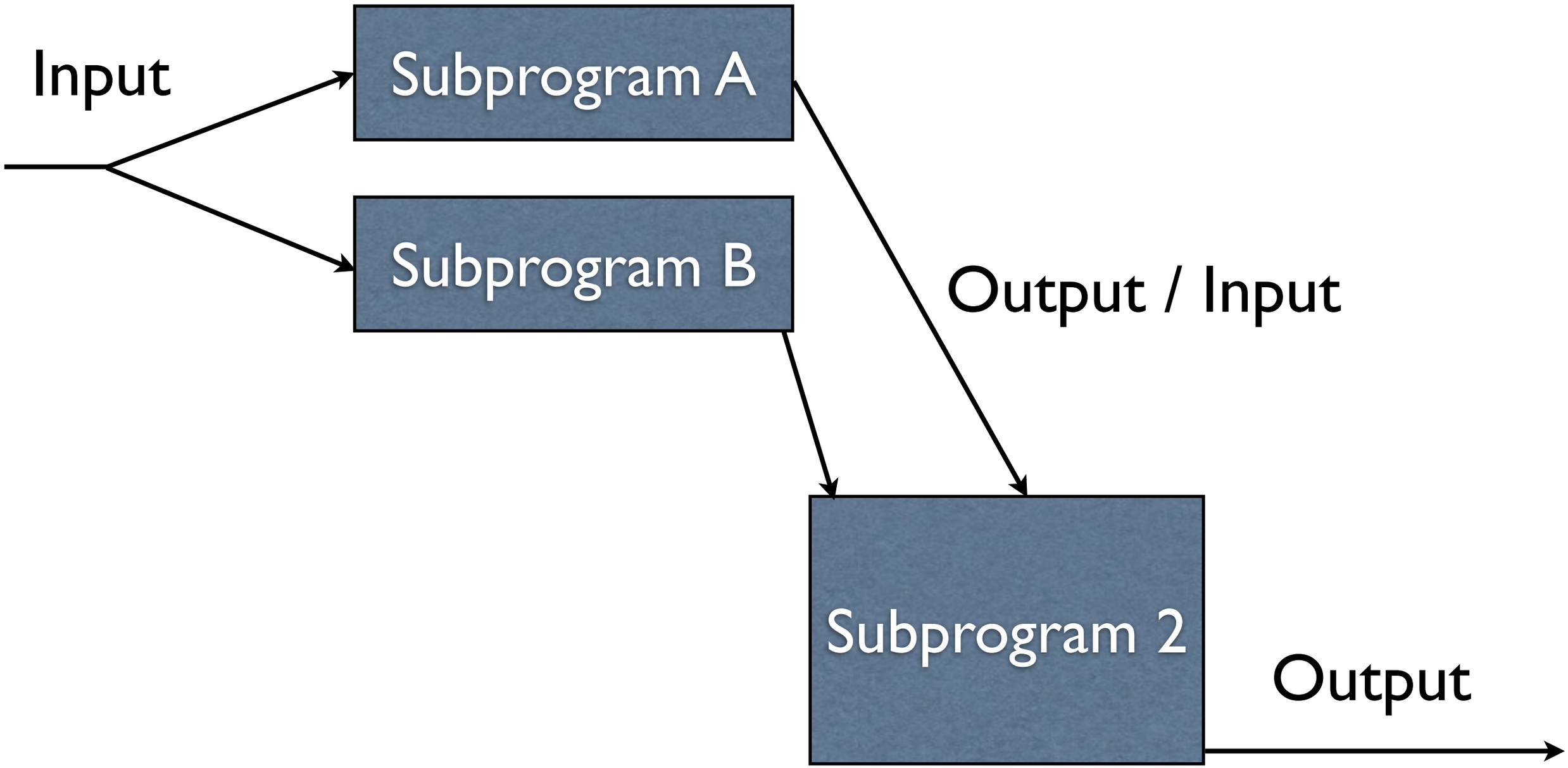
Divide and Conquer



Divide and Conquer



Divide and Conquer



General Five Step Process

1. Problem Statement
2. Input / Output Description
3. Hand Example
4. Algorithm Development
5. Testing

General Five Step Process

1. Problem Statement
2. Input / Output Description
3. Hand Example
4. Algorithm Development
5. Testing

Problem Statement

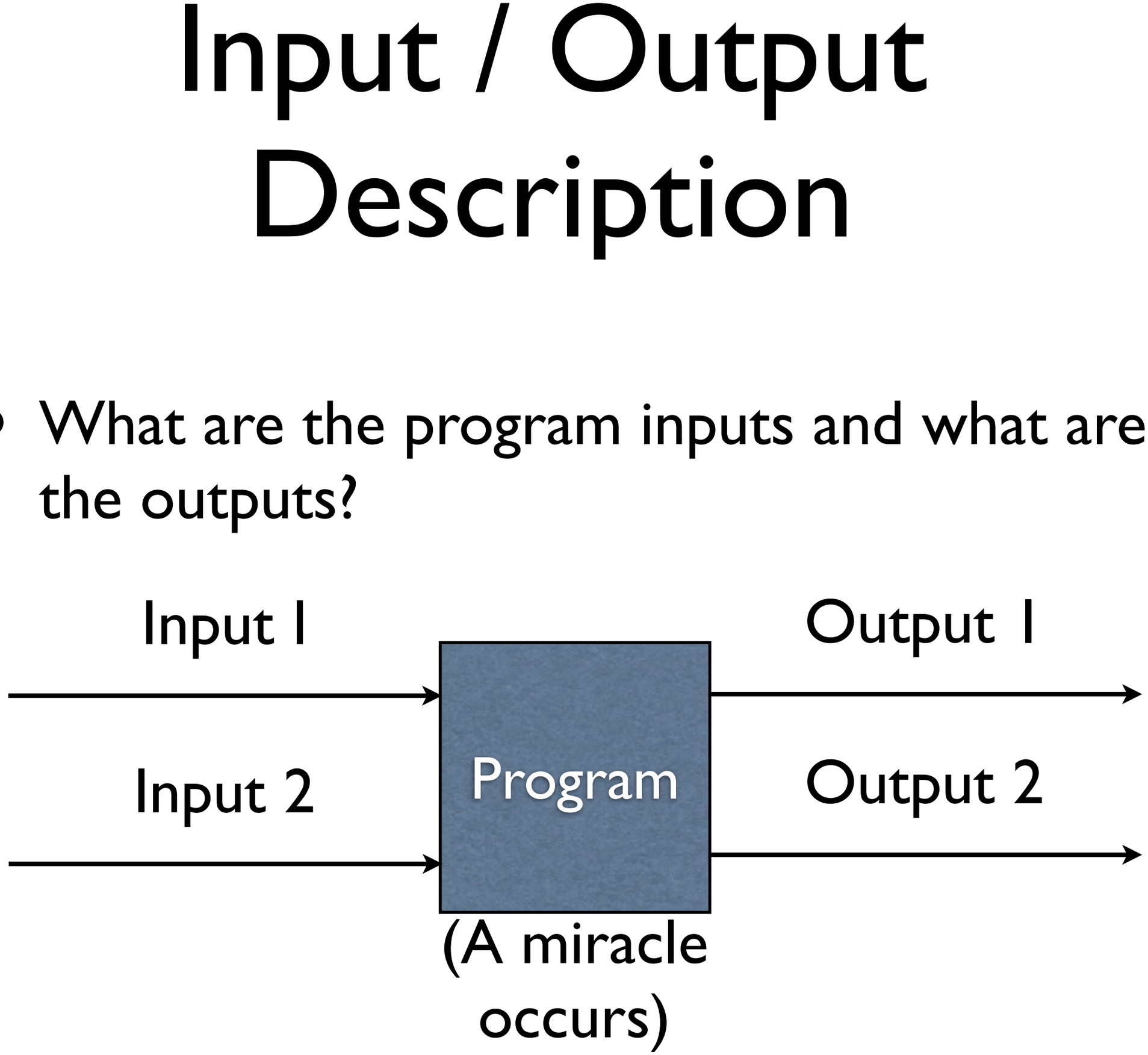
- What are we trying to solve?
- (Believe it or not, real software often goes astray here arguably more than anywhere else)

General Five Step Process

1. Problem Statement
2. Input / Output Description
3. Hand Example
4. Algorithm Development
5. Testing

Input / Output Description

- What are the program inputs and what are the outputs?



General Five Step Process

1. Problem Statement
2. Input / Output Description
3. Hand Example
4. Algorithm Development
5. Testing

Hand Example

- Do the problem by hand
- See if you actually understand what must be done

General Five Step Process

1. Problem Statement
2. Input / Output Description
3. Hand Example
4. Algorithm Development
5. Testing

Algorithm Development

- Steps needed to solve the problem
- Ultimately involves writing code
- “Days of coding can save you hours of planning”

General Five Step Process

1. Problem Statement
2. Input / Output Description
3. Hand Example
4. Algorithm Development
5. Testing

Testing

- See if solution actually works
- Important step!
- “50% of our code is tests”

TDD

- Test-driven development
- Write tests **before** code
- Wildly popular right now
- The point: testing is important!

C

hello.c

Running Code

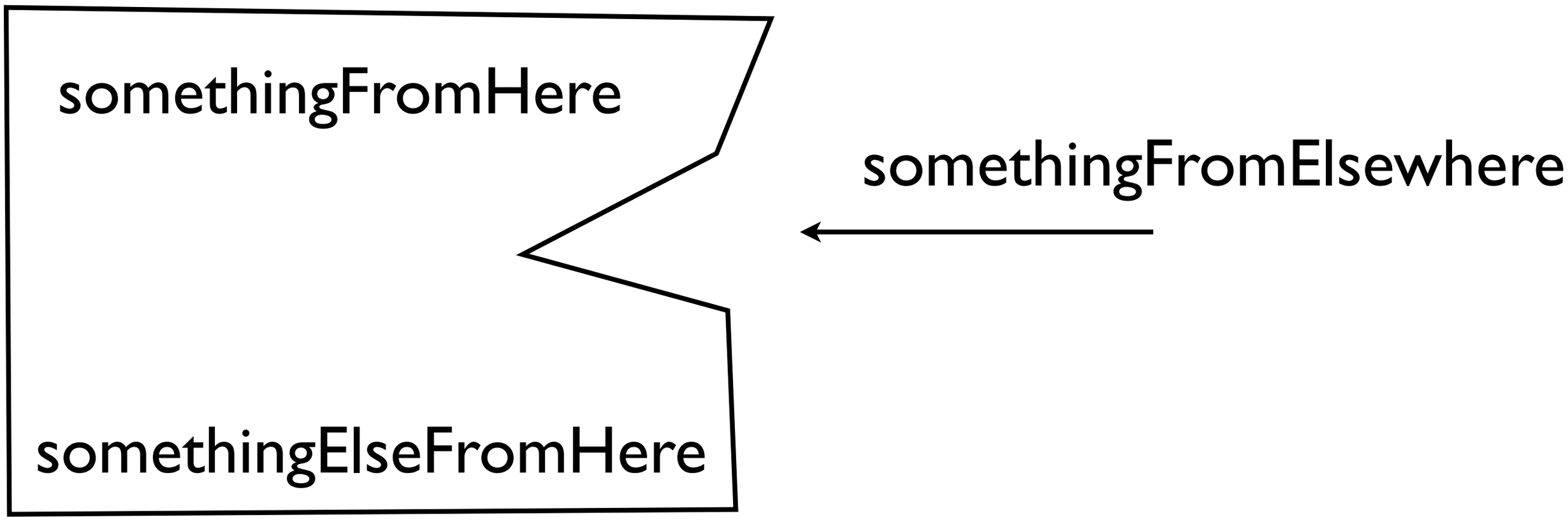
- Via `make` / `gcc` / `cc` (compilation)
- Via `ch` (interpretation)

Compilation

- Code is ultimately converted to a form the machine understands directly
- The result runs as fast as the machine
- Certain interdependencies not handled

Object Files

```
1: somethingFromHere();
2: somethingFromElsewhere();
3: somethingElseFromHere();
```



Linking

- Technically separate from compilation, but the two often get conflated
- Connects different pieces of code together

Linking

somethingFromElsewhere

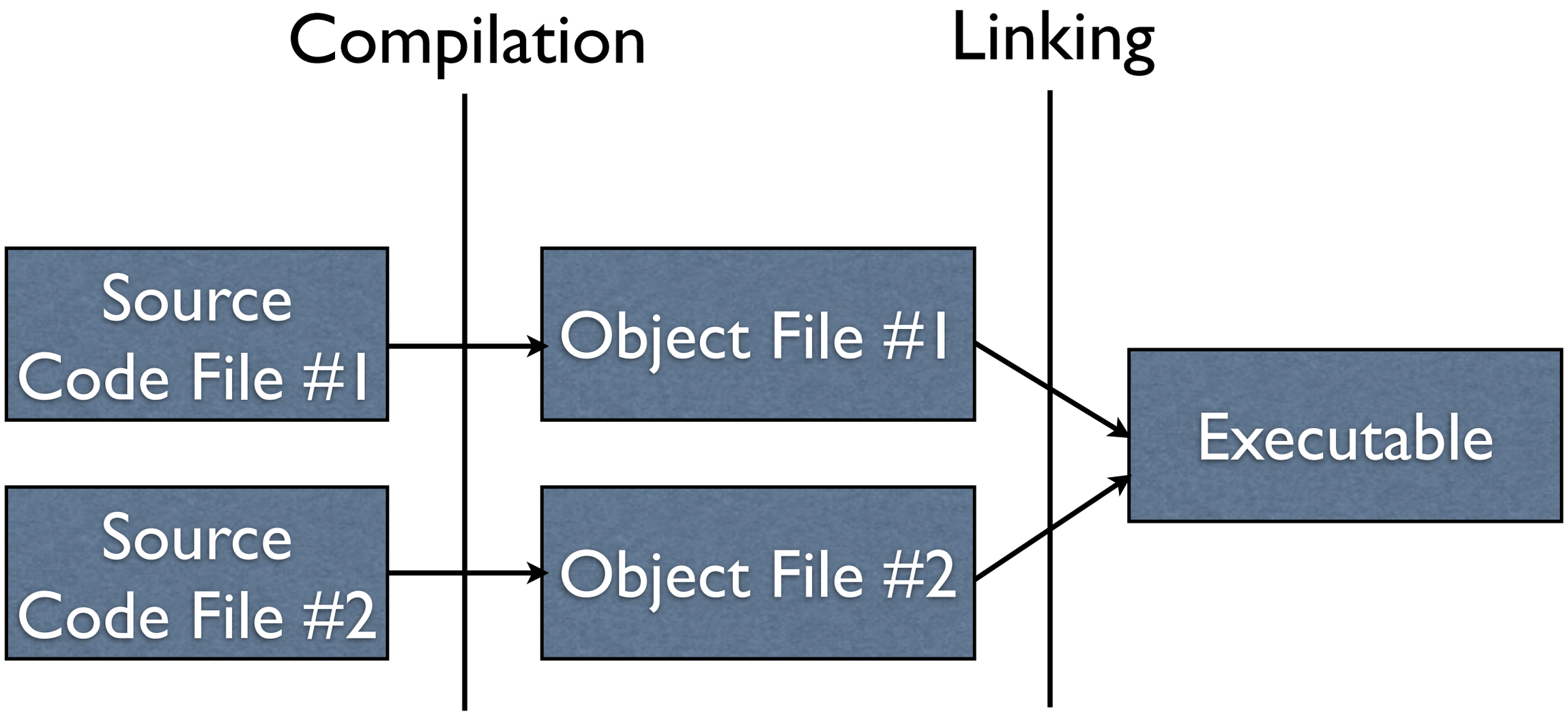
somethingFromHere

somethingElseFromHere

Linking

somethingFromHere
somethingFromElsewhere
somethingElseFromHere

Overall Process



Question

- Why separate compilation and linking?

Interpretation

- As with ch
- A program that understands the language runs it directly
- Runs as fast as the interpreter

Questions

- Why compile versus interpret?
- Why interpret versus compile?

Types

Types

- All data in C has a type
- Types determine:
  - What can be done with the data
  - How much space they take up (as with the `sizeof` operator)

Basic Types

- `int`: Integers (i.e. `1`, `2`, `3`...)
  - Come in both `signed` and `unsigned` forms
- `double`: floating point numbers (i.e. `5.1`, `7.8`, `2.93`, ...)
- `char`: A character (i.e. `a`, `b`, `c`,...)

Variables

- A container for a value
- Must have a type

Variable Declaration

- Telling the compiler that a variable with a given name and type can be used

```
int myInteger;
double myFloatingPoint;
char myCharacter;
```

Question

```
int myVariable;
int x = myVariable + 1;
// what does x equal?
```

Answer?

- `ch: l`
- `gcc: l`

Uninitialized Variables

- Officially, the initial value is "undefined"
- Undefined means "anything goes"
- Can be a source of tricky bugs

Variable Assignment

- The values of variables can be initialized...

```
int myVariable = 0;
```

-or-

```
int myVariable;
myVariable = 0;
```

Variable Assignment

- ...or changed on the fly...

```
int myVariable = 0;
myVariable = 5 + 2;
```

Variable Assignment

- ...or even be used to update the same variable!

```
int myVariable = 0;
myVariable = 5 + 2;
myVariable = 10 - myVariable;
```

Data Representation

Data Representation

- Many programming languages abstract away data representation
- C is not (exactly) one of those languages

Some Definitions

- bit: **b**inary dig**it** (true/false, 0/1, yes/no)
- byte: 8 bits
- kilobyte: 1024 bytes

Sizes

- If a bit can hold two possible values...
- And a byte holds 8 bits...
- How many possible values in a byte?
- A kilobyte?

Sizes and Types

- The actual size of variables depends on the compiler and the particular computer
- `sizeof` is your friend if it's important, as is `limits.h` (see `typelimits.c`)

Questions

- Integers are often 4 bytes. How many possible values can this hold?
- Signed versus unsigned integers: Does this make a difference for size?

Questions

- What's the largest number that can be stored in an unsigned 4 byte integer?
- What's around the largest number that can be stored in a signed 4 byte integer?