Typed Imperative Programming

CS 211

Road map

Hello, functions!

Road map

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Road map

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Road map

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Iteration

- The `while` statement
- The `for` statement

Road map

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Iteration

- The `while` statement
- The `for` statement

Making things happen

- Writing output
- Reading input

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/02_typedimp.tgz
⋮
% cd 02_typedimp
```

Up next

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Iteration

- The `while` statement
- The `for` statement

Making things happen

- Writing output
- Reading input

A first C function

```c
int square(int n)
{
    return n * n;
}
```

A first C function

```c
int square_int(int n)
{
    return n * n;
}

double square_double(double n)
{
    return n * n;
}
```

A first C function

```c
int square_int(int n)
{
    return n * n;
}

double square_double(double n)
{
    return n * n;
}

long square_long(long n)
{
    return n * n;
}
```

Up next

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Iteration

- The `while` statement
- The `for` statement

Making things happen

- Writing output
- Reading input

Definition

$$fib(n) = \begin{cases} n & \text{if } n < 2; \\ fib(n-2) + fib(n-1) & \text{otherwise.} \end{cases}$$

Definition

$$fib(n) = \begin{cases} n & \text{if } n < 2; \\ fib(n-2) + fib(n-1) & \text{otherwise.} \end{cases}$$

n	$fib(n)$
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

In C (don't do this at home!)

```c
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}
```

$$fib(n) = \begin{cases} n & \text{if } n < 2; \\ fib(n-2) + fib(n-1) & \text{otherwise.} \end{cases}$$

In C (don't do this at home!)

```
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}
```

```
long fib(int n)
{
    return (n < 2)? n : (fib(n - 2) + fib(n - 1));
}
```

In C (don't do this at home!)

```c
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}
```

```c
long fib(int n){return n<2?n:fib(n-2)+fib(n-1);}
```

In C (don't do this at home!)

```c
long fib(int n)
{
    return n < 2
        ? n
        : fib(n - 2) + fib(n - 1);
}
```

```c
long fib(int n){
return n<2?n:fib
(n-2)+fib(n-1);}
```

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

In C (less weird, but just as slow)

src/fib.c

```c
long fib_rec(int n)
{
    if (n < 2) {
        return n;
    } else {
        return fib_rec(n - 2) + fib_rec(n - 1);
    }
}
```

In C (less weird, but just as slow)

src/fib.c

```c
long fib_rec(int n)
{
    if (n < 2) {
        return n;
    } else {
        return fib_rec(n - 2) + fib_rec(n - 1);
    }
}
```

Syntax of `if`:

```
if (⟨test-expr⟩) {      // evaluate ⟨test-expr⟩; then...
    ⟨then-stms⟩         // do these if ⟨test-expr⟩ was true
} else {
    ⟨else-stms⟩         // do these if ⟨test-expr⟩ was false
}
```

In C (less weird, but just as slow)

src/fib.c

```c
long fib_rec(int n)
{
    if (n < 2) {
        return n;
    } else {
        return fib_rec(n - 2) + fib_rec(n - 1);
    }
}
```

The `else` clause is optional:

```c
if (⟨test-expr⟩) {      // evaluate ⟨test-expr⟩; then...
    ⟨then-stms⟩         // do these if ⟨test-expr⟩ was true
}
                        // otherwise do nothing
```

Note: Everything nests

```
if (⟨first-test-expr⟩) {
    if (⟨second-test-expr⟩) {
        ⟨A-stms⟩
    } else {
        ⟨B-stms⟩
    }
} else {
    if (⟨third-test-expr⟩) {
        ⟨C-stms⟩
    } else {
        ⟨D-stms⟩
    }
}
```

Problem: `fib` recomputes the same values many times

Problem: `fib` recomputes the same values many times

Solution: Mutation

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

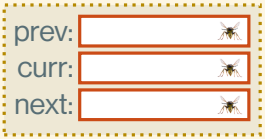
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



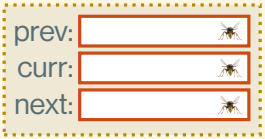
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



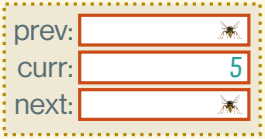
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



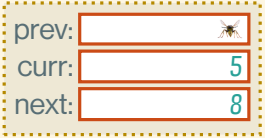
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



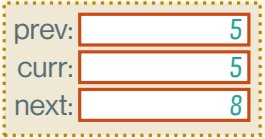
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



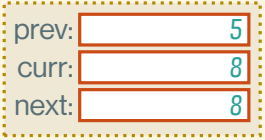
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



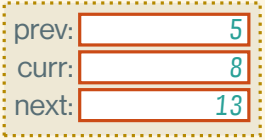
The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

▶   prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```



The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

The essence of imperative programming

```
{
    int prev;
    int curr = 5;
    int next = 8;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;

    prev = curr;
    curr = next;
    next = prev + curr;
}
```

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, images, etc.

Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, images, etc.

  Example: 3 is an `int` value

Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, images, etc.

  Example: 3 is an `int` value
- An object is a chunk of memory that can hold a value of a particular type.

Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, images, etc.

  Example: 3 is an `int` value
- An object is a chunk of memory that can hold a value of a particular type.

  Example: If a function `f` has a parameter `int x`, then each time `f` is invoked, a "fresh" object that can hold an `int` value is "created" to hold it.

Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, images, etc.

  Example: 3 is an `int` value
- An object is a chunk of memory that can hold a value of a particular type.

  Example: If a function `f` has a parameter `int x`, then each time `f` is invoked, a "fresh" object that can hold an `int` value is "created" to hold it.
- A variable is the name of an object, such as `x` from the previous bullet point.

Values, objects, and variables

- Values are the actual information we want to work with: numbers, strings, images, etc.

  Example: 3 is an `int` value
- An object is a chunk of memory that can hold a value of a particular type.

  Example: If a function `f` has a parameter `int x`, then each time `f` is invoked, a "fresh" object that can hold an `int` value is "created" to hold it.
- A variable is the name of an object, such as `x` from the previous bullet point.

Assigning a variable changes the value stored in the object that is named by the variable.

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

What happens?

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

What happens?

z: 5

The first statement is a definition, `int z = 5`. It creates an `int` object, names it `z`, and initializes it to the value 5.

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

What happens?

z: 7

The first statement is a definition, `int z = 5`. It creates an `int` object, names it `z`, and initializes it to the value 5.

The second statement is an assignment, `z = 7;`. It replaces the value 5 stored in the object named by `z` with the value 7.

Example of definition and assignment

```
int z = 5;
z = 7;
z = z + 4;
```

What happens?

z: 11

The first statement is a definition, `int z = 5`. It creates an `int` object, names it `z`, and initializes it to the value 5.

The second statement is an assignment, `z = 7;`. It replaces the value 5 stored in the object named by `z` with the value 7.

The third statement is also an assignment, `z = z + 4;`. It first retrieves the current value of `z` (7), then adds 4 to it, and then stores the result (11) back in the object named by `z`.

The key point: Indirection

A variable in C does not stand directly for a value.

A variable in C refers to a value *indirectly*, by naming an object that *contains* a value.

Problem: It's repetitive

Problem: It's repetitive

Solution: Iteration

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

Up next

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Iteration

- The `while` statement
- The `for` statement

Making things happen

- Writing output
- Reading input

The other ingredient: Iteration with `while`

Syntax:

```
while (⟨test-expr⟩) {
    ⟨body-stms⟩
}
```

The other ingredient: Iteration with `while`

Syntax:

```
while (⟨test-expr⟩) {
    ⟨body-stms⟩
}
```

Semantics:

1. Evaluate ⟨*test-expr*⟩ to a `bool`.
2. If the `bool` is `false` then skip to the statement after the `while` loop.
3. Execute ⟨*body-stms*⟩.
4. Go back to step 1.

In C, iteratively

src/fib.c

```c
long fib_iter(int n)
{
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n    = n - 1;
    }

    return curr;
}
```

In C, iteratively

src/fib.c

```c
long fib_iter(int n)
{
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;     // define local variable

        curr = next;          // assign copy
        next = prev + curr;   // assign sum
        n    = n - 1;         // decrement
    }

    return curr;
}
```

Note: How to decrement a variable

Simple:

```
x = x - 1;
```

Note: How to decrement a variable

Simple:

```
x = x - 1;
```

Terse:

```
x -= 1;
```

Note: How to decrement a variable

Simple:

```
x = x - 1;
```

Terse:

```
x -= 1;
```

Auto-decrement:

```
--x;
```

Note: How to decrement a variable

Simple:

```
x = x - 1;
```

Terse:

```
x -= 1;
```

Auto-decrement:

```
--x;
```

Each of the above is actually an expression, and it has a value—the new value of `x`

Note: How to decrement a variable

Simple:

```
if ((x = x - 1) > 0) ...;
```

Terse:

```
if ((x -= 1) > 0) ...;
```

Auto-decrement:

```
if (--x > 0) ...;
```

Each of the above is actually an expression, and it has a value—the new value of `x`

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

Counting upwards

```
long fib(int n)
{
    long prev, curr = 0, next = 1;
    int i = 0;

    while (i < n) {
        prev = curr;
        curr = next;
        next = prev + curr;
        ++i;                // equivalent to i += 1;
    }

    return curr;
}
```

Counting upwards

Let's change the `while` loop to an equivalent `for`...

```
long fib(int n)
{
    long prev, curr = 0, next = 1;
    int i = 0;

    while (i < n) {
        prev = curr;
        curr = next;
        next = prev + curr;
        ++i;                // equivalent to i += 1;
    }

    return curr;
}
```

Counting upwards with `for`

A `for` loop equivalent to the preceding `while` loop...

```
long fib(int n)
{
    long prev, curr = 0, next = 1;
    int i = 0;

    for (; i < n; ) {
        prev = curr;
        curr = next;
        next = prev + curr;
        ++i;
    }

    return curr;
}
```

Counting upwards with `for`

Move the increment into the “step” header...

```
long fib(int n)
{
    long prev, curr = 0, next = 1;
    int i = 0;

    for (; i < n; ++i) {
        prev = curr;
        curr = next;
        next = prev + curr;
        // ++i
    }

    return curr;
}
```

Counting upwards with `for`

Define the counter in the “start” header...

```
long fib(int n)
{
    long prev, curr = 0, next = 1;
    // int i = 0;

    for (int i = 0; i < n; ++i) {
        prev = curr;
        curr = next;
        next = prev + curr;
        // ++i
    }

    return curr;
}
```

Counting upwards with `for`

Now cleaned up:

```
long fib(int n)
{
    long prev,
         curr = 0,
         next = 1;

    for (int i = 0; i < n; ++i) {
        prev = curr;
        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

The meaning of `for` in terms of `while`

When you write a `for` statement like this:

```
for (⟨start-decl⟩; ⟨test-expr⟩; ⟨step-expr⟩) {
    ⟨body-stms⟩
}
```

it's as if you'd written this `while` statement*:

```
{
    ⟨start-decl⟩;
    while (⟨test-expr⟩) {
        ⟨body-stms⟩
        ⟨step-expr⟩;
    }
}
```

The meaning of `for` in terms of `while`

When you write a `for` statement like this:

```
for (⟨start-decl⟩; ⟨test-expr⟩; ⟨step-expr⟩) {
    ⟨body-stms⟩
}
```

it's as if you'd written this `while` statement*:

```
{
    ⟨start-decl⟩;
    while (⟨test-expr⟩) {
        ⟨body-stms⟩
        ⟨step-expr⟩;
    }
}
```

*not accounting for `continue` statements in ⟨*body-stms*⟩

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

The `main` function

C programs can have multiple functions, but the system always starts them by calling their `main` function:

```
int main(void)
{
    ⟨body-stms⟩

    return 0;
}
```

The `main` function

C programs can have multiple functions, but the system always starts them by calling their `main` function:

```
int main(void)
{
    ⟨body-stms⟩

    return 0;
}
```

- `int` means `main` returns an `int` to the operating system

The `main` function

C programs can have multiple functions, but the system always starts them by calling their `main` function:

```
int main(void)
{
    ⟨body-stms⟩

    return 0;
}
```

- `int` means `main` returns an `int` to the operating system
- `(void)` means our `main` expects no arguments

The `main` function

C programs can have multiple functions, but the system always starts them by calling their `main` function:

```
int main(void)
{
    ⟨body-stms⟩

    return 0;
}
```

- `int` means `main` returns an `int` to the operating system
- `(void)` means our `main` expects no arguments
- `main` should return `0` for success or non-zero for failure

The `main` function

C programs can have multiple functions, but the system always starts them by calling their `main` function:

```
int main(void)
{
    ⟨body-stms⟩
    // only main() does this implicity:
    // return 0;
}
```

- `int` means `main` returns an `int` to the operating system
- `(void)` means our `main` expects no arguments
- `main` should return `0` for success or non-zero for failure

Up next

- Hello, functions!
- Example: Computing Fibonacci numbers
  - The `?:` expression
  - The `if` statement
- Mutation
  - Understanding assignment
- Iteration
  - The `while` statement
  - The `for` statement
- Making things happen
  - Writing output
  - Reading input

Introducing `printf`

The usual way to print in C is the `printf(3)` function, which takes a *format string* followed by arguments to *interpolate* in place of the format string's *directives*:

```
int x, y;
⋮
printf("(%d, %d)\n", x, y);
```

(Displays "`(`", the value of `x`, "`, `", the value of `y`, "`)`", and a line break)

Example: Formatted output

src/output.c

```c
#include <stdio.h>
int main(void)
{
    int x = 5;
    double f = 5.1;
    printf("sizeof x: %zu bytes\n", sizeof x);
    printf("sizeof f: %zu bytes\n", sizeof f);
    printf("x: %d\nf: %.60e\n", x, f);
}
```

Example: Formatted output

src/output.c

```c
#include <stdio.h>
int main(void)
{
    int x = 5;
    double f = 5.1;
    printf("sizeof x: %zu bytes\n", sizeof x);
    printf("sizeof f: %zu bytes\n", sizeof f);
    printf("x: %d\nf: %.60e\n", x, f);
}
```

A directive gives an argument's type & maybe some options:

directive	type	options/(note)
`%zu`	`size_t`	(the result type of `sizeof`)
`%d`	`int`	
`%.60e`	`double`	include 60 digits of precision

Note: Including headers

This is a directive that causes the functions defined in `stdio.h` to be known to the compiler:

```
#include <stdio.h>
```

(Without it, we wouldn't have access to `printf`)

Up next

Hello, functions!

Example: Computing Fibonacci numbers

- The `?:` expression
- The `if` statement

Mutation

- Understanding assignment

Iteration

- The `while` statement
- The `for` statement

Making things happen

- Writing output
- Reading input

Reading user input

To input numbers in C, use the `scanf(3)` function.

Reading user input

To input numbers in C, use the `scanf(3)` function.

`scanf` reads keyboard input, converts it to the required type, and stores it in an existing variable:

```
int x = 0;
scanf("%d", &x);
```

Reading user input

To input numbers in C, use the `scanf(3)` function.

`scanf` reads keyboard input, converts it to the required type, and stores it in an existing variable:

```
int x = 0;
scanf("%d", &x);
```

- Like `printf(3)`, `scanf` uses a format string to determine what type to convert the input to.

Reading user input

To input numbers in C, use the `scanf(3)` function.

`scanf` reads keyboard input, converts it to the required type, and stores it in an existing variable:

```
int x = 0;
scanf("%d", &x);
```

- Like `printf(3)`, `scanf` uses a format string to determine what type to convert the input to.
- An argument `x` would pass the *value* of variable `x` to `scanf`, but `&x` means to pass `x`'s *location*.

Reading user input

To input numbers in C, use the `scanf(3)` function.

`scanf` reads keyboard input, converts it to the required type, and stores it in an existing variable:

```
int x = 0;
scanf("%d", &x);
```

- Like `printf(3)`, `scanf` uses a format string to determine what type to convert the input to.
- An argument `x` would pass the *value* of variable `x` to `scanf`, but `&x` means to pass `x`'s *location*.
- Careful: `scanf`'s directives aren't exactly the same as `printf`'s!

Example: Reading input

src/input.c

```c
#include <stdio.h>

double sqr_dbl(double n)
{
    return n * n;
}

int main(void)
{
    double d = 0.0;

    scanf("%lf", &d);
    printf("%lf squared is %lf\n", d, sqr_dbl(d));
}
```

Example: Reading multiple items

src/multi_input.c

```c
#include <stdio.h>

int main(void)
{
    int x, y;

    printf("Enter two integers: ");
    scanf("%d%d", &x, &y);
    printf("%d * %d = %d\n", x, y, x * y);
}
```

How `scanf` reports errors

`scanf` returns the number of successful conversions.

Example: Checking for input errors

src/check_input.c

```c
#include <stdio.h>

int main(void)
{
    int x, y;

    printf("Enter two integers: ");

    if (scanf("%d%d", &x, &y) != 2) {
        printf("Input error\n");
        return 1;
    }

    printf("%d * %d == %d\n", x, y, x * y);
}
```

A `main` function for the *`fib`* program

src/fib.c

```c
#include <stdio.h>

long fib(int n)
{ … }

int main(void)
{
    int buf;

    while (scanf("%d", &buf) == 1) {
        printf("fib(%d) = %ld\n", buf, fib(buf));
    }
}
```

– Next time: Separate compilation –