

# Contracts

# **A Mystery Function**

# The Story

*Your first task at your new job is to debug this code written by your predecessor, who was fired for being a poor programmer.*

```
int f(int x, int y) {  
    int r = 1;  
    while (y > 1) {  
        if (y % 2 == 1) {  
            r = x * r;  
        }  
        x = x * x;  
        y = y / 2;  
    }  
    return r * x;  
}
```

*This is all you  
are given*

***How do you go about this “friendly” challenge?***

# The Language

- This code is written in **C0**
  - The language we will use for most of this course
- This is also valid **C** code
  - For the most part, C0 programs are valid C programs
  - We will use C0 as a gentler language to
    - learn to write complex code that is correct
    - learn to write code in C itself
- *But what does this function do?*

```
int f(int x, int y) {  
    int r = 1;  
    while (y > 1) {  
        if (y % 2 == 1) {  
            r = x * r;  
        }  
        x = x * x;  
        y = y / 2;  
    }  
    return r * x;  
}
```

# The Programmer

- Is this good code?
  - there are no comments
  - the names are non-descript
    - the function is called `f`
    - the variables are called `x`, `y`, `r`

No! ✘

- No wonder your predecessor was fired as a programmer!

```
int f(int x, int y) {  
    int r = 1;  
    while (y > 1) {  
        if (y % 2 == 1) {  
            r = x * r;  
        }  
        x = x * x;  
        y = y / 2;  
    }  
    return r * x;  
}
```

- *But what does this function do?*

# The Function

- But what does this function do?
- We can run experiments
  - call `f` with various inputs and observe the outputs
- We do so by loading it in the **C0 interpreter** – coin

The command for the C0 interpreter

The file where we saved the function

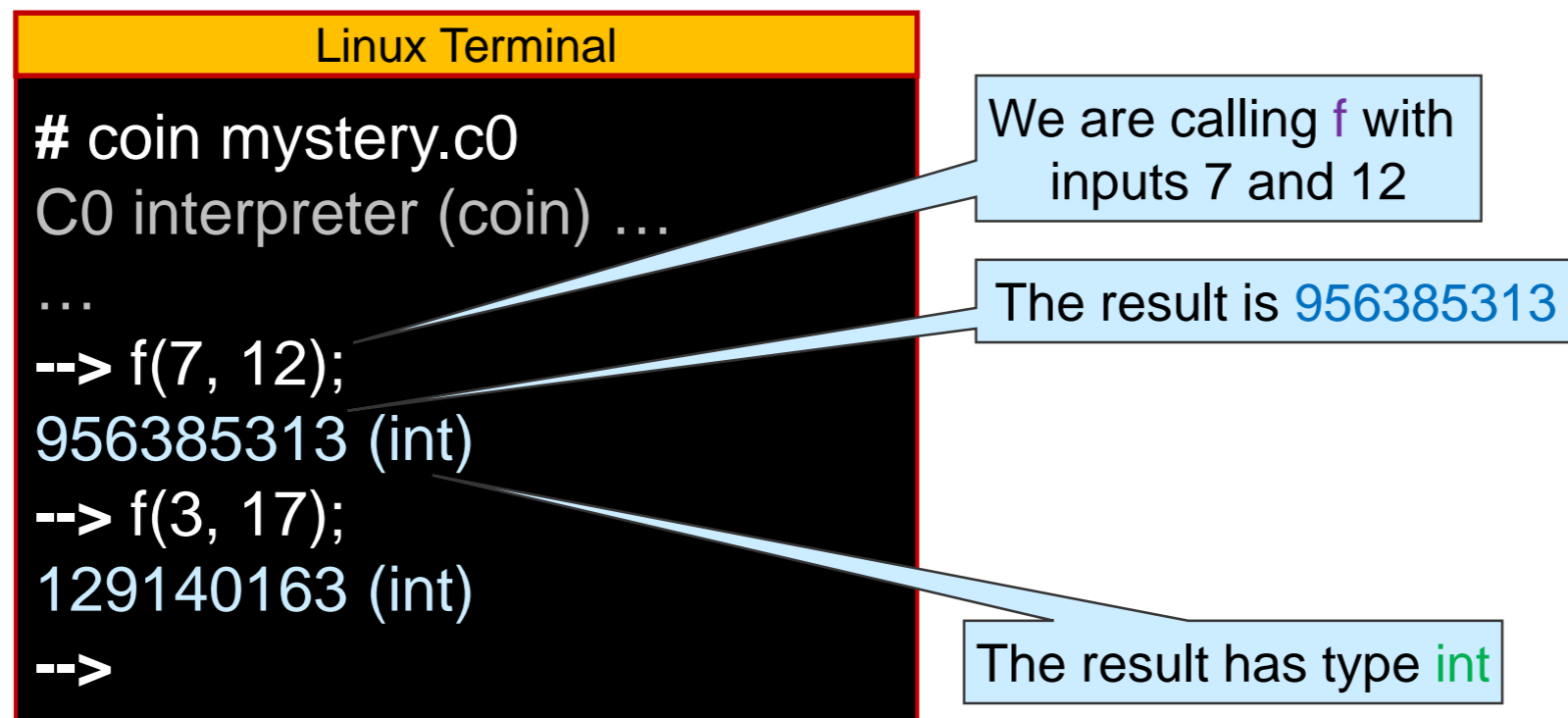
Linux Terminal

```
# coin mystery.c0
C0 interpreter (coin) 0.3.3 'Nickel' (r590, Mon Aug 29 12:04:13 UTC 2016)
Type `#help' for help or `#quit' to exit.
-->
```

The coin prompt

# Running Experiments

- Call `f` with various inputs and observe the outputs



A screenshot of a Linux terminal window titled "Linux Terminal". The terminal shows the following text:

```
# coin mystery.c0
C0 interpreter (coin) ...
...
--> f(7, 12);
956385313 (int)
--> f(3, 17);
129140163 (int)
-->
```

Three callout boxes with light blue backgrounds and black borders point to specific parts of the terminal output:

- The first callout box points to the input `f(7, 12);` and contains the text: "We are calling `f` with inputs 7 and 12".
- The second callout box points to the output `956385313 (int)` and contains the text: "The result is `956385313`".
- The third callout box points to the output `(int)` and contains the text: "The result has type `int`".

- These are not very good experiments
  - they don't help us understand what `f` does

# Running Experiments

- Call `f` with various inputs and observe the outputs
  - we are better off calling `f` with small inputs
  - and vary them by just a little bit so we can spot a pattern

```
Linux Terminal
--> f(2, 3);
8 (int)
--> f(2, 4);
16 (int)
--> f(2, 5);
32 (int)
--> f(2, 6);
64 (int)
-->
```

Much better!

- It looks like  $f(x, y)$  computes  $x^y$
- Let's confirm with more experiments



# Confirming the Hypothesis

- It looks like  $f(x, y)$  computes  $x^y$
- Let's confirm with more experiments

```
Linux Terminal
--> f(2, 2);
4 (int)
--> f(3, 2);
9 (int)
--> f(4, 2);
16 (int)
--> f(5, 2);
25 (int)
-->
```

Yep! That's  $x^y$

- We find a secret memo in a hidden drawer

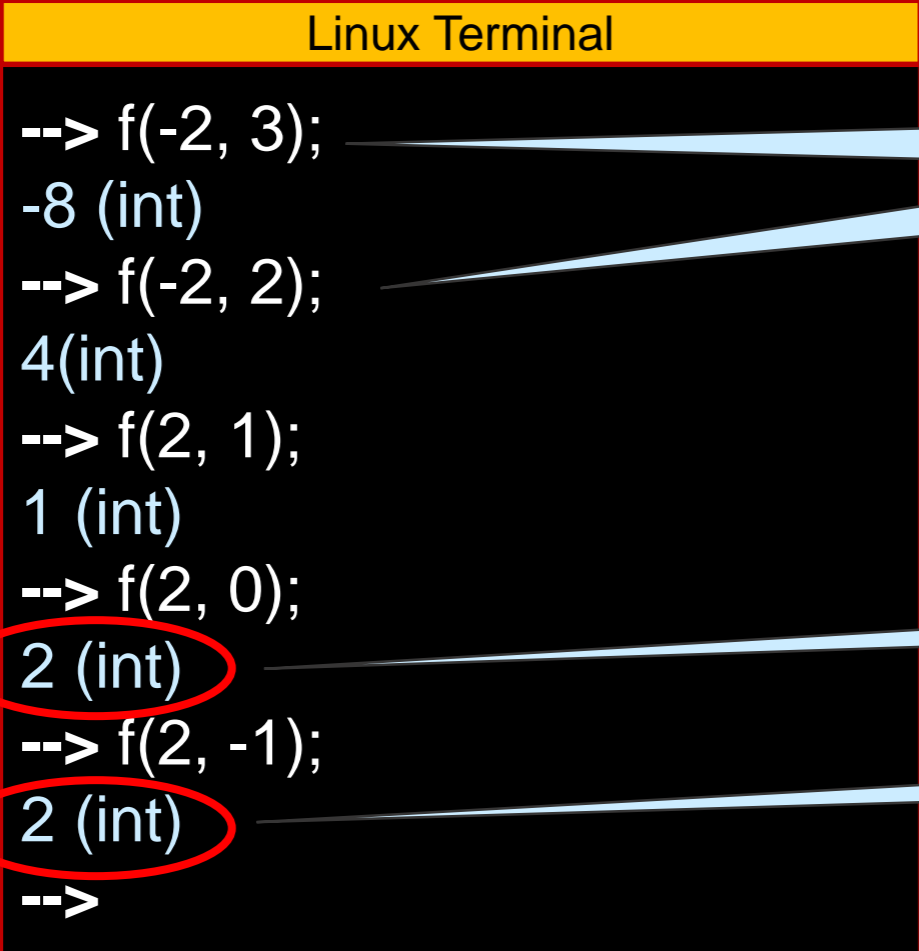
*Power not working.  
Fix by tonight or you're out*

Not the friendliest of work places!

- Let's run a few more experiments to identify the problem

# Discovering the Bug

- $f(x, y)$  is *meant to* compute  $x^y$ 
  - but it doesn't
- Let's find where it fails with more experiments



```
Linux Terminal
--> f(-2, 3);
-8 (int)
--> f(-2, 2);
4(int)
--> f(2, 1);
1 (int)
--> f(2, 0);
2 (int)
--> f(2, -1);
2 (int)
-->
```

Callouts:

- It seems to work for negative values of x
- That's not  $2^0$
- That's definitely not  $2^{-1}$

- Now we have something to chew on

# Preconditions

# The Power Function

- What does it mean to be the power function  $x^y$  ?

- $$\underbrace{x * \dots * x}_{y \text{ times}}$$

➤ Yes, but that's not very precise

- Let's write a *mathematical* definition

- $$\begin{cases} x^0 = 1 \\ x^y = x^{y-1} * x \end{cases}$$

This is a *recursive* definition

and this is its base case

# The Power Function

- What does it mean to be the power function  $x^y$  ?

$$\begin{cases} x^0 = 1 \\ x^y = x^{y-1} * x \end{cases}$$

○ What happens if  $y$  is negative?

➤ we never reach the base case ...

- The power function  $x^y$  on integers is **undefined** if  $y < 0$

$$\begin{cases} x^0 = 1 \\ x^y = x^{y-1} * x \end{cases} \quad \text{if } y > 0$$

This defines  $x^y$  for  $y \geq 0$  only

# The Power Function

```
int f(int x, int y) {  
    int r = 1;  
    while (y > 1) {  
        if (y % 2 == 1) {  
            r = x * r;  
        }  
        x = x * x;  
        y = y / 2;  
    }  
    return r * x;  
}
```

- What does it mean to be the power function  $x^y$  ?

$$\begin{cases} x^0 = 1 \\ x^y = x^{y-1} * x & \text{if } y > 0 \end{cases}$$

- To implement the power function, **f** must disallow negative exponents

- It can raise an error

We need to test **y**.  
This would slow **f** down a bit.

- It can tell the caller that the exponent should be  $\geq 0$

Better!  
no need to test **y**

# Preconditions

- Disallow negative exponents
  - by telling the caller that the exponent should be  $\geq 0$
- A restriction on the admissible inputs to a function is called a **precondition**
  - We need to impose a precondition on `f`
  - In most languages, we are limited to writing a comment
    - and hope the caller reads it

This is how we would write a precondition in C

```
// y must be greater than or equal to 0
int f(int x, int y) {
    int r = 1;
    while (y > 1) {
        if (y % 2 == 1) {
            r = x * r;
        }
        x = x * x;
        y = y / 2;
    }
    return r * x;
}
```

# Preconditions in C0

- We need to impose a precondition on `f`
  - to tell the caller that `y` should be  $\geq 0$
- In C0 we can write an **executable contract directive**

```
//@requires y >= 0;
```

C0 keyword to specify a precondition

- written between the function header and the body
- before the first “{”

- We check contracts by invoking `coin` with the **-d** flag
  - “dynamic checking”
    - but everybody understands it as *debug mode*
- without the **-d** flag, contracts are treated as comments

```
int f(int x, int y)
//@requires y >= 0;
{
    int r = 1;
    while (y > 1) {
        if (y % 2 == 1) {
            r = x * r;
        }
        x = x * x;
        y = y / 2;
    }
    return r * x;
}
```



# Using Contract

## Running with contracts disabled

```
Linux Terminal
# coin mystery.c0
C0 interpreter (coin) ...
--> f(2, 3);
8 (int)
--> f(2, -1);
2 (int)
-->
```

Contracts are treated as comments

cc0, the C0 compiler, works the same way

## Running with contracts enabled

```
Linux Terminal
# coin -d mystery.c0
C0 interpreter (coin) ...
--> f(2, 3);
8 (int)
--> f(2, -1);
mystery.c0:2.4-2.20: @requires annotation failed
Last position: mystery.c0:2.4-2.20
f from <stdio>:1.1-1.9
-->
```

Contracts are executed

- if **true**, execution proceeds normally
- if **false**, execution aborts

Line number where contract failed

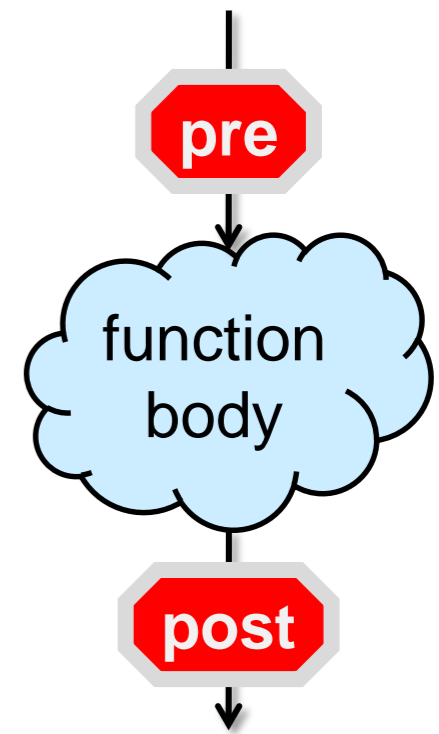
# Safety

- If we call  $f(x,y)$  with a negative  $y$ 
  - with **-d**, execution aborts
  - without **-d**,  $f$  can return an arbitrary result
    - there is **no** right value it could return
- Calling a function with inputs that cause a precondition to fail is **unsafe**
  - execution will never do the right thing
    - either abort
    - or compute a wrong result
- The caller must make sure that the call is **safe**
  - that  $y \geq 0$

# Postconditions

# Contracts about Function Outcomes

- Preconditions are checked *before* the function starts executing
- A contract that is checked *after* it is done executing could tell us if the function did the right thing
  - check that the output is what we expect
  - This is a **postcondition**



# Postconditions in C0

- In C0, the contract directive

`//@ensures <some_condition>;`

C0 keyword to specify a postcondition

- written between the function header and the body
  - after the preconditions (by convention)
- before the first “{“

allows us to write a postcondition

- `<some_condition>` can mention the contract-only variable `\result`
  - what the function returns
  - can only be used with `//@ensures`

```
int f(int x, int y)
//@requires y >= 0;
//@ensures ...;
{
    int r = 1;
    while (y > 1) {
        if (y % 2 == 1) {
            r = x * r;
        }
        x = x * x;
        y = y / 2;
    }
    return r * x;
}
```

# Writing a Postcondition

- The postcondition we want to write is

```
//@ensures \result == x**y;
```

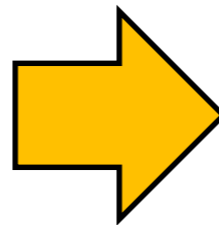
- but  $x^{**}y$  is not defined in C0
  - C0 has no primitive power function!

That's how we write  $x^y$  in Python

- What do we do?

- transcribe the mathematical definition into a C0 function

$$\begin{cases} x^0 = 1 \\ x^y = x^{y-1} * x & \text{if } y > 0 \end{cases}$$



```
int POW(int x, int y)
//@requires y >= 0;
{
    if (y == 0) return 1;
    return POW(x, y-1) * x;
}
```

# Writing a Postcondition

- Then our postcondition is

```
//@ensures \result == POW(x, y);
```

right? ... almost

```
Linux Terminal
# coin -d mystery.c0
mystery.c0:18.5-18.6:error:cannot assign to
variable 'x' used in @ensures annotation
x = x * x;
~
Unable to load files, exiting...
```

- The function modifies **x** (and **y**)
  - Which values of **x** and **y** should C0 evaluate the postcondition with?
    - We want the initial values, but it is checked when returning ...

- To avoid confusion, C0 disallows modified variables in postconditions

```
int POW(int x, int y)
//@requires y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1) * x;
}

int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) {
      r = x * r;
    }
    x = x * x;
    y = y / 2;
  }
  return r * y;
}
```

# Writing a Postcondition

- C0 disallows modified variables in postconditions
  - Make copies `x` and `y` and modify those

```
Linux Terminal
# coin -d mystery.c0
C0 interpreter (coin) ...
--> f(2, 3);
8 (int)
--> f(2, 0);
mystery.c0:11.4-11.33: @ensures annotation failed
Last position: mystery.c0:11.4-11.33
f from <stdio>:1.1-1.8
```

- We're good

Line number  
where contract failed

```
int POW(int x, int y)
//@requires y >= 0;
{
    if (y == 0) return 1;
    return POW(x, y-1) * x;
}

int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```



# Recall Safety

- In the postcondition of `f`, we are making a call to `POW`

- **Is it safe?**

- We need to show that  $y \geq 0$

- The precondition tells us that  $y \geq 0$  ✓

This should always be on our mind

- The body of `POW` makes a call to `POW`

- **Is it safe?**

- We need to show that  $y-1 \geq 0$

- The precondition tells us that  $y \geq 0$

- Since we don't return on the `if`,  $y > 0$

- So  $y-1 \geq 0$  by math ✓

```
int POW(int x, int y)
//@requires y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1) * x;
}

int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1) {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

- These are examples of **point-to reasoning**

- We justify something by pointing to lines of code that justify it

# Specification Functions

- **POW** is used only in contracts
  - It is not executed when contract-checking is disabled
    - without **-d**
- Functions used only in contracts are called **specification functions**
  - They help us state what the code should do
  - They are critical to writing good code

```
int POW(int x, int y)
//@requires y >= 0;
{
    if (y == 0) return 1;
    return POW(x, y-1) * x;
}

int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```

# The Power Function

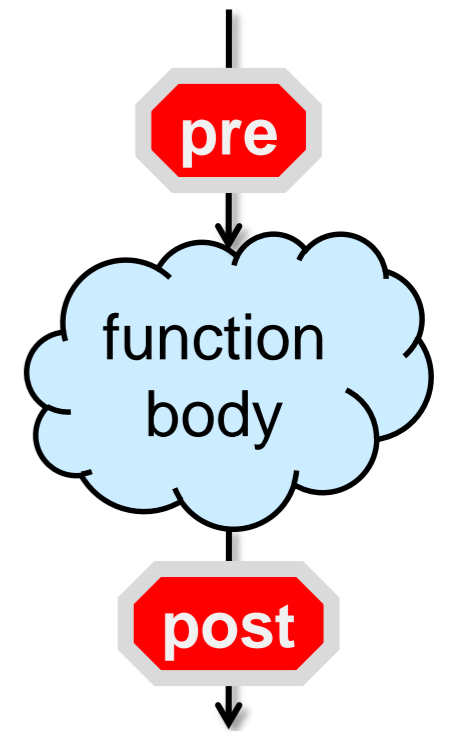
- But wait!
  - `f` was meant to implement the power function
  - ... but **POW** is the power function!
- Let's use it!
  - There may be benefits to fixing `f` instead
    - it may be more efficient than **POW**
  - Keep reading ...

```
int POW(int x, int y)
//@requires y >= 0;
{
    if (y == 0) return 1;
    return POW(x, y-1) * x;
}

int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```

# Correctness

- If a call violates a function's postconditions  
(assuming its preconditions were met so it actually ran)  
it is doing something wrong
  - the function has a **bug**
- The function is **incorrect**
  - Our mystery function **f** is incorrect
- The writer of the function must make sure that it is **correct**
  - i.e., that its postconditions will be satisfied for any input that passes its preconditions



# Blame

- If a function preconditions fail, it's the caller's fault
  - the caller passed invalid inputs
  - the call is **unsafe**
- If its postconditions fail, it's the implementation's fault
  - the function code does the wrong thing
  - the function is **incorrect**

We will develop methods to make sure that the code we write is **safe** and **correct**

# How to Use Contracts

- Contract-checking helps us write code that works as expected
  - Use **-d** while writing our code
  - At this stage, this is **development code**
    - bugs are likely
- Once we are confident our code works, compile it without **-d**
  - The code can be used in its intended application
  - At this stage, this is **production code**
    - there should be no bugs
- Why not use **-d** always?
  - it slows down execution

# Function Contracts

# Where are we?

- We have learned a lot about **f**
  - the preconditions describe what valid inputs are
  - the postconditions describe what it is supposed to do
    - on valid inputs
- We have a fully documented function
- We have not looked at all at its body
  - but we know there is a bug in there
  - it is incorrect

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```



# The Caller's Perspective

*Preconditions describe valid inputs*

*Postconditions describe what it does*

- That's what the **caller** needs to know to use the function

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
```

Header:

- function name
- number and type of its arguments

Contracts:

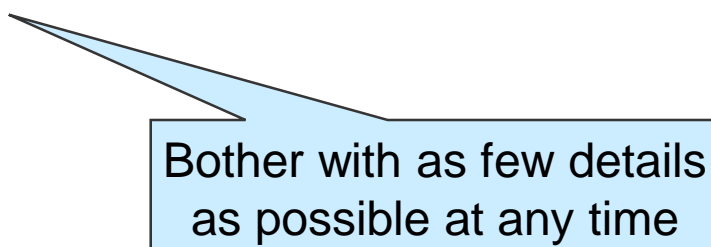
- pre- and post-conditions

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  +
}
```

- The caller should be able to use it without knowing anything about how it is implemented
  - The implementation details are **abstracted away**

# Abstraction

- Split a complex system into **small** chunks that can be understood **independently**



Bother with as few details  
as possible at any time

- Computer science is all about abstraction

# The Function's Perspective

*Preconditions describe valid inputs*

*Postconditions describe what it does*

- That's what the implementation is to do
  - guidelines to write the body of the function
- How to write good code
  - **First write the contracts**
  - and then the body
    - in this way, you always know what you are aiming for

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```

*Now, we need to look at the body of  $f$  to find the bug*

# Loop Invariants

# Diving In


- We need to look at the body of **f**
  - The complicated part is the **loop**
    - the values of the variables change at each iteration
    - it's unclear how many iterations there are
  - If we understand the loop, we understand the function
- How to go about that?

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```

# Abstraction

- If we understand the loop, we understand the function
- How to go about that?
  - Contracts summarize what a function does so we don't need to bother with the details of its implementation
    - An abstraction over functions
  - Come up with a summary of the loop so we don't need to bother with the details of its implementation
    - **An abstraction over loops!**

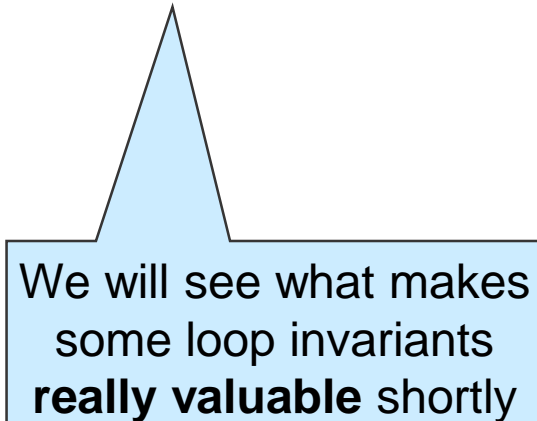
```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        // ...
    }
    return r * b;
}
```



# Loop Invariants

The values of the variables change at each iteration

- One valuable abstraction is what does **not** change
  - This is called a **loop invariant**
    - a quantity that remains constant at each iteration of the loop
      - a quantity may be an expression, not just a variable



We will see what makes some loop invariants **really valuable** shortly

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```

# Tracing Code

- How to find a **loop invariant**?

- a quantity that remains constant at each iteration of the loop

- Run the function on sample inputs

- Track the value of the variables

- b, e, r

- no need to bother with x and y since they don't change

- just before the loop guard is tested

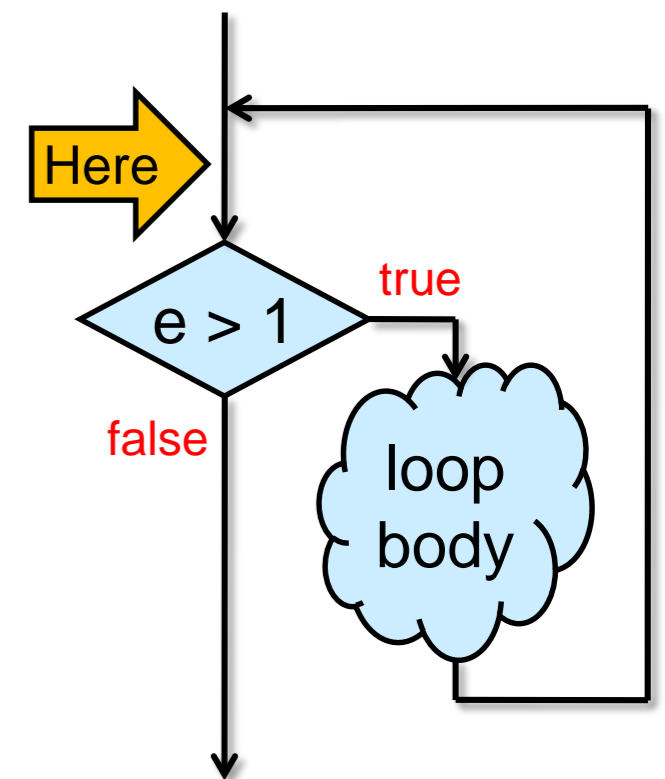
- That's  $e > 1$

This is called **tracing** an execution

- Look for patterns

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1) {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

Loop guard





# Tracing Code

- Run the function on sample inputs and track the value of the variables
  - Let's try with  $f(2,8)$

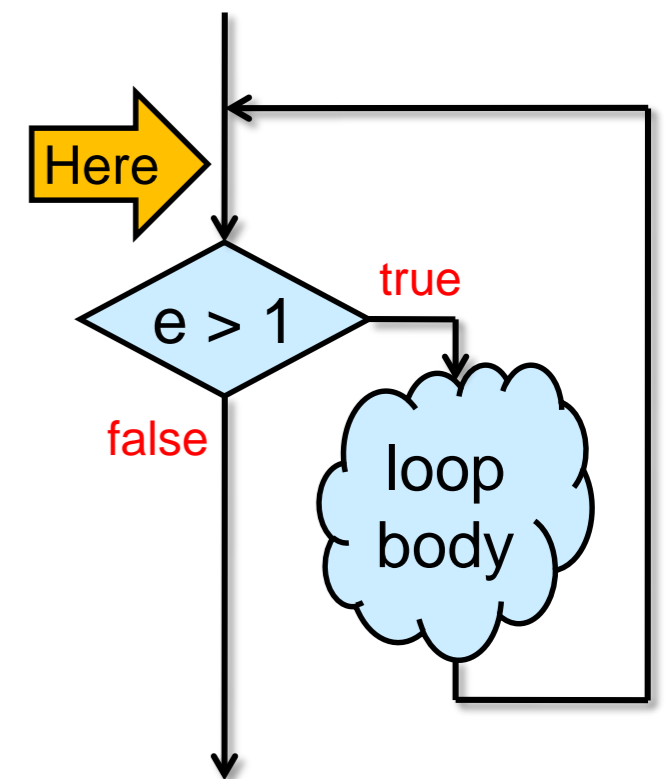
b	e	r
2	8	1
4	4	1
16	2	1
256	1	1

At this point we exit the loop

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1) {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

This checks if e is odd

- Can we spot a quantity that doesn't change?



# Tracing Code

- Trying with  $f(2,8)$

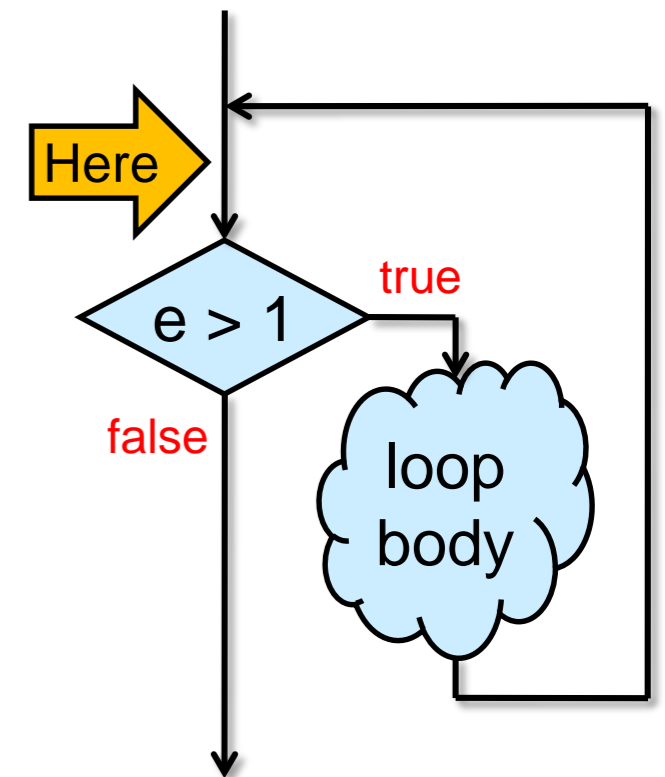
- Can we spot a quantity that doesn't change?
- $b^e$  is always 256

b	e	r	$b^e$
2	8	1	256
4	4	1	256
16	2	1	256
256	1	1	256

- This is a **candidate loop invariant**

- $b^e$  is constant on one set of inputs
- a loop invariant must stay constant on all inputs

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1) {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```



# Tracing Code

- $b^e$  is a *candidate* loop invariant

- Let's try with  $f(2,7)$

b	e	r	$b^e$
2	7	1	128
4	3	2	64
16	1	8	8

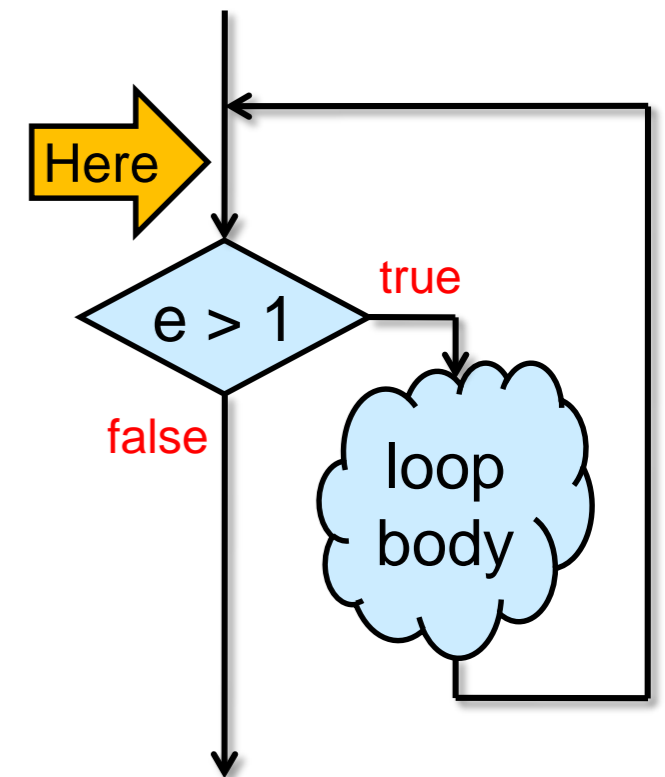
Not constant  
on these  
inputs

- $b^e$  is **not** invariant on these inputs!

➤ It was a candidate that didn't pan out

- Can we spot another quantity that doesn't change?

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1) {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```



# Tracing Code

- Trying with  $f(2,7)$

- Can we spot a quantity that doesn't change?
- $b^e * r$  is always 128

b	e	r	$b^e$	$b^e * r$
2	7	1	128	<b>128</b>
4	3	2	64	<b>128</b>
16	1	8	8	<b>128</b>

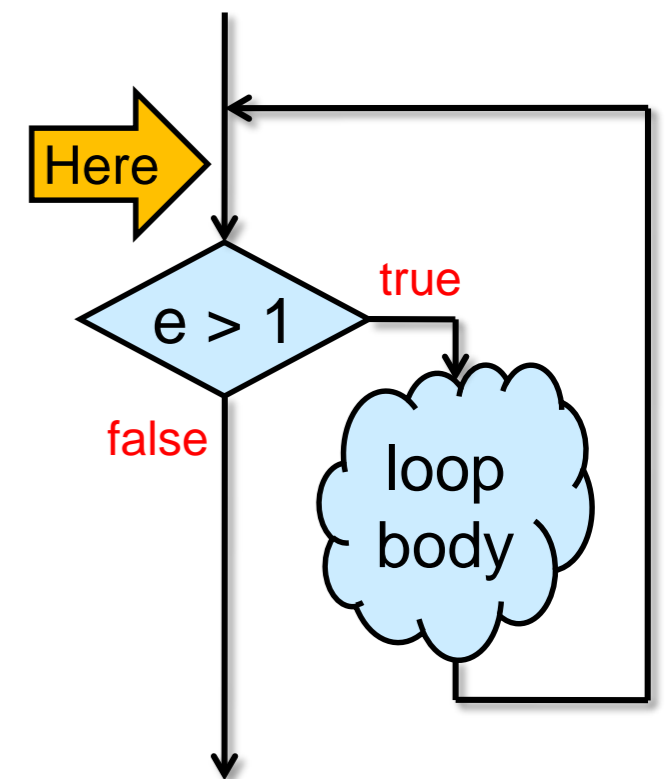
- This is another candidate loop invariant

- Let's test it on  $f(3,5)$

b	e	r	$b^e * r$
3	5	1	<b>243</b>
9	2	3	<b>243</b>
81	1	3	<b>243</b>

- This seems to work

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1) {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```



# A Candidate Loop Invariant

- $b^e * r$  is a promising candidate loop invariant

- It works on *three* inputs!

- How do we know it works in general?

- We can't test it on all inputs

- We need to provide a **proof**

- *But first, let's add it to our code*

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1) {
        if (e % 2 == 1) {
            r = b * r;
        }
        b = b * b;
        e = e / 2;
    }
    return r * b;
}
```

# Loop Invariants in C0

- In C0, we use the directive

`//@loop_invariant`

to specify a loop invariant

C0 keyword to specify a loop invariant  
• written between the loop guard and the loop body

- Then, simply write

`//@loop_invariant POW(b, e) * r;`

○ ... this won't work

- C0 would need to keep track of the values of this expression across all iterations of the loop
- also, what if the loop runs 0 times?

- In C0, loop invariants must be **boolean expressions**

○ **true** means it was satisfied in the current iteration

○ **false** means it wasn't

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1)
  //@loop_invariant ... ;
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

# Loop Invariants in C0

- They are boolean expressions

- **true** means satisfied

- What can we use?

b	e	r	$b^e * r$
2	7	1	128
4	3	2	128
16	1	8	128

- As we enter the loop,  
b is x and e is y

- so  $x^y$  is 128 too

- thus,  $b^e * r = x^y$

- Then, we can write

```
//@loop_invariant POW(b, e) * r == POW(x, y);
```

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1)
    //@loop_invariant POW(b,e) * r == POW(x,y);
    {
      if (e % 2 == 1) {
        r = b * r;
      }
      b = b * b;
      e = e / 2;
    }
  return r * b;
}
```

Execution will abort  
when ran with **-d**  
if LI is ever **false**

# Safety

We have two new calls to **POW**

○ **Are they safe?**

● **POW(x, y)**

➤ **To show:**  $y \geq 0$

○  $y \geq 0$  by line 2 (precondition of **f**)



● **POW(b, e)**

➤ **To show:**  $e \geq 0$

○ “e is *initially* equal to y which is  $\geq 0$  and it is halved at *each* iteration of the loop so e is *always*  $\geq 0$ ”

○ This is an example of **operational reasoning**

➤ The justification relies on what is happening in all the iterations of the loop

□ This is error-prone

➤ We will disallow safety proofs based on operational reasoning on loops



```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant POW(b,e) * r == POW(x,y);
10.    {
11.      if (e % 2 == 1) {
12.        r = b * r;
13.      }
14.      b = b * b;
15.      e = e / 2;
16.    }
17.   return r * b;
18. }
```



# Safety

## POW(b, e)

➤ To show:  $e \geq 0$

○ We can sort of do it with operational reasoning

➤ error prone!

○ but we really want to prove it using point-to reasoning

● We do believe that  $e \geq 0$  at every iteration of the loop

○ Turn it into a candidate loop invariant!

`//@loop_invariant e >= 0;`

➤ We will need to prove later that it is valid

○ Then we prove that `POW(b, e)` is safe by pointing to line 9

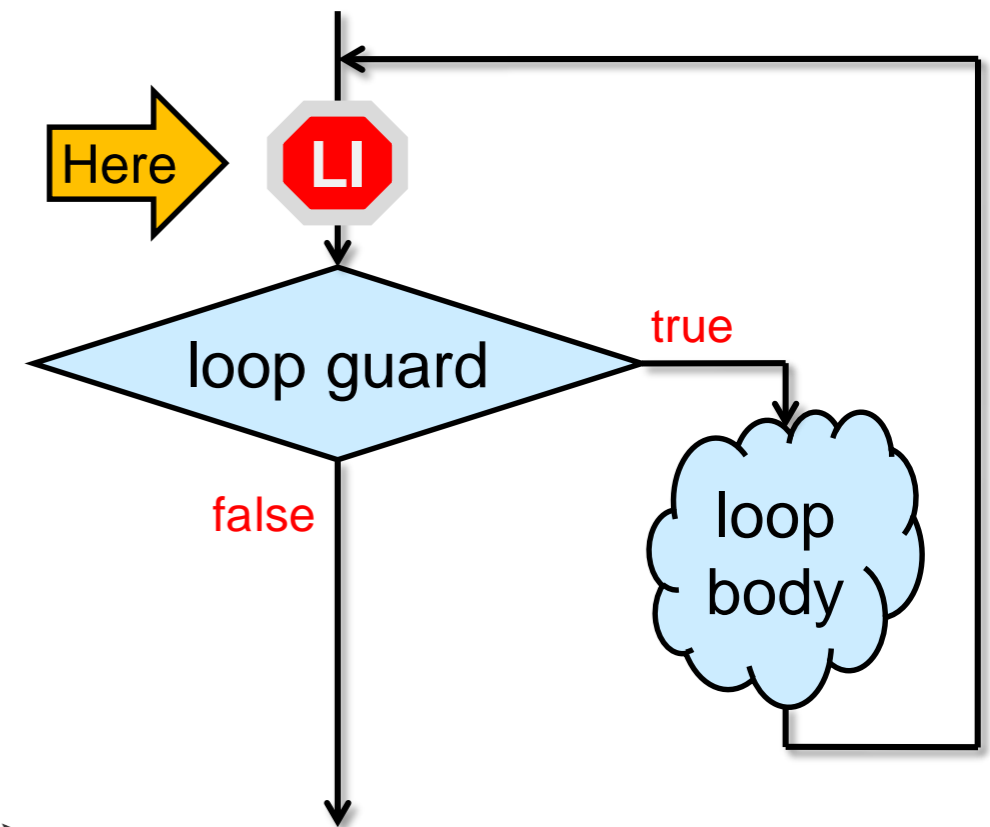


```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.      if (e % 2 == 1) {
13.        r = b * r;
14.      }
15.      b = b * b;
16.      e = e / 2;
17.    }
18.   return r * b;
19. }
```

An operational hunch is often a good candidate loop invariant

# How Loop Invariants Work

- Loop invariants are checked **just before** the loop guard is tested
- If the loop runs  $n$  times,
  - the loop invariant is checked  $n+1$  times
    - must be **true** all  $n+1$  times
  - the loop guard is tested  $n+1$  times too
    - **true** the first  $n$  times and **false** the last time
- When we exit the loop
  - the loop invariant is **true**
  - the loop guard **false**



Note that  $n$  could be 0

Important!

# **Validating Loop Invariants**

# Where are we?

- We have learned even more about **f**
  - The contracts tell us what it is meant to do
  - The loop invariants give us useful information about how the loop works
    - but these are **candidate** loop invariants
    - we need to prove that they are valid
- We have started learning about proving things about code
  - just safety so far
  - point-to reasoning: **good**
  - operational reasoning: **error prone**

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.     int b = x;
6.     int e = y;
7.     int r = 1;
8.     while (e > 1)
9.         //@loop_invariant e >= 0;
10.        //@loop_invariant POW(b,e) * r == POW(x,y);
11.        {
12.            if (e % 2 == 1) {
13.                r = b * r;
14.            }
15.            b = b * b;
16.            e = e / 2;
17.        }
18.        return r * b;
19. }
```

# Proving a Loop Invariant Valid

- We cannot show a loop invariant is valid by running it on all possible inputs

- We need to supply a proof
  - using point-to reasoning

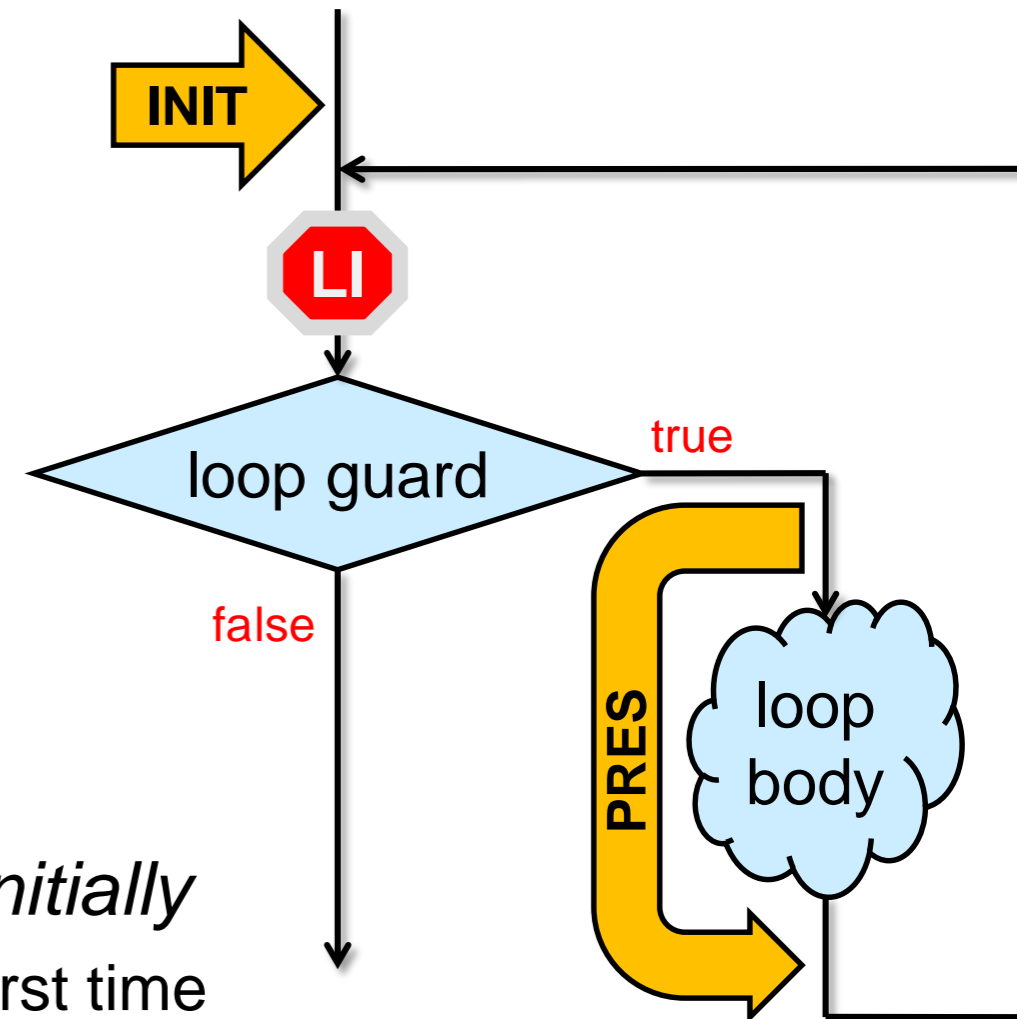
- Two steps

**INIT:** show that the loop invariant is true *initially*

- just before we test the loop guard the very first time

**PRES:** show that the loop invariant is **preserved** by the loop

- if it is true at the beginning of an **arbitrary iteration** of the loop,
- then it is also true at the end of this iteration



But it may become false temporarily in the middle of the loop body

We use math notation for brevity

# Validity of $e \geq 0$

## INIT:

➤ To show:  $e \geq 0$  initially

A.  $y \geq 0$  by line 2

B.  $e = y$  by line 6

C.  $e \geq 0$  by math on A and B

This is a typical proof format in this course



```

1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.      if (e % 2 == 1) {
13.        r = b * r;
14.      }
15.      b = b * b;
16.      e = e / 2;
17.    }
18.    return r * b;
19. }

```

## PRES:

LI at **start** of current iteration

LI at **end** of current iteration

➤ To show: if  $e \geq 0$ , then  $e \geq 0$

But isn't this trivially true?

- The value of  $e$  changes in the body of the loop
- We need a way to distinguish the value at the start and end of the current iteration

➤  $e$  ← value of  $e$  at the **start** of the current iteration

➤  $e'$  ← value of  $e$  at the **end** of the current iteration

# Validity of $e \geq 0$

**INIT:**  $e \geq 0$  initially ✓

**PRES:**

LI at **start** of current iteration

LI at **end** of current iteration

➤ To show: if  $e \geq 0$ , then  $e' \geq 0$

A.  $e \geq 0$  by assumption

B.  $e/2 \geq 0$  by math on A

C.  $e' = e/2$  by line 16

D.  $e' \geq 0$  by B and C ✓

Both INIT and PRES were proved by point-to reasoning

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.      if (e % 2 == 1) {
13.        r = b * r;
14.      }
15.      b = b * b;
16.      e = e / 2;
17.    }
18.   return r * b;
19. }
```

# Validity of $b^e r = x^y$

## INIT:

➤ To show:  $b^e r = x^y$  initially

- A.  $b = x$  by line 5
- B.  $e = y$  by line 6
- C.  $r = 1$  by line 7
- D.  $b^e r = x^y$  by math on A, B, C ✓

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.      if (e % 2 == 1) {
13.        r = b * r;
14.      }
15.      b = b * b;
16.      e = e / 2;
17.    }
18.   return r * b;
19. }
```

## PRES:

LI at **start** of current iteration

LI at **end** of current iteration

x and y don't change in the loop

➤ To show: if  $b^e r = x^y$ , then  $b'^e r' = x^y$

○ We need to distinguish 2 cases based on the test  $e \% 2 == 1$

- $e \% 2 == 1$  is true —  $e$  is odd
- $e \% 2 == 1$  is false —  $e$  is even



# Validity of $b^e r = x^y$

## PRES:

➤ To show: if  $b^e r = x^y$ , then  $b^{e'} r' = x^y$

➤ Case **e is odd** ( $e \% 2 == 1$ )

□ Then  $e = 2n+1$  for some  $n$

- A.  $b' = b*b$  by line 15
- B.  $e' = e/2$  by line 16
- C.  $= n$  by case assumption and math
- D.  $r' = b * r$  by line 13
- E.  $b^{e'} r' = (b*b)^n b*r$  by A, B, C, D
- F.  $= b(b^2)^n r$  by math
- G.  $= b^{2n+1} r$  by math
- H.  $= b^e r$  by case assumption
- I.  $= x^y$  by assumption

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.      if (e % 2 == 1) {
13.        r = b * r;
14.      }
15.      b = b * b;
16.      e = e / 2;
17.    }
18.   return r * b;
19. }
```

This is one of the most complex proofs in this course

○ This proves the first case

# Validity of $b^e r = x^y$

## PRES:

➤ To show: if  $b^e r = x^y$ , then  $b^{e'} r' = x^y$

➤ Case **e is even** ( $e \% 2 == 0$ )

□ Then  $e = 2n$  for some  $n$

- A.  $b' = b * b$  by line 15
- B.  $e' = e / 2$  by line 16
- C.  $= n$  by case assumption and math
- D.  $r' = r$  since  $r$  is unchanged
- E.  $b^{e'} r' = (b * b)^n r$  by A, B, C, D
- F.  $= (b^2)^n r$  by math
- G.  $= b^{2n} r$  by math
- H.  $= b^e r$  by case assumption
- I.  $= x^y$  by assumption

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.      if (e % 2 == 1) {
13.        r = b * r;
14.      }
15.      b = b * b;
16.      e = e / 2;
17.    }
18.   return r * b;
19. }
```

**PRES** holds  
for  $b^e r = x^y$



o This proves the second case too

# Loop Invariants

- $e \geq 0$  is valid ✓

- it holds **INIT**ially

- it is **PRE**Served by an arbitrary iteration of the loop

- if  $e \geq 0$ , then  $e' \geq 0$

- $b^e r = x^y$  is valid ✓

- it holds **INIT**ially

- it is **PRE**Served by an arbitrary iteration of the loop

- if  $b^e r = x^y$ , then  $b^{e'} r' = x^y$

- This shows that both are **genuine loop invariants**

- not just candidates

- we can forget about the body of the loop when reasoning about this function

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.     int b = x;
6.     int e = y;
7.     int r = 1;
8.     while (e > 1)
9.         //@loop_invariant e >= 0;
10.        //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.        if (e % 2 == 1) {
13.            r = b * r;
14.        }
15.        b = b * b;
16.        e = e / 2;
17.    }
18.    return r * b;
19. }
```

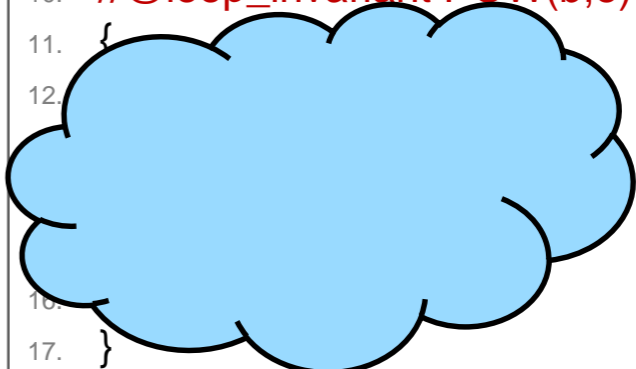
# **Proof-directed Debugging**

# Where are we?

- The contracts tell us what the function is *meant to do*
  - but we know there is a bug in there
- The loop invariants abstract away the details of the loop

But what to do with them is still a bit mysterious

- *Let's find the bug!*

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.     int b = x;
6.     int e = y;
7.     int r = 1;
8.     while (e > 1)
9.         //@loop_invariant e >= 0;
10.        //@loop_invariant POW(b,e) * r == POW(x,y);
11.        {
12.            
13.            // ...
14.            // ...
15.            // ...
16.            // ...
17.        }
18.     return r * b;
19. }
```

# After the Loop

- What do we know when execution exits the loop?

- the loop guard is **false**

- $e \leq 1$

- the loop invariants are **true**

- $e \geq 0$

- $b^e r = x^y$

- From  $e \leq 1$  and  $e \geq 0$ , we have that

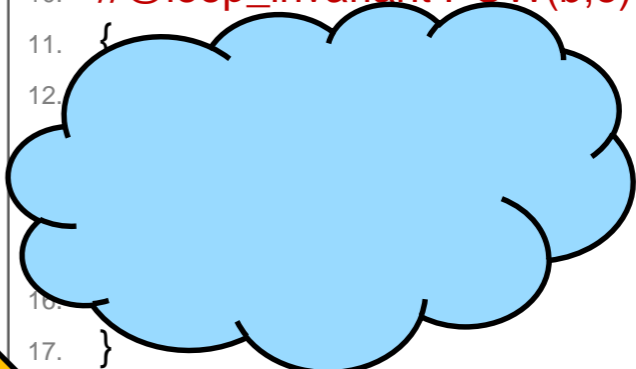
- either  $e = 0$

- or  $e = 1$

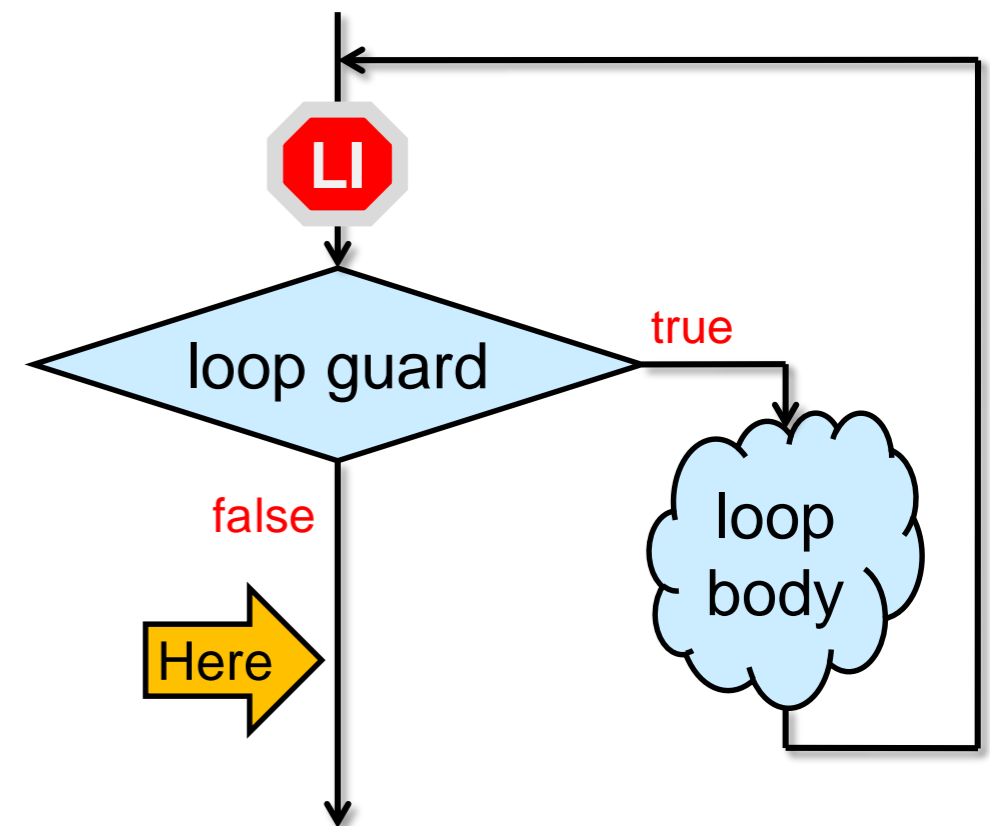
as we exit the loop

Recall that  $e$  has type `int`

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.
13.
14.
15.
16.
17.    }
18.   return r * b;
19. }
```



Here

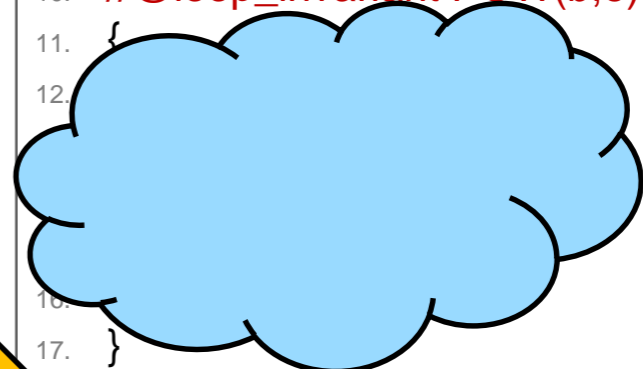


# After the Loop

- Either  $e = 0$  or  $e = 1$

- Let's plug these values in the other loop invariant,  $b^e r = x^y$

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   while (e > 1)
9.     //@loop_invariant e >= 0;
10.    //@loop_invariant POW(b,e) * r == POW(x,y);
11.    {
12.    }
13.
14.
15.
16.
17. }
18. return r * b;
19. }
```



→ If  $e = 1$ , then  $x^y = b^e r = b^1 r = r b$

- Thus,  $x^y = r b$  in this case

This is exactly what `f` returns.



→ if  $e = 0$ , then  $x^y = b^e r = b^0 r = r$

- Thus,  $x^y = r$  in this case

- $x^y \neq r b$

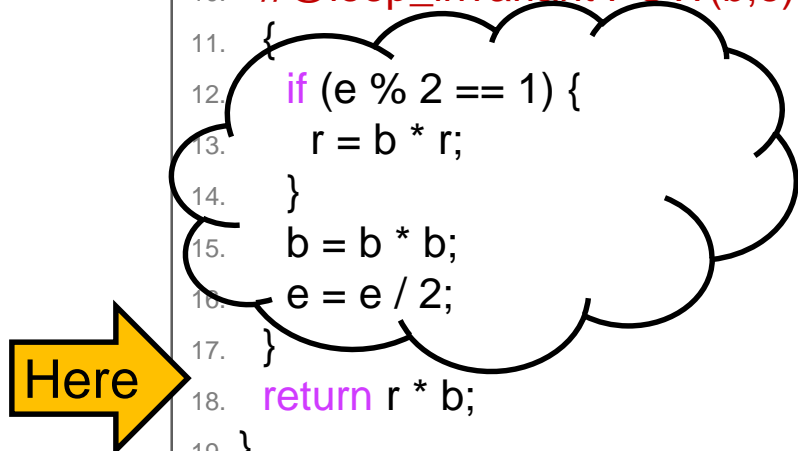
This is **not** what `f` returns.  
**This is the bug!**



# Tracking the Bug

- The bug is when  $e = 0$  as we exit the loop
- This can happen **only** if  $f$  is called with 0 as  $y$ 
  - if  $e = 1$ , the loop doesn't run and  $e$  stays 1
  - if  $e > 1$  at the start of an iteration, then  $e' \geq 1$  as we end it

```
1. int f(int x, int y)
2. // @requires y >= 0;
3. // @ensures \result == POW(x,y);
4. {
5.     int b = x;
6.     int e = y;
7.     int r = 1;
8.     while (e > 1)
9.         // @loop_invariant e >= 0;
10.        // @loop_invariant POW(b,e) * r == POW(x,y);
11.        {
12.            if (e % 2 == 1) {
13.                r = b * r;
14.            }
15.            b = b * b;
16.            e = e / 2;
17.        }
18.     return r * b;
19. }
```





# Fixing the Bug

## Idea #1: return 1 if $y = 0$

- This works but it introduces a **special case** in the code
- Special cases leads to contrived, unmaintainable code
  - sometimes unavoidable
  - but let's see if we can do better

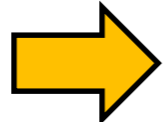
```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  if (y == 0) return 1;
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

# Fixing the Bug

**Idea #2:** change the precondition to  $y > 0$

- This forces the caller to have special cases in their code!

○ calls to `f` need to be **guarded**

`int c = f(a, b)`  `int c = 1;`  
`if (b > 0) c = f(a, b);`

```
int f(int x, int y)
//@requires y > 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 1)
        //@loop_invariant e >= 0;
        //@loop_invariant POW(b,e) * r == POW(x,y);
        {
            if (e % 2 == 1) {
                r = b * r;
            }
            b = b * b;
            e = e / 2;
        }
    return r * b;
}
```

- This also means that `f` is not the power function any more
  - undefined when exponent is 0

- Not a great solution



# Fixing the Bug

**Idea #3:** forget about **f** and use **POW** instead

- Recall the trace of  $f(2,8)$ 
  - the loop ran 4 times
- Trace  $POW(2, 8)$ 
  - 9 recursive calls
- **f** is much more efficient



b	e	r
2	8	1
4	4	1
16	2	1
256	1	1

x	y
2	8
2	7
2	6
2	5
2	4
2	3
2	2
2	1
2	0

```
int POW(int x, int y)
//@requires y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1) * x;
}

int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

# Fixing the Bug

**Idea #4:** make `f` return only when `e = 0`

o change the loop guard to `e > 0`

➤ the loop always end with `e = 0`

o return `r` instead of `r * b`

➤ that's what we had to return when `e = 0`

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 0)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r;
}
```

No special cases!



Rather than getting rid of the **bad** case (`e = 0`), we make it the **good** case and do away with the other case (`e = 1`)

How's this for a movie plot?

**Correctness**

# Did we Really Fix the Bug?

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 0)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  return r;
}
```

- The loop invariants are still valid
  - we didn't change the body of the loop
  - we changed the loop guard
    - but it doesn't impact the validity proof

Check for yourself

- Right after the loop, we know that
  - the loop guard is **false**:  $e \leq 0$
  - the 1<sup>st</sup> loop invariant is **true**:  $e \geq 0$
  - the 2<sup>nd</sup> loop invariant is **true**:  $b^e r = x^y$

} so  $e = 0$

➤ so  $x^y = b^e r = b^0 r = r$

This is what  $f$  returns now



# Assertions

Right after the loop, we know that  $e = 0$

- We can note this with the directive

```
//@assert e == 0;
```

- checked only when running with **-d**
- aborts execution if the test is **false**

- **//@assert** is a great way to note

- intermediate steps of reasoning
- expectations about execution

- These are all the run-time directives of C0

```
//@requires, //@ensures, //@loop_invariant, //@assert
```

There are no others

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 0)
        //@loop_invariant e >= 0;
        //@loop_invariant POW(b,e) * r == POW(x,y);
        {
            if (e % 2 == 1) {
                r = b * r;
            }
            b = b * b;
            e = e / 2;
        }
    //@assert e == 0;
    return r;
}
```

**//@assert** can appear  
anywhere a statement  
is expected

# Is the Function Correct?

**Correctness:** for any input that satisfies the preconditions, the postconditions will be true

- We just proved that, as we exit the loop,  $r = x^y$ 
  - just before `return r;`
- This tells us that `f` will **never return the wrong result**
- but will it *always return the right result?*

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 0)
        //@loop_invariant e >= 0;
        //@loop_invariant POW(b,e) * r == POW(x,y);
        {
            if (e % 2 == 1) {
                r = b * r;
            }
            b = b * b;
            e = e / 2;
        }
    return r;
}
```



# Is the Function Correct?

**Correctness:** for any input that satisfies the preconditions, the postconditions will be true

- Can a function **never return the wrong result** and yet not necessarily ***always return the right result***?
  - Let's empty out the loop body in our example

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 0)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {}
  return r;
}
```

This is legal  
C0 code

The loop invariants are valid

- **INIT** is unchanged
- **PRES** holds trivially

If execution were to reach **return r**,

- **e == 0** would have to be **true**
- **r** would have to contain  $x^y$

But it never reaches **return r**!  
So the postcondition will never be **true**

This code is **not** correct.

- ... only if it never returns
  - if the loop runs for ever

# Termination

- We need to have a reason to believe the loop terminates

- it doesn't run for ever

- Here's a proof of termination

- *as the loop runs,*

- e gets strictly smaller and it can never become smaller than 0*

- so the loop must terminate



```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    //@loop_invariant POW(b,e) * r == POW(x,y);
    {
      if (e % 2 == 1) {
        r = b * r;
      }
      b = b * b;
      e = e / 2;
    }
  //@assert e == 0;
  return r;
}
```

This is an **operational** proof:  
we are not pointing to anything


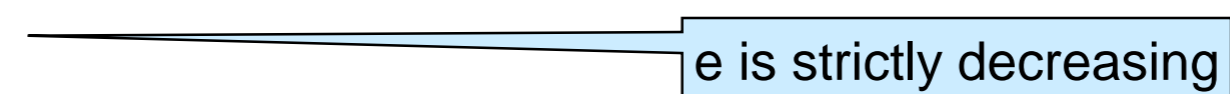
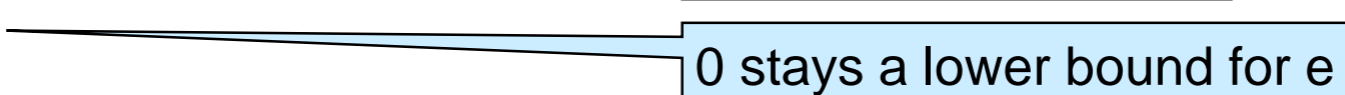
# Termination

- Operational proof

- *as the loop runs, e gets strictly smaller and it can never become smaller than 0*
- so the loop must terminate

- Can we prove it using point-to-reasoning?

- Yes! Here's what we need to show
- in an arbitrary iteration of the loop,

- if  $e \geq 0$ ,  0 is a lower bound for e
- then  $e' < e$   e is strictly decreasing
- and  $e' \geq 0$   0 stays a lower bound for e

if e starts  $\geq 0$ ,  
it gets strictly smaller and  
can never become smaller than 0

```
int f(int x, int y)
  //@requires y >= 0;
  //@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 0)
    //@loop_invariant e >= 0;
    //@loop_invariant POW(b,e) * r == POW(x,y);
    {
      if (e % 2 == 1) {
        r = b * r;
      }
      b = b * b;
      e = e / 2;
    }
  //@assert e == 0;
  return r;
}
```

# Termination

- Point-to proof

➤ **To show:** if  $e \geq 0$ , then  $e' < e$  and  $e' \geq 0$

A.  $e > 0$  by line 8 (loop guard)

B.  $e' = e/2$  by line 16

C.  $e' < e$  by math

D.  $e' \geq 0$  by math



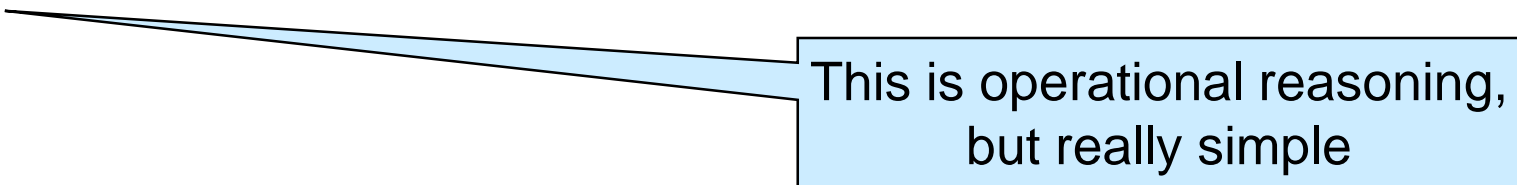
```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == POW(x,y);
4. {
5.     int b = x;
6.     int e = y;
7.     int r = 1;
8.     while (e > 0)
9.         //@loop_invariant e >= 0;
10.        //@loop_invariant POW(b,e) * r == POW(x,y);
11.        {
12.            if (e % 2 == 1) {
13.                r = b * r;
14.            }
15.            b = b * b;
16.            e = e / 2;
17.        }
18.        //@assert e == 0;
19.        return r;
20.    }
```

However,  
for termination proofs,  
we will generally be Ok with an operational argument

# **Reasoning about Code**

# Reasoning about C0

- C0 programs have a precise behavior
  - we can reason about them mathematically
- We used two types of reasoning
  - **Operational reasoning:** drawing conclusions about how things change when certain lines of code are executed
  - **Point-to reasoning:** drawing conclusions about what we know to be true by pointing to specific lines of code that justify them
    - boolean expressions
    - basic mathematical properties
    - variable assignments



This is operational reasoning,  
but really simple

# Operational Reasoning

- Examples

- Value of variables right after an assignment ✓
- Things happening in the body of a loop from outside this loop ✗
- Things happening in the body of a function being called ✗
- Previously true statements after variables in it have changed ✗

- Operational reasoning is hard to do right consistently

- very error prone!

- We want to stay away from anything beyond simple assignments

- except in termination proofs

If a proof about loops uses words like “**always**”, “**never**”, “**each**”, you are doing operational reasoning

But operational intuitions are a good way to form conjectures that we can then prove using point-to reasoning

# Point-to Reasoning

- Examples

- Boolean conditions

- condition of an **if** statement in the “then” branch
    - negation of the condition of an **if** statement in the “else” branch
    - loop guard inside the body of a loop
    - negation of the loop guard after the loop

- Contract annotations

- preconditions of the current function
    - postconditions of a function just called
    - loop invariant inside the loop body
    - loop invariant after the loop
    - earlier fully justified assertions

- Math

- laws of logic
    - some laws of arithmetic

- Value of variables right after an assignment





# Safety

- The inputs of a function call satisfy the function's preconditions
  - we will generalize this definition in the future

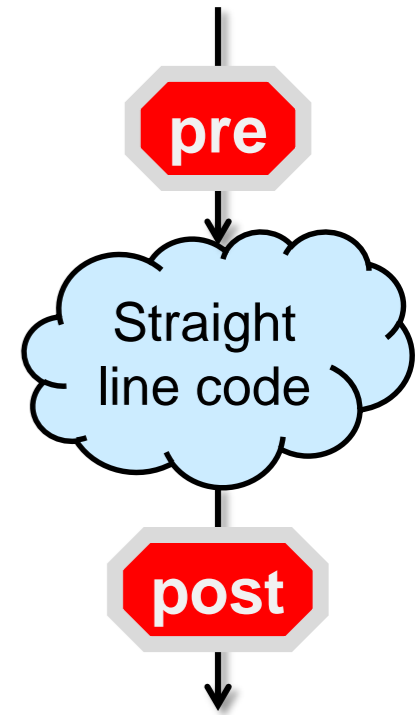
***We will exclusively use point-to reasoning to justify safety***

# Correctness

- The postconditions of a function will be true on any call that satisfies the preconditions
  - We will not need to generalize this definition

# Straight Line Functions

A non-recursive function without loops



- Proving correctness amounts to combining assignments

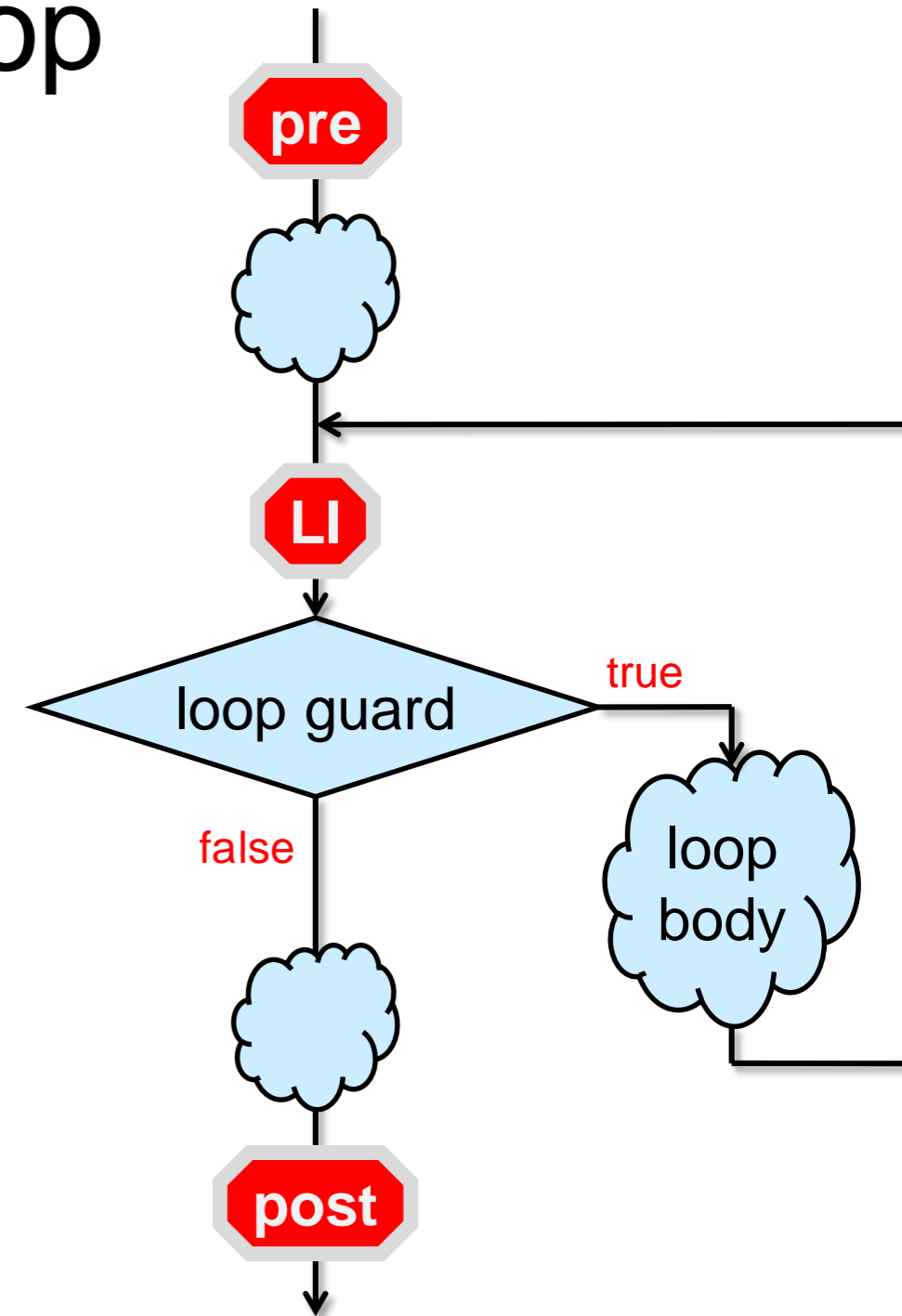
➤ To show:  $\backslash result = x$

- A.  $b = x$  by line 5
- B.  $r = 1$  by line 7
- C.  $\backslash result = r * b$  by line 8
- D.  $r * b = x$  by math on A, B, C

```
1. int f(int x, int y)
2. //@requires y >= 0;
3. //@ensures \result == x;
4. {
5.   int b = x;
6.   int e = y;
7.   int r = 1;
8.   return r * b;
9. }
```

# Functions with One Loop

- Proving correctness involves 3 steps
  - Show that the loop invariants are *valid*
    - **INIT**: the LI are true initially
    - **PRES**: the LI are preserved by an arbitrary iteration of the loop
  - **EXIT**: the LI and the negation of the loop guard imply the postcondition
  - **TERM**: the loop terminates



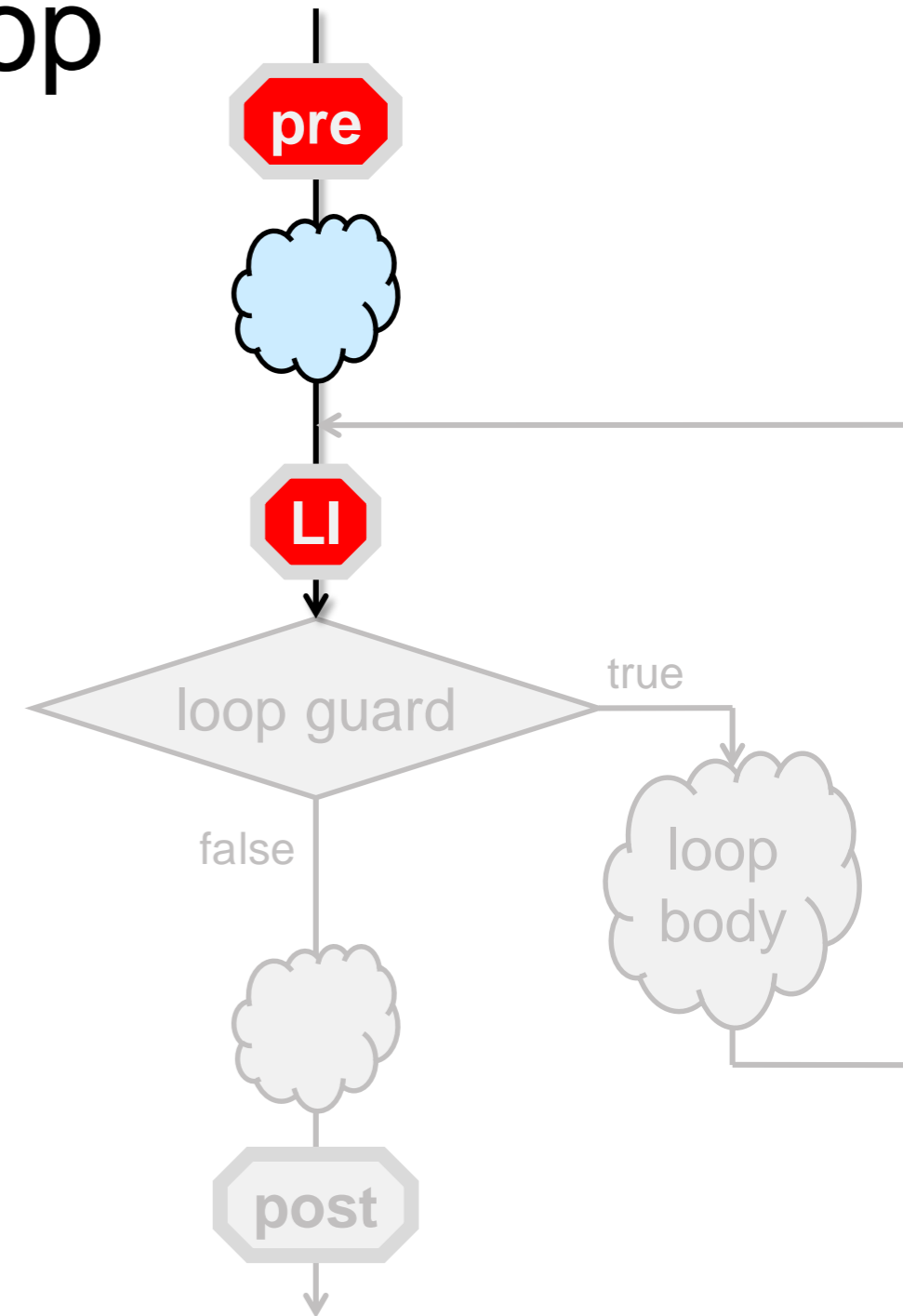
That's exactly what we did for our mystery function

These steps can be proved in any order

# Functions with One Loop

**INIT:** the loop invariant is true initially

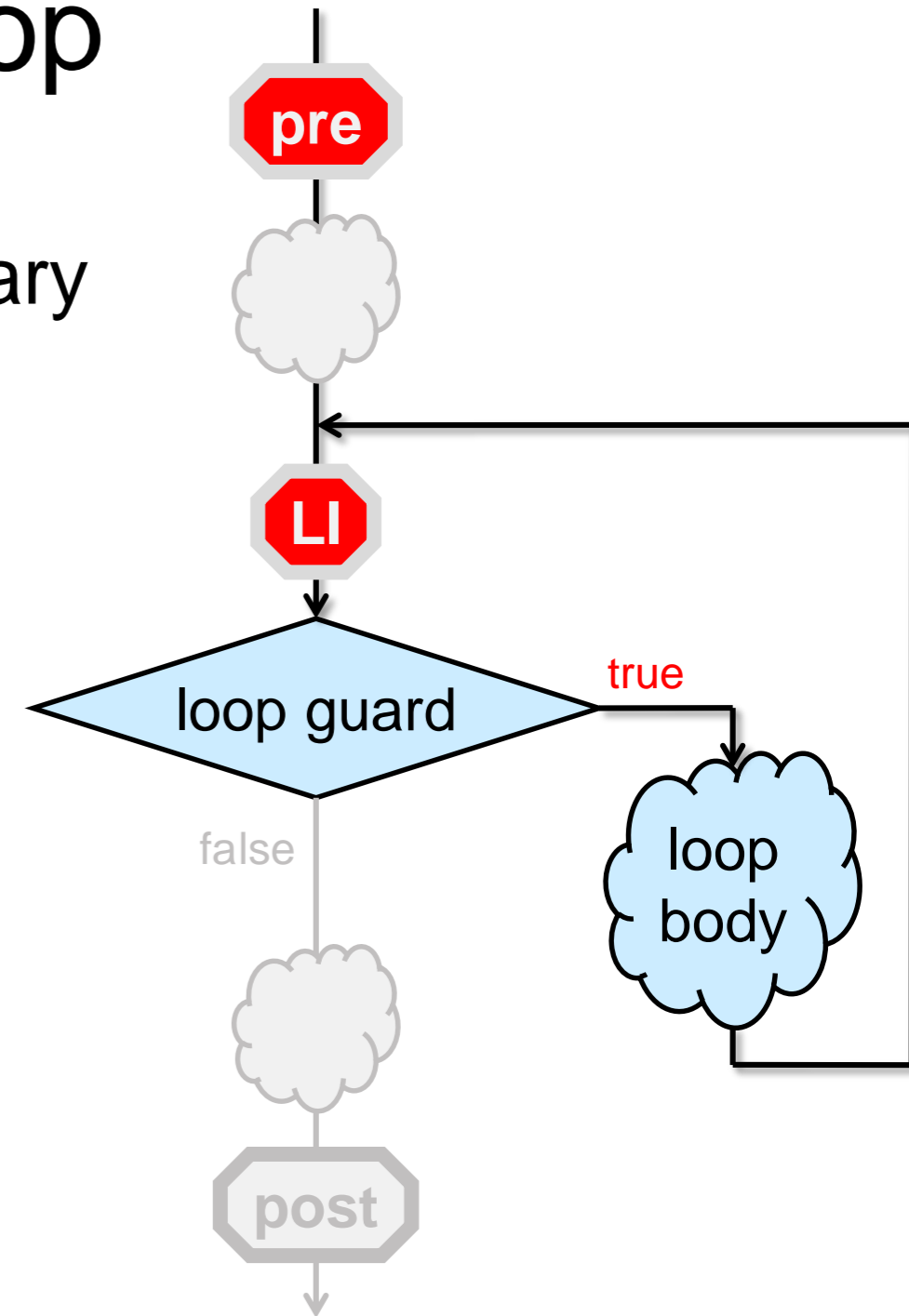
- proved by point-to reasoning  
typically using
  - the preconditions
  - simple assignments before the loop



# Functions with One Loop

**PRES:** the LI are preserved by an arbitrary iteration of the loop

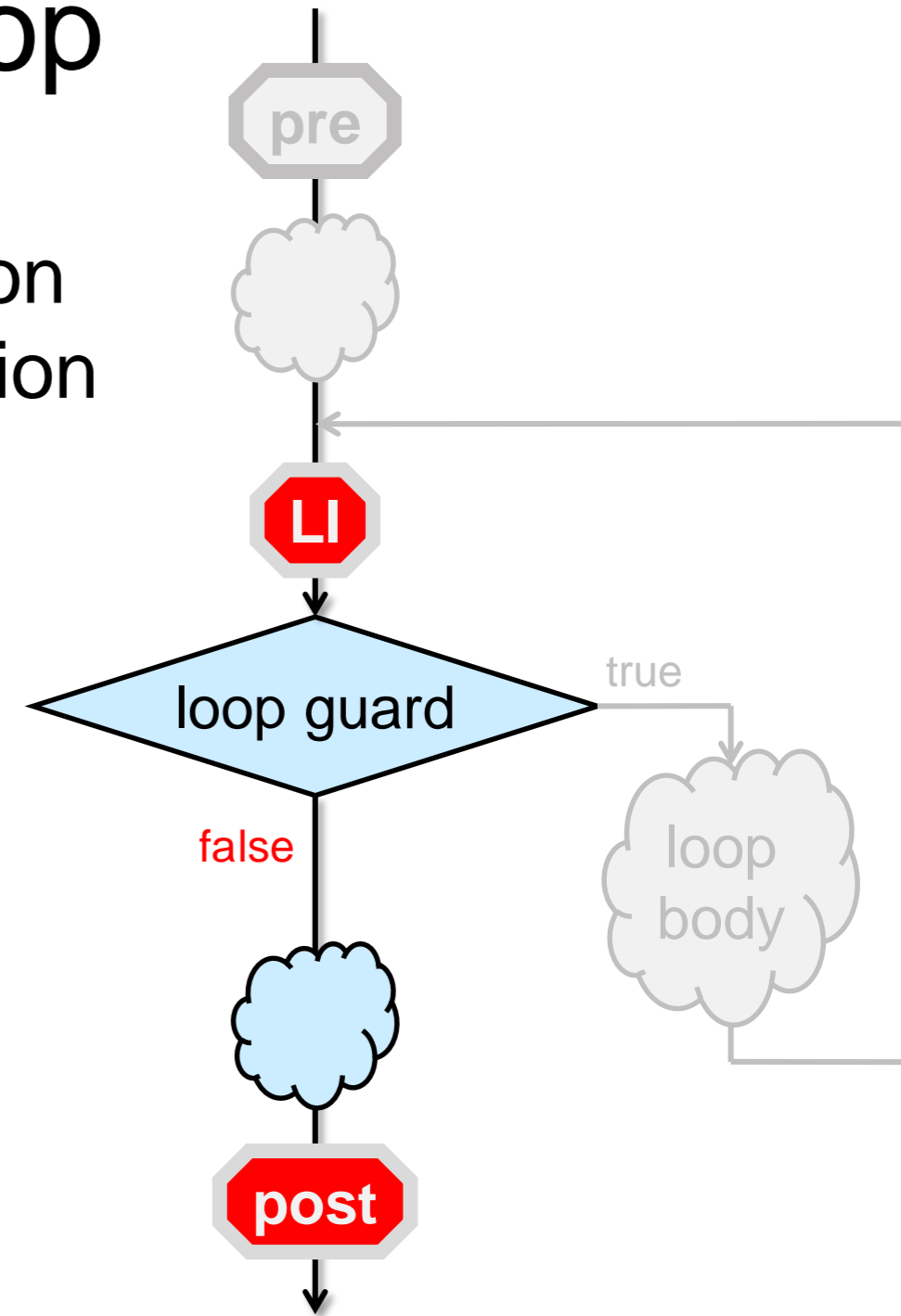
- proved by point-to reasoning typically using
  - the assumption that the LI is true at the beginning of the iteration
  - the loop guard
  - simple assignments and conditionals in the loop body
  - the preconditions (sometimes)



# Functions with One Loop

**EXIT:** the loop invariants and the negation of the loop guard imply the postcondition

- proved by point-to reasoning typically using
  - the loop invariant
  - the negation of the loop guard
  - simple assignments and conditionals after the loop

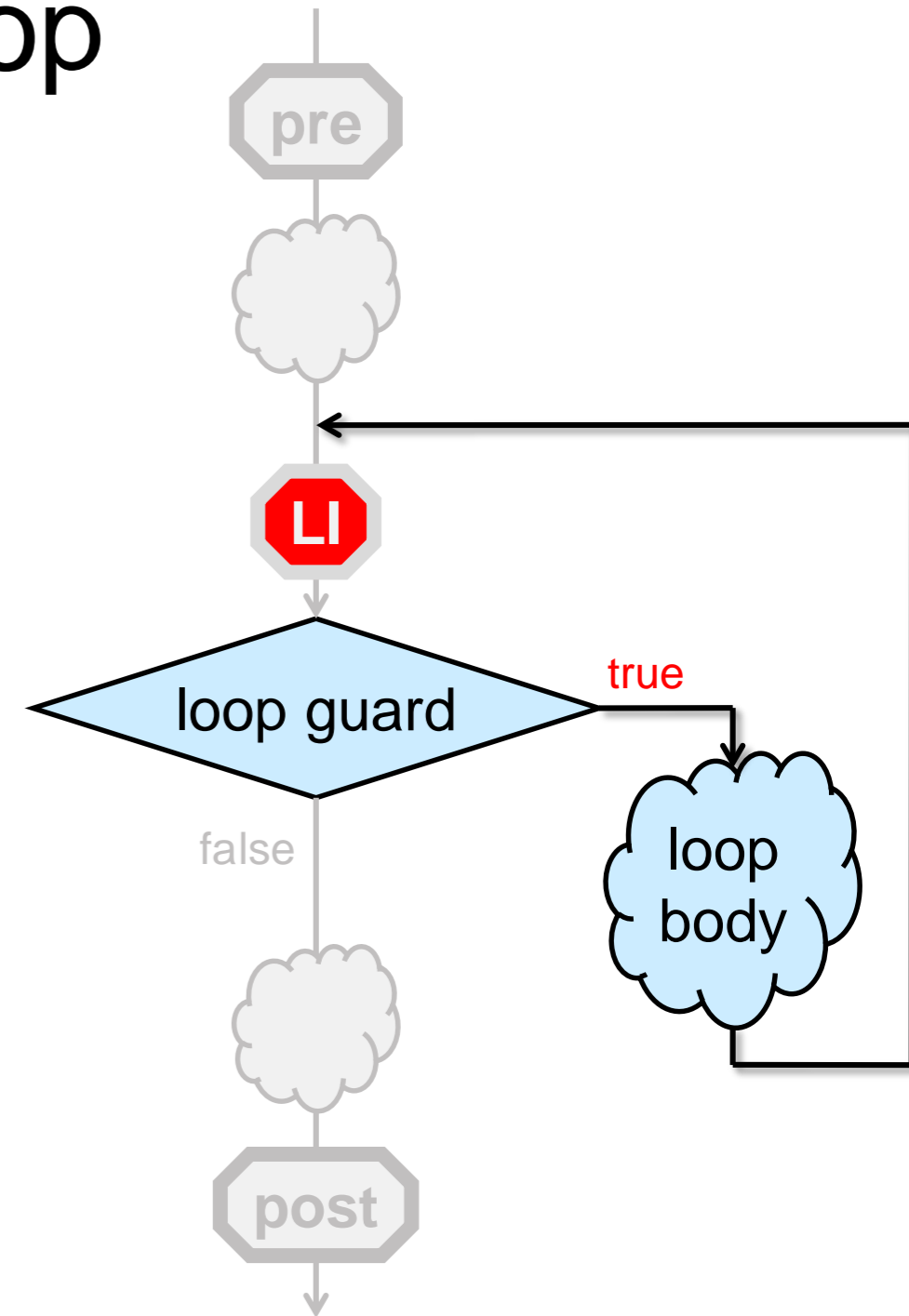


# Functions with One Loop

**TERM:** the loop terminates

- proved by operational reasoning typically using
  - the assumption that the LI is true at the beginning of the iteration
  - the loop guard
  - simple assignments and conditionals in the loop body

But it can also be proved by **point-to reasoning**



# Functions with One Loop

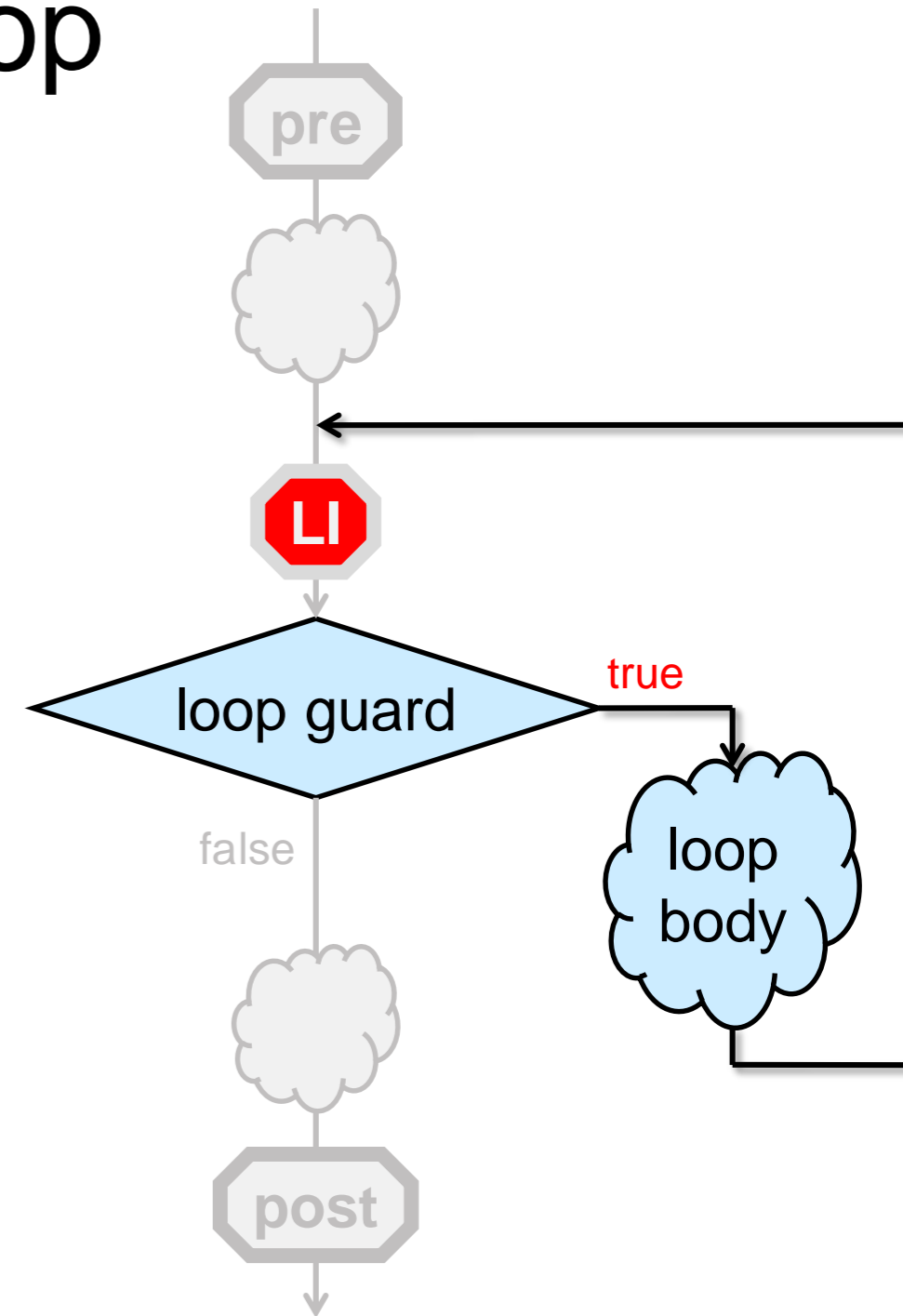
**TERM:** the loop terminates

- Format of a termination proof using operational reasoning

*“on an arbitrary iteration of the loop, the quantity \_\_\_\_\_ gets strictly smaller but it can’t ever get smaller than \_\_\_\_\_”*

**or**

*“on an arbitrary iteration of the loop, the quantity \_\_\_\_\_ gets strictly bigger but it can’t ever get bigger than \_\_\_\_\_”*



A **quantity** may be an **expression**, not necessarily a variable



# More Complex Functions

- These techniques can be extended
  - but we will rarely deal with functions with more than one loop
- We can also factor out nested loops and the like into helper functions
  - and then use the technique we just saw

# Seriously??

- *All these proofs and complicated reasoning seem overkill!*
  - *the mystery function wasn't all that hard after all*
  - *we could just spot what was going on*
- Yes, but it won't be that easy for more complex functions
  - the technique we saw is **systematic** and **scalable**
  - reasoning about code will pay off
- Point-to reasoning is what we do in our head all the time when programming
  - writing it down as loop invariants and contracts makes it easier not to get confused
  - and the **-d** flag will catch lingering issues at run time

# Epilogue

# Where are we?

- We fully documented **f**
  - function contracts
  - loop invariants
  - key assertions
- We fixed the bug
- We gave mathematical proofs that
  - all the calls it makes are safe
  - it is correct
- Let's enjoy the fruit of our labor with some more testing!

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
    int b = x;
    int e = y;
    int r = 1;
    while (e > 0)
        //@loop_invariant e >= 0;
        //@loop_invariant POW(b,e) * r == POW(x,y);
        {
            if (e % 2 == 1) {
                r = b * r;
            }
            b = b * b;
            e = e / 2;
        }
        //@assert e == 0;
    return r;
}
```

# Sanity Checks

- Let's do a last round of testing

Linux Terminal

```
# coin -d mystery.c0
C0 interpreter (coin) ...
--> f(2, 0);
1 (int)
--> f(2, 1);
2 (int)
--> f(2, 7);
128 (int)
--> f(2, 8);
256 (int)
--> f(2, 19);
524288 (int)
--> f(2, 31);
-2147483648 (int)
--> f(2, 32);
0 (int)
-->
```

Bug fixed!

Looking good

Plausible

What?

What?

```
int f(int x, int y)
//@requires y >= 0;
//@ensures \result == POW(x,y);
{
  int b = x;
  int e = y;
  int r = 1;
  while (e > 0)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

The story continues ...