

Loops and Loop Invariants

CSE 331: Software Design and Implementation, Autumn 2026

Your job is to deliver code that you know works.

TODO

- [Reading 2](#): due today, Friday, October 16, 11:59pm
- Homework 2: out today, due Friday, October 23, 11:59pm
- Section next Tuesday: Floyd-Hoare logic

So far

- Lectures 1 to 3: "works" means "satisfies specification" (on all runs)
- Lectures 4 to 6: testing gives evidence of "works" on *some* runs
- Lecture 7: reasoning about *all* runs for code with no loops
- Today: reasoning about loops

Hoare triples and validity

- Hoare triple $\{\{ P \}\}$ code $\{\{ Q \}\}$
- **Valid:** Running code from state satisfying P leads to a state satisfying Q .
 - If run doesn't terminate, then validity makes no claims about it.
 - Code from last lecture always terminated. What about loops?

The rules turn the crank

- **Mechanical:** the claim before a line and the line give the claim after
 - No decision to make
- **Forward:** the strongest postcondition, x_0 for the old value
 - $\{\{ P \}\} x = e; \{\{ P[x \mapsto x_0] \text{ and } x = e[x \mapsto x_0] \}\}$
- **Backward:** the weakest precondition, a substitution
 - $\{\{ Q[x \mapsto e] \}\} x = e; \{\{ Q \}\}$
- **Sequence:** the claim after one line is the claim before the next

Rule: consequence

$\{\{ P \}\}$
 $\{\{ A \}\}$
code
 $\{\{ B \}\}$
 $\{\{ Q \}\}$

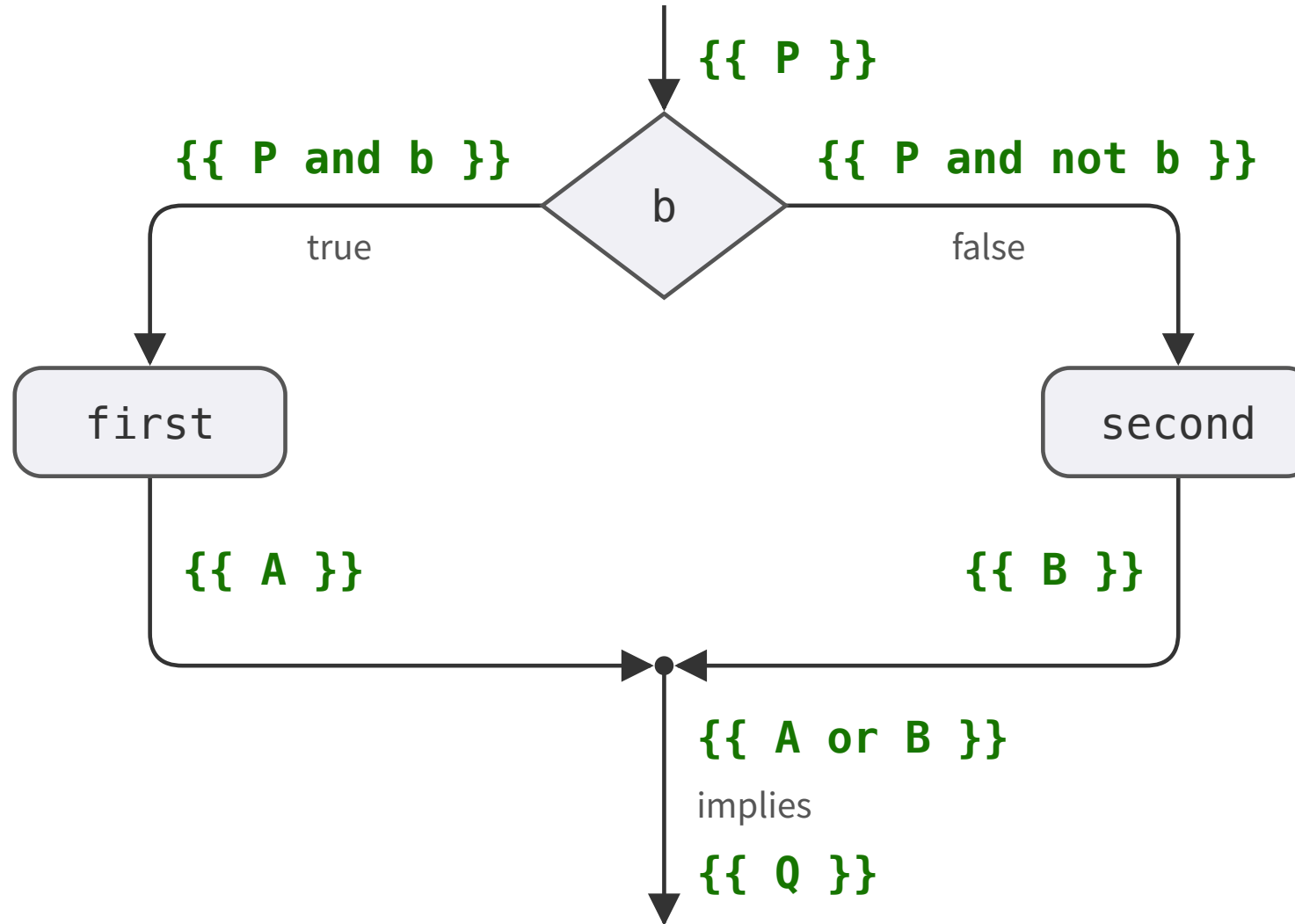
- We can "massage" the claims given by a rule via implication.
- Goal: show $\{\{ P \}\}$ code $\{\{ Q \}\}$ is valid. Break it down:
 - $\{\{ A \}\}$ code $\{\{ B \}\}$ is valid,
 - P implies A, and
 - B implies Q

Rule: conditional

```
{{ P }}  
if (b) {  
    {{ P and b }}  
    first  
    {{ A }}  
} else {  
    {{ P and not b }}  
    second  
    {{ B }}  
}  
{{ A or B }}  
{{ Q }}
```

- The condition flows into each branch.
- Mechanically reason through each branch.
- Merge the resulting postconditions with **or**.
- Check the result implies **Q**.

The conditional as a picture



Half sheet, question 1 (3 minutes)

- 1. Reason forward through `abs` (on the slide):
 - what assertion holds just after line 18?

```
9  /**
10  * Gives the absolute value of a number.
11  *
12  * @param n the number
13  * @requires -2147483647 <= n
14  * @return n if 0 <= n, and -n if n < 0
15  */
```

```
16  int abs(int n) {
17      {{ n = n0 }}
18      if (n < 0) {
19          {{ n = n0 and n < 0 }}
20          n = -n;
21          {{ n0 < 0 and n = -n0 }}
22      }
23      {{ (n0 < 0 and n = -n0) or
24         (0 <= n0 and n = n0) }}
25      return n;
26  }
```

Another way to think about it

- Reason about each possible path separately .
- First case: went into if statement, $n0 < 0$ and $n = -n0$
 - Spec requires us to return $-n0$, which equals n .
- Second case: skipped if statement, $0 \leq n0$ and $n = n0$
 - Spec requires us to return $n0$, which equals n .
- Both paths are true, so abs satisfies its specification

sum

```
6  /**
7   * Adds up the elements of an array.
8   *
9   * @param a the array
10  * @return a[0] + ... + a[a.length - 1],
11  *         the sum of the elements
12  */
13 long sum(int[] a) {
```

```
13 long sum(int[] a) {
14     long s = 0;
15     int i = 0;
16     while (i < a.length) {
17         s = s + a[i];
18         i = i + 1;
19     }
20     return s;
21 }
```

- After one pass: $0 < a.length$ and $i = 1$ and $s = a[0]$
- After two passes: $0 < a.length$ and $1 < a.length$ and $i = 2$ and $s = a[0] + a[1]$
- In general, no rule computes the strongest claim after a loop
 - So we annotate the loop by hand

sum on {3, 4, 5}: one row each time line 16 checks its condition

check	i	s	$i < a.length$	$s = a[0] + \dots + a[i - 1]$
1	0	0	$0 < 3$: true	true: 0 is the sum of no terms
2	1	3	$1 < 3$: true	true: $3 = 3$
3	2	7	$2 < 3$: true	true: $7 = 3 + 4$
4	3	12	$3 < 3$: false	true: $12 = 3 + 4 + 5$

Two facts, every time

- $0 \leq i$ and $i \leq a.length$
 - i starts at 0; only line 18 changes it, by adding 1
 - A pass runs only when $i < a.length$: after it, $i \leq a.length$
- $s = a[0] + \dots + a[i - 1]$
 - At the first condition check, i and s are 0: the sum of no terms
 - Suppose it is true at a condition check, and a pass runs
 - Line 17 adds $a[i]$ to s : the first $i + 1$ elements
 - Line 18 adds 1 to i : again the first i elements
- Neither reason depends on the array we picked

Definition: loop invariant

Definition: loop invariant

A **loop invariant** of `while (b) { body }` is an assertion that holds each time the loop checks condition `b`.

Before the first iteration, between iterations, and after the last.

Inv:

```
13  long sum(int[] a) {  
14      long s = 0;  
15      int i = 0;  
      {{ Inv: 0 <= i and i <= a.length and  
          s = a[0] + ... + a[i - 1] }}  
16      while (i < a.length) {  
17          s = s + a[i];  
18          i = i + 1;  
19      }  
20      return s;  
21  }
```

- **Inv:** puts **I** at the condition check, not before the loop

Rule: while loop

```
{{ P }}  
{{ Inv: I }}  
while (b) {  
    {{ I and b }}  
    body  
    {{ I }}  
}  
{{ I and not b }}  
{{ Q }}
```

b is the loop's condition.

I is a claim the programmer supplies.

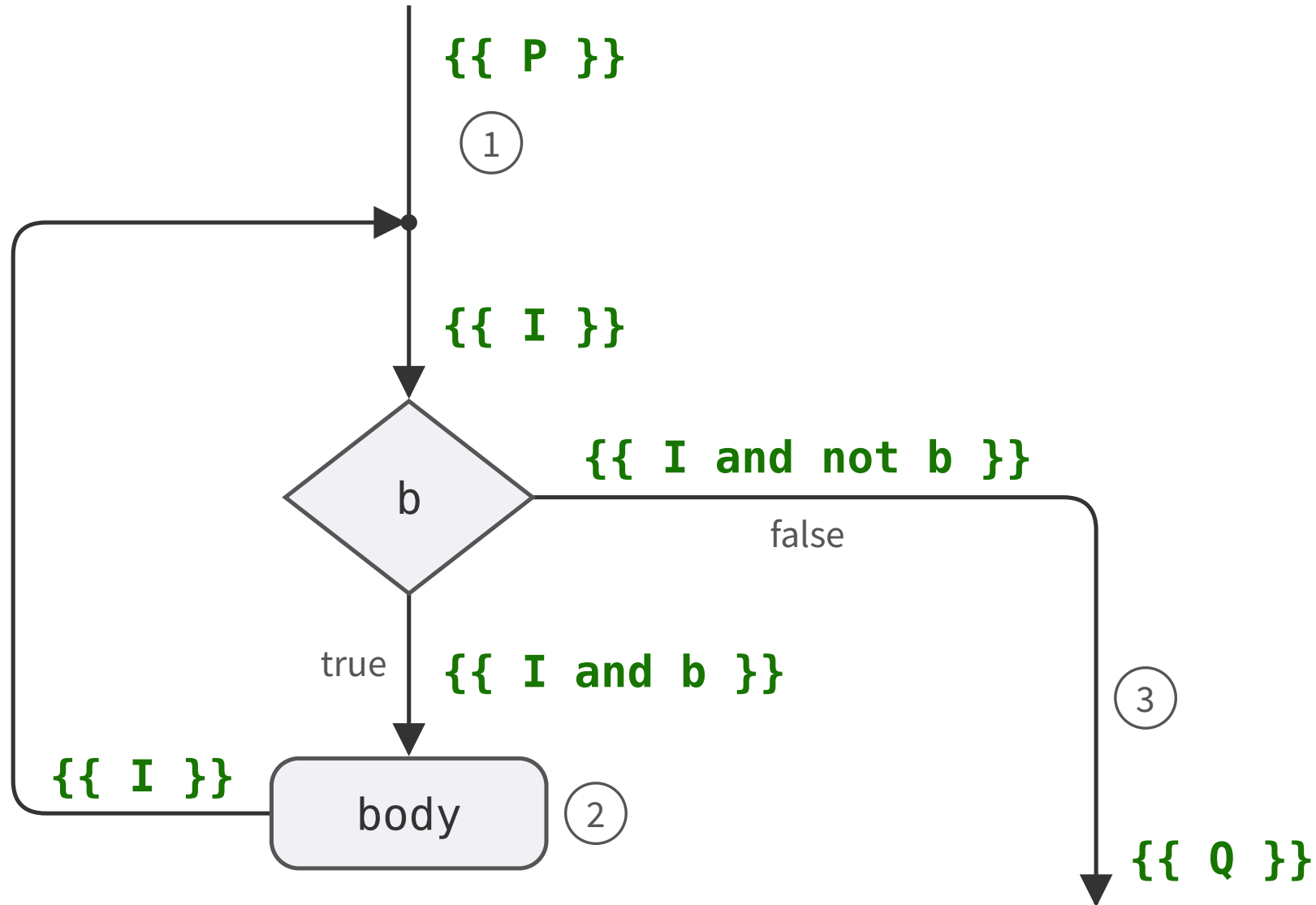
The triple $\{\{ P \}\} \text{ while } \dots \{\{ Q \}\}$ is valid if three checks pass:

(1) P implies I .

(2) The triple $\{\{ I \text{ and } b \}\} \text{ body } \{\{ I \}\}$ is valid.

(3) $I \text{ and not } b$ implies Q .

The rule as a picture



Why these checks

- In the figure, every way into the diamond crosses the arrow labeled I
- **First iteration:** P is true, so by (1), I is true
- **Between iterations:** each comes at the end of a pass
 - If I was true at the one before, the pass began with I and b
 - So by (2), the pass ended with I
- **After last iteration:** b is false, so I and not b after the loop
 - By (3), Q is true there

Each check is a step we know

- (1) and (3): implications, no code between
 - Each is a use of the rule of consequence
- (2): a triple about the body
 - No loop in the body, as in `sum`: Wednesday's rules check it
 - Forward or backward
- What is new is `I`
 - For now, `I` is given

Proving sum correct

```
long s = 0;
int i = 0;
while (i < a.length) {
    s = s + a[i];
    i = i + 1;
}
return s;
```

```
13  long sum(int[] a) {
      {{ true }}
14      long s = 0;
      {{ s = 0 }}
15      int i = 0;
      {{ s = 0 and i = 0 }}
      {{ Inv: 0 <= i and i <= a.length and
           s = a[0] + ... + a[i - 1] }}
```

```
      {{ Inv: 0 <= i and i <= a.length and
           s = a[0] + ... + a[i - 1] }}
16      while (i < a.length) {
          {{ 0 <= i and i <= a.length and
              s = a[0] + ... + a[i - 1] and i < a.length }}
17          s = s + a[i];
          {{ 0 <= i and i < a.length and s = a[0] + ... + a[i] }}
18          i = i + 1;
          {{ 1 <= i and i <= a.length and
              s = a[0] + ... + a[i - 1] }}
19      }
      {{ 0 <= i and i <= a.length and
          s = a[0] + ... + a[i - 1] and not (i < a.length) }}
      {{ s = a[0] + ... + a[a.length - 1] }}
20      return s;
21  }
```

Check (1)

$0 \leq i$	since $i = 0$
$i \leq a.length$	since $i = 0$ and $0 \leq a.length$
$a[0] + \dots + a[i - 1] = 0$	since $i = 0$: the sum of no terms
$= s$	since $s = 0$

- The claim before the loop, $s = 0$ and $i = 0$, has to imply I
- Each of I's three facts follows, one step to a line
- The second line: no array has a negative length
- So check (1) passes

Check (2), line 17

- $\{\{ I \text{ and } i < a.length \}\} \quad s = s + a[i]; \quad i = i + 1; \quad \{\{ I \}\}$

- Two lines in a row: **the rule for a sequence**

- Line 17: put the claim's sum for s_0 , the old s

$s = s_0 + a[i]$	the forward rule
$= a[0] + \dots + a[i - 1] + a[i]$	since $s_0 = a[0] + \dots + a[i - 1]$
$= a[0] + \dots + a[i]$	

- Between the lines, with $i \leq a.length$ dropped:
 - $0 \leq i \text{ and } i < a.length \text{ and } s = a[0] + \dots + a[i]$

Check (2), line 18

- Line 18: the old i is $i - 1$
 - Put $i - 1$ for i in each fact of the claim between the lines

$0 \leq i - 1$	from $0 \leq i$
$1 \leq i$	adding 1 to both sides
$i - 1 < a.length$	from $i < a.length$
$i \leq a.length$	since i and $a.length$ are integers
$s = a[0] + \dots + a[i - 1]$	from $s = a[0] + \dots + a[i]$

- That claim implies I, since $1 \leq i$ implies $0 \leq i$
- So check (2) passes

Check (3)

<code>a.length <= i</code>	since not (<code>i < a.length</code>)
<code>i = a.length</code>	since <code>i <= a.length</code>
<code>s = a[0] + ... + a[i - 1]</code>	from I
<code> = a[0] + ... + a[a.length - 1]</code>	since <code>i = a.length</code>

- The calculation ends at the postcondition: check (3) passes
- **The bound:** the step to `i = a.length` used `i <= a.length`
 - Without it, only `a.length <= i`, not `i = a.length`
 - The loop has not changed, so the claim is too weak
- **All three checks pass:** the triple for the body of `sum` is valid

Half sheet, question 2 (3 minutes)

- 2. Where does the proof break for the swapped body
 - $i = i + 1; s = s + a[i];?$

```
14  long s = 0;
15  int i = 0;
16  while (i < a.length) {
17      s = s + a[i];
17      i = i + 1;
18      i = i + 1;
18      s = s + a[i];
19  }
20  return s;
```

```
{ { Inv: 0 <= i and i <= a.length and
    s = a[0] + ... + a[i - 1] } }
```

Does the loop end?

- **Variant:** an integer upper bound on the passes remaining
 - Bounded below and strictly decreasing, so the loop ends
- For `sum: a.length - i`, the elements not yet added
 - An integer, since `a.length` and `i` are `ints`
 - At least 0 at each condition check, since I says `i <= a.length`
 - Smaller by 1 each pass, since line 18 adds 1 to `i`
- So at most `a.length` passes: `sum` satisfies its specification

while (true) { }

- A loop that never ends, with `true` for I
- Checks (1) and (2) pass: every claim implies `true`
- Check (3): `true and not true` has to imply Q
 - It is false in every state, so it implies every Q
- So check (3) passes, whatever Q is
 - Q = `false`: the checks would prove the postcondition `false`
- **The three checks pass for a loop that never ends**

Finding the invariant

- So far, the invariant of `sum` has been given to us
- *Where does an invariant come from?*
- Finding one takes some creativity
- In simple cases: **read I off the postcondition**
 - Q says what is true when the loop is done
 - I says how the loop makes progress toward Q

sum: read I off Q

- Q: $s = a[0] + \dots + a[a.length - 1]$
- At each condition check, i is the count of elements added so far
- So we put i for $a.length$ in Q: the main fact of I
 - $s = a[0] + \dots + a[i - 1]$
 - At the exit, $i = a.length$, so the main fact turns back into Q

sum: the bounds on i

- From where *i* starts and where it stops:
 - $0 \leq i$: line 15 starts *i* at 0
 - $i \leq a.length$: a pass runs only when $i < a.length$
- Check (3) needs the bound $i \leq a.length$
- So an invariant has two kinds of facts:
 - **What has been done so far:** $s = a[0] + \dots + a[i - 1]$
 - **How far along the loop is:** $0 \leq i$ and $i \leq a.length$

Half sheet, question 3 (3 minutes)

- 3. Find an invariant for the loop in `sumTo`.

```
13  long sumTo(int n) {  
14      long s = 0;  
15      int i = 0;  
16      while (i < n) {  
17          i = i + 1;  
18          s = s + i;  
19      }  
20      return s;  
21  }
```

Too weak, too strong

- **Too weak:** drop the bound $i \leq a.length$ from `sum`'s I
 - The calculation for check (3) cannot reach the postcondition
- **Too strong:** add $0 < a.length$ to `sum`'s I
 - Check (1) fails on an empty array
- A failed check means that the code is wrong
 - or that the claim we chose for I is
- Here `sum` is right, so both times the claim is what has to change

Back to Sum

- Lecture 1 argued about `Sum` in prose
 - "The loop added each element once"
- `sum` is the second loop of `Sum`
- For `sum`, that sentence is now one claim, the invariant
 - And three checks that Lecture 7's rules settle

What reasoning tells us

- Lecture 7 closed on three things: here is where each stands for loops
- **A false implication:** each check is one, or ends in one
 - A refuting test refutes the argument too, so we still test
- **Arithmetic that is not Java's:** the claims are about the integers
 - Java's `int` wraps: so a bound on the inputs, or a `long`
- **A claim the rules do not produce:** no rule gives I
 - Finding the invariant is the programmer's work

Your job is to deliver code that you know works.

Coming up

- Reading 2: due tonight, 11:59pm
- **Monday:** the [Reading 2](#) discussion
 - And web backends and 90s frontends
- **Tuesday:** section, Floyd-Hoare logic
- Homework 2: due Friday, October 23, 11:59pm
- Hand in your half sheet at the door