CSE 331
Software Design & Implementation

Kevin Zatloukal

Fall 2017

Lecture 4 – Writing Loops

(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Reminders

- HW2 on loops will be posted today
  - due next Thursday
  - harder, so start early

- Reading Quiz 1 is due Friday

Previously on CSE 331...

- Reasoning on straight-line code
  - turn the crack process
  - forward reasoning gives strongest postconditions
  - backward reasoning gives weakest preconditions
  - both generate valid Hoare triples
  - check validity of {{ P }} S {{ Q }} to Q' from forward reasoning or P' from backward reasoning

- Loops are more difficult
  - checking correctness requires a **loop invariant**

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ ______________ }}
i = 0;
{{ ______________ }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ ______________ }}
  s = s + b[i];
  {{ ______________ }}
  i = i + 1;
  {{ ______________ }}
}
{{ ______________ }}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ ______________ }}
  s = s + b[i];
  {{ ______________ }}
  i = i + 1;
  {{ ______________ }}
}
{{ ______________ }}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ ______________ }}
{{ s = b[0] + ... + b[n-1] }}
```

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
  {{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ ______________ }}
{{ s = b[0] + ... + b[n-1] }}

{{ s + b[i] = b[0] + ... + b[i] }}
```
s = s + b[i];
```
{{ s = b[0] + ... + b[i] }}
```
i = i + 1
```
{{ s = b[0] + ... + b[i-1] }}

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
  {{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

{{ s + b[i] = b[0] + ... + b[i] }}
```
s = s + b[i];
```
{{ s = b[0] + ... + b[i] }}
```
i = i + 1
```
{{ s = b[0] + ... + b[i-1] }}

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

Are we done?

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Does invariant hold initially?

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Example: sum of array

i = 3: s = b[0] + b[1] + b[2]
i = 2: s = b[0] + b[1]
i = 1: s = b[0]
i = 0: s = 0

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Holds initially? Yes: i = 0 implies s = b[0] + … + b[-1] = 0

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
   {{ s = b[0] + ... + b[i-1] and i != n }}
   s = s + b[i];
   {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
   i = i + 1;
   {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Are we done?
No, need to also check...

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Does postcondition hold on termination?

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
   s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
   i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Postcondition holds? Yes, since i = n.

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

```
{{ }}
s = 0;
{{ s = 0 }}
i = 0;
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
while (i != n) {
  {{ s = b[0] + ... + b[i-1] and i != n }}
  s = s + b[i];
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
  i = i + 1;
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
}
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}
```

Does loop body preserve invariant?

```
{{ s + b[i] = b[0] + ... + b[i] }}
s = s + b[i];
{{ s = b[0] + ... + b[i] }}
i = i + 1
{{ s = b[0] + ... + b[i-1] }}
```

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
{{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
{{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
{{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

Does loop body preserve invariant?

{{ s + b[i] = b[0] + ... + b[i] }}
```
s = s + b[i];
```
{{ s = b[0] + ... + b[i] }}
```
i = i + 1
```
{{ s = b[0] + ... + b[i-1] }}

Yes. Weaken by dropping “i-1 != n”

Example: sum of array

The following code to compute `b[0] + … + b[n-1]`:

Are we done?
No, need to also check...

{{ }}
```
s = 0;
```
{{ s = 0 }}
```
i = 0;
```
{{ s = 0 and i = 0 }}
{{ Inv: s = b[0] + ... + b[i-1] }}
```
while (i != n) {
```
  {{ s = b[0] + ... + b[i-1] and i != n }}
```
  s = s + b[i];
```
  {{ s = b[0] + ... + b[i-1] + b[i] and i != n }}
```
  i = i + 1;
```
  {{ s = b[0] + ... + b[i-2] + b[i-1] and i-1 != n }}
```
}
```
{{ s = b[0] + ... + b[i-1] and not (i != n) }}
{{ s = b[0] + ... + b[n-1] }}

Does loop body preserve invariant?

{{ s + b[i] = b[0] + ... + b[i] }}
```
s = s + b[i];
```
{{ s = b[0] + ... + b[i] }}
```
i = i + 1
```
{{ s = b[0] + ... + b[i-1] }}

Yes. If Inv holds, then so does this (just add b[i] to both sides of Inv)

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  i = i + 1;
  s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Example: sum of array (attempt 2)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
   i = i + 1;
   s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

- (s = 0 and i = -1) implies I
  - as before
- {{ I and i != n-1 }} S {{ I }}
  - reason backward:

{{ s + b[i+1] = b[0] + ... + b[i+1] }}

{{ s + b[i] = b[0] + … + b[i] }}

- (I and i = n-1) implies Q
  - as before

Example: sum of array (attempt 3)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n) {
   i = i + 1;
   s = s + b[i];
}
```

{{ s = b[0] + ... + b[n-1] }}

Suppose we use i != n instead of i != n-1…

We can spot this bug because the postcondition no longer follows.

When i = n, we get:

s = b[0] + … + b[n]

which is wrong

Example: sum of array (attempt 4)

Consider the following code to compute `b[0] + … + b[n-1]`:

{{ b.length >= n }}

```
s = 0;
i = -1;
```

{{ Inv: s = b[0] + ... + b[i] }}

```
while (i != n-1) {
  s = s + b[i];
  i = i + 1;
}
```

{{ s = b[0] + ... + b[n-1] }}

Suppose we misorder the assignments to `i` and `s`…

We can spot this bug because the invariant does not hold:

{{ s + b[i] = b[0] + ... + b[i+1] }}
{{ s = b[0] + ... + b[i+1] }}

First assertion is not I.

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

{{ 0 <= n <= b.length }}

```
i = k = 0;
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}

(Also: b contains the same numbers since we use swaps.)

(Also: P is true throughout the code. I'll skip writing that to save space...)

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

{{ 0 <= n <= b.length }}

```
i = k = 0;
```

{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}

```
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}

- I holds initially:
  - b[0], …, b[-1] is empty
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
  {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

{{ 0 <= n <= b.length }}

```
i = k = 0;
```

{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}

```
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}

```
  i = i + 1;
}
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
    if (b[i] < 0) {
        swap b[i], b[k];
        k = k + 1;
    }
   {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
    i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
    if (b[i] < 0) {
        {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
        swap b[i], b[k];
        k = k + 1;
        {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
    } else {
        {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] >= 0 }}
        {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
    }
    i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  } else {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] >= 0 }}
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
```

- I holds initially
- I and i = n implies postcondition

equivalent (the two assertions in the else branch)

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

{{ 0 <= n <= b.length }}

```
i = k = 0;
```

{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}

```
while (i != n) {
  if (b[i] < 0) {
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}

```
    swap b[i], b[k];
    k = k + 1;
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}

```
  }
  i = i + 1;
}
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}

- I holds initially
- I and i = n implies postcondition

Remain to check this…

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}
    swap b[i], b[k];
    k = k + 1;
    {{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i] }}
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

{{ 0 <= n <= b.length }}

```
i = k = 0;
```

{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}

```
while (i != n) {
  if (b[i] < 0) {
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] and b[i] < 0 }}

```
    swap b[i], b[k];
```

{{ b[0], ..., b[k] < 0 <= b[k+1], ..., b[i] }}

```
    k = k + 1;
  }
  i = i + 1;
}
```

{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}

This is a valid triple.
(Takes some thought.)

- I holds initially
- I and i = n implies postcondition

Example: partition array

Consider the following code to put the negative values at the beginning of array `b`:

```
{{ 0 <= n <= b.length }}
i = k = 0;
{{ Inv: b[0], ..., b[k-1] < 0 <= b[k], ..., b[i-1] }}
while (i != n) {
  if (b[i] < 0) {
    swap b[i], b[k];
    k = k + 1;
  }
  i = i + 1;
}
{{ b[0], ..., b[k-1] < 0 <= b[k], ..., b[n-1] }}
```

- I holds initially
- I and i = n implies postcondition
- I holds after loop body

Loop Invariants

- There is no general way to deduce the invariant from the code
- Why would we ever need to do this?
- It suggests coding like this:

Idea ➡ Code ➡ Invariant ➡ Proof

Loop Invariants

- There is no general way to deduce the invariant from the code

- Why would we ever need to do this?
- Don't do this:

~~Idea → Code → Invariant → Proof~~

Loop Invariants

- There is no general way to deduce the invariant from the code.

- Don't do this:

~~Idea ➡ Code ➡ Invariant ➡ Proof~~

- Instead, do this:

Idea ➡ Invariant ➡ Code ➡ Proof

Loop Invariants Before Code

- Loop invariant comes out of the algorithm idea
  - describes partial progress toward the goal
    - how you will get from start to end
  - contains the essence of the algorithm idea

- A good invariant will make the code easier to write
  - a great invariant makes the code “write itself”
  - (we will see the same thing with invariants for ADTs etc.)

Loop Invariants in this Course

- We advocate writing invariants before the code
  - if the code is there, the invariant should be there too

- You will not be asked to find the invariant for the code
- Types of problems in HW2:
  - given invariant and code, prove it correct
  - given invariant, write code
  - write invariant and (then) code [for simple algorithms]

- When writing code, document your loop invariants (if nontrivial)
  - don’t make readers re-discover them
  - improves changeability and understandability

Loop Invariant Design Patterns

- Often loop invariant is a weakening of the postcondition
  - partial progress with completion a special case

- Example: sum the values in an array
  - postcondition: s = b[0] + … + b[n-1]
  - loop invariant: s = b[0] + ... + b[i-1] for some i
    - postcondition is special case i = n

- Only *slightly* weakened postcondition: Inv and not `cond` implies Q
- Stronger is usually better
  - if it is strong enough, there is only one way to write body
  - (but if it's too strong, there may be no way to write the body!)

Filling in code, given invariant

Can often deduce correct code directly from loop invariant

Filling in code, given invariant

Can often deduce correct code directly from loop invariant:

- what is the easiest way to satisfy the loop invariant?
  - this gives you the initialization code

Filling in code, given invariant

Can often deduce correct code directly from loop invariant:

- what is the easiest way to satisfy the loop invariant?
  - this gives you the initialization code
- when does loop invariant satisfy the postcondition?
  - this gives you the termination condition

Filling in code, given invariant

Can often deduce correct code directly from loop invariant:

- what is the easiest way to satisfy the loop invariant?
  - this gives you the initialization code
- when does loop invariant satisfy the postcondition?
  - this gives you the termination condition
- how do you make progress toward termination?
  - if condition is i != n (and i <= n), try i = i + 1
  - if condition is i != j (and i <= j), try i = i + 1 or j = j – 1
  - write out the new invariant with this change (e.g. i+1 for i)
  - figure out code needed to make the new invariant hold
    - usually just a small change (since Inv change is small)

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
??
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
??
```

Easiest way to make this hold?

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
??
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Easiest way to make this hold?
Take i = 1 and m = max(b[0])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

When does Inv imply postcondition?

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (?) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

When does Inv imply postcondition?
Happens when i = n

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we progress toward termination?

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??

}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we progress toward termination?
We start at i = 1 and end at i = n, so…

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we progress toward termination?
We start at i = 1 and end at i = n, so
Try this.

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

When i becomes i+1, Inv becomes:
m = max(b[0], …, b[i])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we get
from m = max(b[0], …, b[i-1])
to m = max(b[0], …, b[i])?

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {

   ??
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we get
from m = max(b[0], …, b[i-1])
to m = max(b[0], …, b[i])?

Set m = max(m, b[i])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {
   if (b[i] > m)
     m = b[i];
   i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

How do we get
from m = max(b[0], …, b[i-1])
to m = max(b[0], …, b[i])?

Set m = max(m, b[i])

Example: max of array

Write code to compute max(b[0], …, b[n-1]):

{{ b.length >= n and n > 0 }}

```
int i = 1;
int m = b[0];
```

{{ Inv: m = max(b[0], ..., b[i-1]) }}

```
while (i != n) {
  if (b[i] > m)
    m = b[i];
  i = i + 1;
}
```

{{ m = max(b[0], ..., b[n-1]) }}

Filling in code, given invariant

As you saw, we can often deduce correct code directly from Inv

- cases where this happens are the best invariants

The invariant is *often* the essence of the algorithm **idea**

- then rest is just details that follow from the invariant

Finding the loop invariant

Not every loop invariant is simple weakening of postcondition, but…

- that is the easiest case
- it happens a lot

In this class (e.g., exams):

- if I ask you to find the invariant, it will *very likely* be of this type
- I may ask you to inspect code with more complex invariants
- to learn about more ways of finding invariants: CSE 421

Examples: finding loop invariants

1. sum of array
   – postcondition: s = b[0] + b[1] + … + b[n-1]

Examples: finding loop invariants

1. sum of array
   - postcondition: s = b[0] + b[1] + … + b[n-1]
   - loop invariant: s = b[0] + b[1] + … + b[i-1]
     - gives postcondition when i = n
     - gives s = 0 when i = 0

Examples: finding loop invariants

1. sum of array
   - postcondition: s = b[0] + b[1] + … + b[n-1]
   - loop invariant: s = b[0] + b[1] + … + b[i-1]
     - gives postcondition when i = n
     - gives s = 0 when i = 0

2. max of array
   - postcondition: m = max(b[0], b[1], …, b[n-1])

Examples: finding loop invariants

1. sum of array
   - postcondition: s = b[0] + b[1] + … + b[n-1]
   - loop invariant: s = b[0] + b[1] + … + b[i-1]
     - gives postcondition when i = n
     - gives s = 0 when i = 0

2. max of array
   - postcondition: m = max(b[0], b[1], …, b[n-1])
   - loop invariant: m = max(b[0], b[1], ..., b[i-1])
     - gives postcondition when i = n
     - gives m = b[0] when i = 1