CSE 331
Software Design & Implementation

Kevin Zatloukal

Summer 2016

Lecture 4.5 – Writing Loops

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

Announcements

- HW1 near-universal mistake:

  {{ |x| > 8 }}

  ```
  x = x / 2;
  ```

  {{ ? }}

Announcements

- HW1 near-universal mistake:

  {{ |x| > 8 }}

  ```
  x = x / 2;
  ```

  {{ |x| >= 4 }} Example: if x = 9, then x/2 = 4

Announcements

- HW1 near-universal mistake:
  - integer division is tricky
  - this won't come up again in class, but be aware IRL

Announcements

- HW1 near-universal mistake:
  - integer division is tricky
  - this won't come up again in class, but be aware IRL
- HW2:
  - more time?

Announcements

- HW1 near-universal mistake:
  - integer division is tricky
  - this won't come up again in class, but be aware IRL
- HW2:
  - more time?
  - less work?

Announcements

- HW1 near-universal mistake:
  - integer division is tricky
  - this won't come up again in class, but be aware IRL

- HW2:
  - more time?
  - less work?

- HW3:
  - if you will work from your own laptop, **bring it to quiz section**
    - install Java JDK & Eclipse before ("Working at Home" doc)
  - if you have any problems, contact staff to get extension
  - will shortly see emails from Gitlab (ignore until tomorrow)

Agenda

Plan for today:

1. Review important ideas about loop invariants
2. Ask me questions about HW2
3. Move on to specifications… (continued Friday)

Review

Checking correctness of loops

Not just about finding in assertions after each line…

Also need to check that loop invariant:

1. holds initially
2. is preserved by the loop body
3. implies postcondition upon termination

Problems 1-2 on HW2 ask you to fill in the assertions and also Check that 1-2 hold (I didn't ask you to do 3)

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, we need to check 1-3**

```
{{ }}
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, we need to check 1-3

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

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

```
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]`:

```
{{ }}
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, we need to check 1-3

```
{{ 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, we need to check 1-3

```
{{ }}
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, we need to check 1-3

{{ }}
```
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]`:

{{ }}
```
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, we need to check 1-3

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, we need to check 1-3

```
{{ }}
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)

Reasoning more quickly

Your speed at reasoning will improve with practice

Experts typically do not write down assertions for every line

- instead do much of it in their head
- sometimes reason multiple lines at a time (last lecture)
- but still fall back to line-by-line assertions for **tricky code**
  - e.g., binary search

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

Can often deduce correct code directly from loop invariant

- ones 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 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

Example: Dutch National Flag (HW0)

Postcondition says we need to produce this:

Red	White	Blue

And it starts out like this:

Mixed colors: red, white, blue

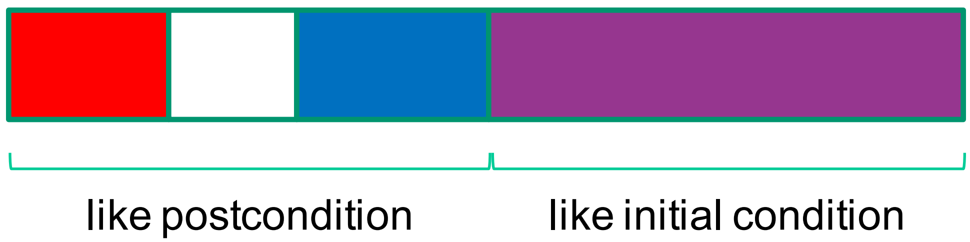
Loop invariant should (essentially) have

- postcondition as a special case
- initial condition as a special case

Loop invariant describes continuum of partial progress

Example: Dutch National Flag

The first idea that comes to mind:

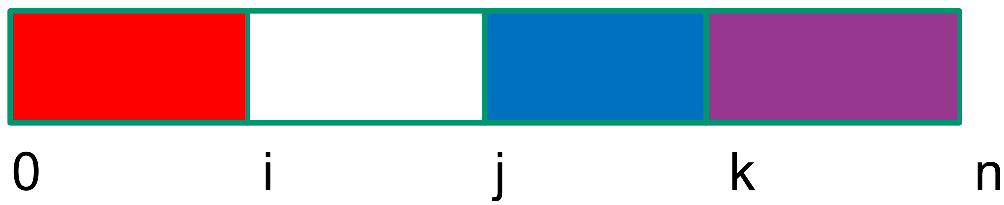


Example: Dutch National Flag

The first idea that comes to mind works.

Example: Dutch National Flag

To describe this mathematically, create names for split points



Create indices i, j, k with 0 <= i <= j <= k <= n

The invariant is then

- A[0], A[1], …, A[i-1] is red
- A[i], A[i+1], …, A[j-1] is white
- A[j], A[j+1], …, A[k-1] is blue
- (and A[k], A[k+1], …, A[n-1] is unconstrained)