CS 1110

Lecture 20: Loop invariants

Announcements

A3 and Academic Integrity Policy

We have a number of cases of A3 submissions where proper attribution/credit was not given, generally observable through computationally-analyzed too-similar code.

(Some parties have already been contacted requesting more information about the situation; some have not, due to volume. We will be in touch later with *both* previously-contacted and non-contacted parties about scheduling academic integrity hearings.)

We therefore remind you to re-review http://www.cs.cornell.edu/courses/cs1110/2014sp/about/integrity.php.

Slides by D. Gries, L. Lee, S. Marschner, W. White

Extract from the policy (but read the whole thing)

"So, if you talked to someone not in your CMS group and got an idea from them that became part of your solution, write this at the top of your submission. If a friend gave you some debugging help and suggested a fix, write this at the top of your submission. If a consultant suggested a useful test case that you incorporated in your code, write this at the top of your submission. If you found a key solution to some sticky problem in a post on stackoverflow.com and incorporated that idea into your program, write this at the top of your submission. And so on—you get the idea. When you hand in an assignment without remarking on others' contributions, you are claiming credit for everything in it as your own creation. **To turn in code that someone else invented and claim it as your own is fraudulent and punishable under the Code of Academic Integrity."**

Announcements

A3 grades released

Approximate grade centers will be announced on Piazza.

The Utility of Assertions

- **assertion**: statement regarding belief about state at a given point
  - Can be expressed as comments or `assert` statements
- **invariant**: assertion supposed to "always" be true (When state changes to temporarily invalidate the invariant, there should be code that immediately truthifies the invariant again. Example: class invariants and __init__ methods.)
- **loop invariant**: assertion supposed to be true before and after each iteration of the loop

Assertions prevent bugs, by helping you keep track of what your variables mean and what you are doing.

And assertions help find bugs, by making it easier to check *belief–code mismatches*, the root of all bugs!

Preconditions & Postconditions

while n ....

precondition

```
# x  = sum of 1..n-1
x = x + n
n = n + 1
# x =  sum of 1..n-1
```

postcondition

n

1 2 3 4 5 6 7 8

x contains the sum of these (6)

n

1 2 3 4 5 6 7 8

x contains the sum of these (10)

- **Precondition:** assertion placed before a segment
- **Postcondition:** assertion placed after a segment
- Intervening code should guarantee that:

  if precondition is true, *then* the postcondition will be.

Truthifying a postcondition

precondition

```
# x  = sum of 1..n

n = n + 1
# x =  sum of 1..n
```

postcondition

What statement do you put here to make progress ~~the postcondition true~~?

A: x = x + 1

B: x = x + n

C: x = x + n+1

D: None of the above

E: I don't know

~~Hint: what does the range 1.."new n" have that 1.."old n" doesn't?~~

Invariants: Assertions That Do Not Change

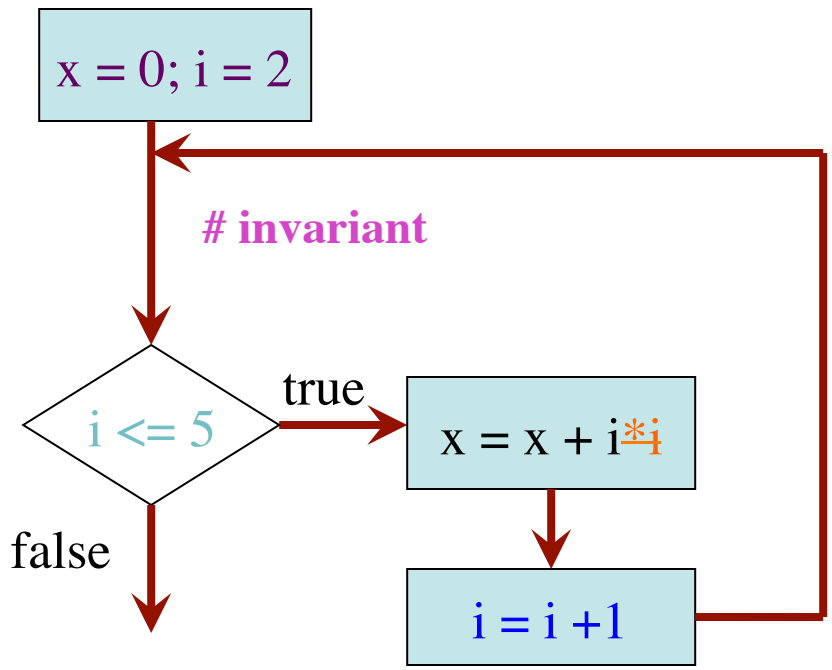
(Dumb) task: add the numbers in 2..5

Invariant:

x = sum of ~~squares of~~ 2..i-1

(i = start of unprocessed)

```
x = 0; i = 2
# pre: x = sum of ~~sqs of~~ 2..1
while i <= 5:
    x = x + i~~*i~~
    i = i +1
# post:  x = sum of 2..5
```



Designing while-loops

1. Write the command and equivalent postcondition to figure out the loop invariant.
   - Alternately: figure out the compromise between knowing "everything" (termination) and knowing nothing (initialization)
2. Figure out the initialization (how does it start)?
3. Figure out the loop condition: when is there still work to do?
4. Figure out how to make progress (~~"increment)"~~
5. Update ("increment") variables to indicate new state, preserving invariant

Initialize variables (if necessary) to make invariant true

```
# Invariant: range b..k-1 has been processed
while k <= c:
    <do the right thing to maintain invariant when update happens>
    k = k + 1
```

~~Finding an Invariant~~ Another example invariant

```
# Make b True if n is prime, False otherwise
b = True; k = 2
# invariant: b is True if no int in 2..k-1 divides n, False otherwise
#  (k is start of unchecked-for-dividing)
while k < n:
    # Process k
    if n % k == 0:
        b = False
    k = k +1
# b is True if no int in 2..n-1 divides n, False otherwise
```

Command to do something

Equivalent postcondition

What invariant to use?
(there are several possibilities).

1 2 3 … k-1 k k+1 … n

~~Finding and~~ Using an Invariant

```
# set x to # adjacent equal pairs in s[0..len(s)-1]

# inv: x = # adjacent equal pairs in s[0..k-1]
while     :

# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

if s = 'ebeee', set x to 2

Equivalent postcondition

k: next thing to check for ending a pair

~~Finding and~~ Using an Invariant

```
# set x to # adjacent equal pairs in s[0..len(s)-1]
 x = 0
 k = 1 # this choice means s[k-1] always exists
# inv: x = # adjacent equal pairs in s[0..k-1]
while
```

Command to do something

if s = 'ebeee', set x to 2

```
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Equivalent postcondition

k: next thing to check for ending a pair

What initialization for k?

A: k = 0

B: k = 1

C: k = –1

D: I don't know

~~Finding and~~ Using an Invariant

```
# set x to # adjacent equal pairs in s[0..len(s)-1]
x = 0
k = 1 # this choice means s[k-1] always exists
# inv: x = # adjacent equal pairs in s[0..k-1]
while  k < len(s):

   x += (1 if s[k-1] == s[k] else 0) # can also do x+= (s[k-1]==s[k])
   k += 1
# x = # adjacent equal pairs in s[0..len(s)-1]
```

Command to do something

if s = 'ebeee', set x to 2

Equivalent postcondition

k: next thing to check for ending a pair

Which elements are compared when we "process" k?

A: s[k] and s[k+1]

B: s[k-1] and s[k]

C: s[k-1] and s[k+1]

D: s[k] and s[n]

E: I don't know

Another example regarding initialization

```
# s is a string; len(s) >= 1
# Set c to largest element in s
c = ??
k = ??
# inv:  c is largest element in s[0..k–1]
while k < len(s):
    # Process k
    k = k+1
#  c = largest char in s[0..len(s)–1]
```

Command to do something

Equivalent postcondition

1. What is the invariant?
2. How do we initialize c and k?

A: `k = 0; c = s[0]`

B: `k = 1; c = s[0]`

C: `k = 1; c = s[1]`

D: `k = 0; c = s[1]`

E: None of the above

An empty set of characters or integers has no maximum. Therefore, be sure that 0..k–1 is not empty. You must start with k = 1.