
CSE 374

Programming Concepts & Tools

Hal Perkins

Winter 2011

Lecture 17 – Specifications, error checking & assert

Where we are

Talked about testing, but not what (partially) correct was

- What does it mean to say a program is “correct”?
- How do we talk about what a program should “do”?
- What do we do when it “doesn’t”?

Specifying code?

- We made a *big* assumption, that we know what the code is supposed to do!
- Often, a complete *specification* is at least as difficult as writing the code. But:
 - It's still worth thinking about.
 - *Partial* specifications are better than none.
 - *Checking* specifications (at compile-time and/or run-time) is great for finding bugs early and “assigning blame”.

Full specification

- Often tractable for very simple stuff: “Given integers $x, y > 0$, return their greatest common divisor.”
- What about sorting a doubly-linked list?
 - Precondition: Can input be NULL? Can any prev and next fields be NULL? Should the list be circular or not?
 - Postcondition: Sorted (how to specify?)
- And there’s often more than “pre” and “post” – time/space overhead, other effects (such as printing), things that may happen in parallel
- Specs should guide programming and testing! Should be declarative (“what” not “how”) to decouple implementation and use.

Pre/post conditions and invariants

- Pre- and post-conditions apply to any statement, not just functions
 - What is assumed before and guaranteed after
- Because a loop “calls itself” its body’s post-condition better imply the loop’s precondition.
 - A loop invariant
- Example: find max (next slide)
- Meaning of “correct”: a segment of code is correct if, when it begins execution in a state where its precondition is true, it is guaranteed to terminate in a state in which the postcondition is true

Pre/post conditions and invariants

```
// pre: arr has length len; len >= 1
int max = arr[0];
int i=1;
while(i<len) {
    if(arr[i] > max)
        max = arr[i];
    ++i;
}
// post: max >= all arr elements
```

loop-invariant: For all $j < i$, $\max \geq \text{arr}[j]$.

- to show it holds after the loop body, must assume it holds before loop body
- loop-invariant plus $!(i < \text{len})$ after body, enough to show post

Partial specifications

The difficulty of full specifications need not mean abandoning all hope.

Useful partial specs:

- Can args be NULL?
- Can args alias?
- Are pointers to stack data allowed? Dangling pointers?
- Are cycles in data structures allowed?
- What is the minimum/maximum length of an array?
- ...

Guides callers, callees, and testers.

Beyond testing

- Specs are useful for more than “things to think about while coding” and testing and comments.
- Sometimes you can check them dynamically, e.g., with assertions (all examples true for C and Java)
 - Easy: argument not NULL
 - Harder but doable: list not cyclic
 - Impossible: Does the caller have other pointers to this object?

assert in C

```
#include <assert.h>
void f(int *x, int*y) {
    assert(x!=NULL);
    assert(x!=y);
    ...
}
```

- assert is a macro; ignore argument if NDEBUG defined at time of #include, else evaluate and if zero (false!) exit program with file/line number in error message.
- Watch Out! Be sure that none of the code in an assert has side effects that alter the program's behavior. Otherwise you get different results when assertions are enabled vs. when they are not.

assert style

- Many guidelines are overly simply and say “always” check everything you can. But:
 - Often not on “private” functions (caller already checked)
 - Unnecessary if checked statically
- “Disabled” in released code because:
 - executing them takes time
 - failures are not fixable by users anyway
 - assertions themselves could have bugs/vulnerabilities
- Others say:
 - Should leave enabled; corrupting data on real runs is worse than when debugging

assert vs exceptions; error checking

- Suppose a condition should be true at a given point in the program, but it's not. What do we do?
- Common practice:
 - If the condition involves preconditions of a public interface ($x \geq 0$, list not full, p not NULL,...), treat a failure as an error and throw an exception or terminate with an error code.
 - Don't trust client code you don't control!
 - If the condition is an internal matter, a failure represents a programming error (bug). Check this with assert.

Static checking

- A stronger type system or other code-analysis tool might take a program and examine it for various kinds of errors.
 - Plusses: earlier detection (“coverage” without running program), faster code
 - Minus: Potential “false positives” (spec couldn’t ever actually be violated, but tool can’t prove that)
- Deep CS theory fact: Every code-analysis tool proving a non-trivial fact has either false positives (unwarranted warning) or false negatives (missed bug) or both.
- Deep real-world fact: That doesn’t make them unuseful.