On Testing

Robbert van Renesse
Tip 1: use assertions in your data structure code (and not in your test code)

• Pepper your queue code with assertions before testing
  • think carefully about invariants
  • check invariants as often as possible
  • write code to check invariants

• Example:
  • write function queue_check(q) that walks the linked list. When it gets to the end, check that the tail points where it’s supposed to and that the length is correct
  • also check that (q->len == 0) == (q->head == NULL) == (q->tail == NULL)
  • assert(queue_check(q)) at the beginning and end of every function
  • Note that assertions automatically are turned off in production code
    • May want to comment them out before submission!
Quick aside on using assertions

• `assert(P)` --- executable comment

• Important: `P` should have no side effects
  • so, don’t do `assert(queue_dequeue(q) == 0)`

• assert statements should be no-ops and can be turned off

• use assert statements to check correctness, not to detect failures
  • so, don’t do `p = malloc(); assert(p != NULL)`

• split conjunctions
  • so, don’t do `assert(P && Q)` but do `assert(P); assert(Q)`
Tip 2: don’t ignore warnings

• Compile with –g –Wall
  • e.g., cc –g –Wall x.c

• Do *not* submit code with outstanding warnings

• Do *not* get rid of warnings by hasty casting
  • Be very careful and only cast if you know exactly what you’re doing
Tip 3: run small tests

• Don’t run very large tests (10s of operations or more)
  • you are unlikely to find bugs that you can’t find with small tests
  • it’s hard to figure out what went wrong
  • tests may take a long time for no good reason
Tip 4: use valgrind

• Will immediately notify you if
  • you are using uninitialized memory (e.g., from malloc())
  • you are accessing illegal memory
  • you are leaking memory
• It will give you lots of information about how it happened
• Easiest to install under Linux, so use a virtual machine or log into CSUGlab Linux machines
Tip 5: only check things that are specified

- Carefully read the spec and design tests for each specified case
- Do not check things that are not specified
  - queue_length(NULL) has unspecified behavior---don’t test it
Tip 6: think carefully about corner cases

• dequeuing from an empty queue
  • does it return the right error value?

• deleting the last element enqueued
  • does it update the tail pointer? How do you test for that?
    • enq(1); enq(2); del(2); enq(3); deq() == 1?; deq() == 3?

• deleting the last element of a queue
  • does it reset everything?

• deleting a non-existent element?
  • does it leave the queue unchanged?

• insert on an empty queue?
  • does it update the tail pointer?

• enqueue(item = NULL); delete(item = NULL);
Tip 7 (advanced): be systematic
(no need to worry about corner cases)

• Define a set of basic operations you might want to use
  • enq(1), enq(2), enq(NULL), deq(pitem == NULL), deq(pitem != NULL), ins(1),
    ins(2), ins(NULL), len(), del(1), del(2), del(NULL), len() iterate()

• Systematically check all sequences of operations
  • all sequences of length 1
  • all sequences of length 2
  • all sequences of length 3
  • ...

• With 13 basic operations, there are $13^6$ (approx. 5 million) sequences
  of length 6
  • likely to trigger any bug
Checking a sequence of operations

• How do you know a sequence worked correctly?
• Check against specification!
• But what if spec is written in English?
• Translate spec into C!

```c
#define MAX_QUEUE_SIZE 100
typedef struct queue {
    void *buffer[MAX_QUEUE_SIZE];
    unsigned in, out;
} *queue_t;

queue_t gold_new() {
    struct queue *q = calloc(1, sizeof(*q));
    q->in = q->out = MAX_QUEUE_SIZE / 2;
    return q;
}

int gold_enqueue(queue_t q, void* item) {
    assert(q->out <= q->in);
    assert(q->in < MAX_QUEUE_SIZE);
    q->buffer[q->in++] = item;
    return 0;
}
```