CSE 374: Lecture 20

Software Specification, HW6

Why Specs?

What is testing?

Software testing evaluates the **effectiveness** of a software sol

★ Systemat

★ Objective

But how do we know what it is supposed to do?

Effectiveness:

Does what it is supposed to do

Fails gracefully

Uses memory safely and efficiently

Computes in reasonable time

https://en.wikipedia.org/wiki/Software_testing

"Test your software or your users will."
Hunt & Thomas -- The Pragmatic Programmer

Full Specification

- Tractable for very simple stuff:
  - “Given integers *x*,*y>0*, return the greatest common divisor.”
- What about sorting a doubly-linked list?
  - Precondition: Can input be **`NULL`**? Can any **`prev`** and **`next`** fields be **`NULL`**? Can the list be circular or not?
  - Postcondition: Sorted (how to specify?, on what condition?)
- Beyond “pre” and “post” – time/space overhead, other effects (such as printing), things that happen in parallel
- Specs guide programming and testing!
- Declarative (“what” not “how”)
  - decouples implementation and use.

Basics: Pre and Post Conditions

- Pre- and post-conditions apply to any statement, not just functions
  - What is promised before and guaranteed after
- Because a loop “calls itself” its body’s post-condition better imply the loop’s precondition
  - A loop invariant
- 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
- Example: find max (next slide)

Find Max / Loop-invariant

```
// 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 >= arr[j]`.

to show it holds after the loop body, must assume it holds before loop body

loop-invariant plus `!(i<len)` after body, enough to show post

Partial Specification

It may not be possible to completely specify an algorithm (or expedient)

Partial Specs:

- ➔ What is each argument precisely? Can arguments be null?
- ➔ Are pointers to stack data allowed? (what if stack is popped?)
- ➔ Are cycles in data structures allowed?
- ➔ Are there min and max sizes of data?

Checking specifications as part of code

- Specs are useful for more than writing code and testing
- Check them dynamically, e.g., with assertions
  - Easy: argument not `NULL`
  - Harder but doable: list not cyclic
  - Impossible: Does the caller have other pointers to this object?

Use ‘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

Remember this?

Unit Testing

Test small components of code individually

Basic approach - ‘assert’ desired performance.

(Note: Use conditional compilation
`Ifdef NDEBUG`
Plus macro
`#define assert(ignore)((void) 0)`
To compile without test code.)

```
#include <assert.h>
#include <stdlib.h>
#include "f.h"

// Assert statements will fail with a message
// if not true.
int main(int argc, char** argv) {

    assert(!f(0, 0));  // Test 1: f(0,0) => 0
    assert(f(0, 1));   // Test 2: f(0,1) => T
    assert(f(1, 0));   // Test 3: f(1,0) => T
    assert(f(1,1));    //  Test 4: f(1,1) => T

    // Test case 5: f(-1,1) => not-0
    assert(f(-1,1));
    return EXIT_SUCCESS;
}
```

```
OUTPUT >> program: f.c:9: main: Assertion
`!f(0,0)' failed.   Abort (core dumped)
```

Assert Style

- Often guidelines are simple and say “always” check everything, **but**:
  - Often not on “private” functions (caller already checked)
  - Unnecessary if checked statically
- Usually “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

Exceptions

- Assert is used to verify internal expectations in code controlled by user
  - If asserts are violated code can be modified
- Exceptions are used to check expectations of code outside your control
  - Such as the return of a library function
  - Should usually exit (EXIT_FAILURE)
- Language dependent - Java offers asserts on top of its exception handling, C does not offer exception handling.
  - User is expected to anticipate trouble and catch it
  - Returning success/failure codes can be very helpful
- *Other Language dependent tools exist*
  - *Example: strong type checking prevents some sorts of specification violations*

API: Application Programming Interface

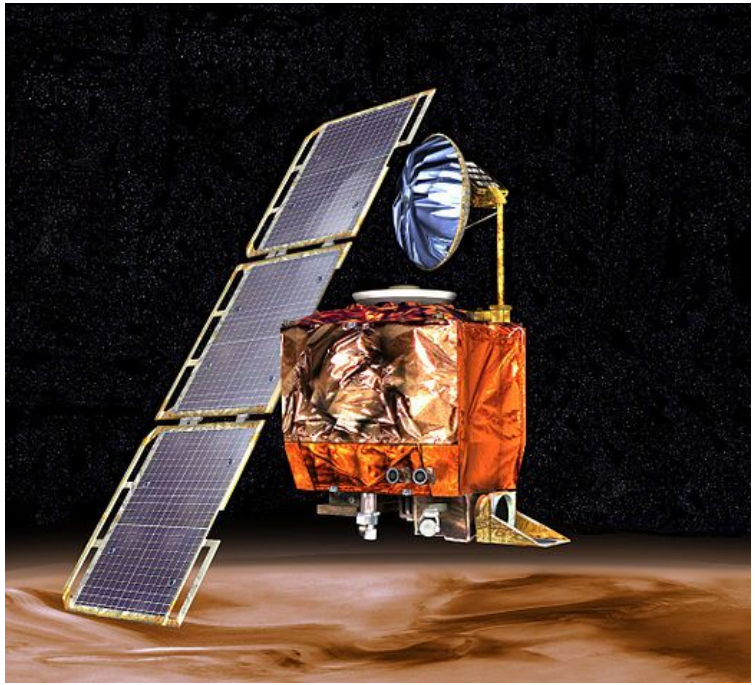
- Defines input and output for ‘applications’
  - Can be entire apps, or subfunctions, or classes
  - Library APIs describe available functions in library
- Useful for writing & testing
  - API dictates function prototype
  - (Black box?) Tests that show API adherence

Javadocs: Great example of an API standard

@param
@returns
@throws
@see
@author

Scientific Computing

Notes: worth specifying units in the function description and perhaps argument names.



HW6

In C: `malloc` and `free` are wrappers to system calls that reserve space in memory, or cancel the reservation.

(System calls deal with memory management, I/O stream management, access files, access the network.)

But `malloc` and `free` are more user friendly than the essential system calls.

Implement equivalents:

```
// acts like ‘malloc’ and returns address in memory
void* getmem(uintptr_t size)
// acts like ‘free’ and releases memory
void freemem(void* p)
```

Note:

`Uintptr_t` is an integer type that holds a pointer.

`void*` is a pointer to an unspecified type

HW6: Approach

1. We use a system call (aka malloc) to get a big chunk of memory - like 4k-10k bytes.
2. We then parcel out pieces of this chunk to individual calls to getmem and mark them as reserved.
3. When someone calls freemem, we return the chunks to the set of free chunks.
4. How do we keep track of all of the available chunks vs reserved chunks?
    a. Use something called a "free list", which is a linked list of nodes that store information about available chunks.
    b. Shared by both getmem and freemem.
    c. Each block on the free list starts with an uintptr_t integer that gives its size followed by a pointer to the next block on the free list.
    d. To help keep data in dynamically allocated blocks properly aligned, we require that all of the blocks be a multiple of 16 bytes in size, and that their addresses also be a multiple of 16 (this is the same way that the built-in malloc works).

Approach, Cont.

Getmem request? Scan the free list looking for a block of storage that is at least as large as the amount requested, delete that block from the free list, and return a pointer to it to the caller.

Freemem: return the given block to the free list, combining it with any adjacent free blocks if possible to create a single, larger block instead of several smaller ones.

Approach: getting memory blocks

If, a large enough block exists, ‘getmem’ splits the block into an appropriate sized chunk and pointer to the block

Else, getmem needs to

Get a good-sized block of storage from the underlying system.

Add it to the free list

Split it up, yielding a block that will satisfy the request (‘**if**’ condition)

Note, Initial call to getmem finds it with no memory, and results in ‘**else**’ condition.

Approach: returning memory

- Freemem gets a pointer to a block of storage and adds it to the free list, combining it with adjacent blocks on the list.
- Freemem isn't told is how big the block is and must find the size of the block.
- The usual way this is done is to have getmem actually allocate a block of memory that is a bit larger than the user's request, store the free list node or just the size of the block at the beginning of that block.
- The returned pointer is actually points a few bytes beyond the real start of the block.
- When freemem is called, it can take the pointer it is given, subtract the appropriate number of bytes to get the real start address of the block, and find the size of the block there.

Use ‘assert’ in C: `void check_heap ();`

Check for possible problems with the free list data structure. This function should use `assert`s to verify that:

- Blocks are ordered with increasing memory addresses
- Block sizes are positive numbers and no smaller than whatever minimum size you are using
- Blocks do not overlap (the start + length of a block is not an address in the middle of a later block on the list)
- Blocks are not touching (the start + length of a block should not be the address of the next block on the list)

If no errors are detected, this function should return silently after performing these tests. If an error is detected, then an `assert` should fail and cause the program to terminate at that point.

```
void check_heap() {

  if (!freelist) return;
  freeNode* currNode = freelist;
  uintptr_t mins= \
currNode->size;

  < ……>
  assert (mins >= MINSIZE);
}
```

HW6 : using ‘extern’ (a shared global variable)

- Where does the free list head pointer live?
  - Needs to be accessible in both getmem and freemem implementation .c files.
- Could put it in a shared header file?
  - But, `int x;` allocates space for ‘x’ which is bad in a header file.
- Can we DECLARE ‘x’, but not DEFINE it?
  - Yes!: `extern int x;`
- Then in a .c file, you can actually define it (only in one file!).