Week 11: Proof of correctness

- Formal verification
- Symbolic execution
- Role of assertions
- Memory corruption

Proof of correctness

- Using symbolic logic, we may prove that an algorithm (or program function) conforms to its specification.
- But isn't it just as likely that we'll make an error in a formal proof as in a program?
- So, what good is such a "proof"?

Limitations of proof

- If it's valid, a proof shows only that an algorithm satisfies its specification.
- It can't determine whether the specification is appropriate.
- We should always fix a faulty spec. before worrying about proving correctness.

Symbolic execution

- Before running a unit-test, we often simulate execution on paper.
  - That used be called "desk checking".
- Advantage: We see the effect of each step (statement or group of closely-related statements)
  - You can also do that in any debugging environment that permits single-stepping
  - We often spot errors that way.
- Disadvantage:
  - It's tedious and
  - poorly suited to extremely complex logic or large number of variables
Example: Binary search

```c++
template<typename T>
int bts(const T tbl[], const T key, const int low, const int high)
{
    assert(low <= high);
    int lb = low, hb = high;
    while (lb <= hb)
    {
        int mp = (lb + hb) / 2; // Mid-point
        if (tbl[mp] == key) return mp; // Found!
        if (tbl[mp] > key) hb = m-1; // Bottom half
        if (tbl[mp] < key) lb = m+1; // Top half
    }
    return -(mp+1); // Not found!
}
```

Questions about our binary search function

- Is it always true that
  \[ lb \leq mp \leq hb \]?

- Is the return value
  - always the index of the matching element, if one is found?
  - never within the search bounds if no match is found?

- What will happen if `tbl` isn't properly sorted upon entry?
  - Do we need to validate that condition in testing?
  - Should we validate that `low` <= `high`?

Undefined results

- It's perfectly all right for a function specification to say a disclaimer, such as:
  ```c++
  // The result is undefined
  // if the array isn't sorted.
  ```
- provided that the function just produces the wrong result. It must not:
  - lose control
  - or corrupt some unrelated entity
- and should avoid
  - throwing an exception
- A program doesn't have to keep validating things that have already been validated.
  ```c++
  // How does OOP help?
  ```

To test binary search function

- We need at least one argument
  - matching a table entry
    - in the upper half
    - in the lower half
    - at the lower bound
    - at the upper bound
    - exactly in the middle
  - not matching any table entry
    - between two elements
    - preceding the first element
    - after the last element
    - anything else?
Memory corruption
- In assembly languages and in C (why?) it's possible for a program to refer to unallocated areas of memory (how?)
- Java and C# were designed to make such errors impossible (how?)
- In C++ such errors are still possible
  - But easy to avoid (how?)
  - Some say: inexcusable (why?).
- The difficulty in testing is that the symptom may appear long after the error occurred and in an unrelated part of the program.

Java & C# run-time protection
- Array subscripts are always checked
- So are string indices
- Dangling pointers (references) to objects that have been freed are impossible, because of reference counting and garbage collection.
- But those protections cost execution time
- You can implement those protections yourself in other languages.

The general pointer problem
- In C, we retrieve whatever a pointer points to by dereferencing the pointer \( *p \)
- You can add an integer to a pointer to do address arithmetic!
  \( * (p + k) \) retrieves the data item \( k \) positions after whatever \( p \) points to.
- This allows uncontrolled references to memory, as if you were coding in assembly language.
  Efficient but scary!

Another inexcusable error
- We keep reading in the trade literature about some serious software failure caused by a buffer overflow or overrun
- That occurs when the program tries to copy an array or a string into an area that isn't big enough.
- It's easy to do that in C! Why? How?
Background

- In C there are no character strings, but you can represent one by an array of `char`:
  - a pointer to the first character
  - a null character following the last character

- You reserve space by specifying a dimension in the usual way: `char a[n]`.
  - The array name is actually a pointer
  - The program can use either subscript notation `a[k]` or pointer notation `*(a+k)` to fetch a single character from the string `a`.

- Why did Ritchie design it that way?

Buffer overflow example

- `void fetchData(char* source)`
  - `char buffer[256];`  
  - `strcpy(buffer, source);`
  - `}`

  where `source` is untrusted (i.e. not under the control of this program)

- What's likely to happen?

- Why is this error *inexcusable*?

Buffer overflow

- Is a beginner's error. Experienced programmers know better than to trust strings they didn't create themselves.

- Nevertheless, surprisingly, programmers keep making this mistake (in haste?)

- Malicious (virus) programs take advantage by trying to trick a program into fetching a C string with no terminator.

What about other languages in the C family

- You can do the same thing in C++
  - But you shouldn't. Use a `String` class

- And even, with some limitations in C!

- Java and C# provide protection against out-of-bounds references to strings or arrays
  - There's a small cost.
Assertions

- Not officially part of the language, a useful library facility.
- Issues a diagnostic message (for the programmer, not the end user) and (usually) terminates execution.
- Meant to detect *impossible* conditions or validate *required* ones, not for general condition testing.

*Should they be left in production code?*

*What about our binary search function?*

N-Unit assertions

- N-Unit provides a range of verbs that do pretty much the same thing:
  - `Assert.IsTrue(condition)`
  - `Assert.IsFalse(condition)`
  - `Assert.IsNull(object)`
  - `Assert.IsEmpty(container)`
  - and so on.

*Do we need all of those?*

Binary search revisited

```cpp
template<typename T>
int bts(const T tbl[], const T key, const int low, const int high)
{
    assert (low <= high);
    int lb = low, hb = high;
    while (lb <= hb)
    { int mp = (lb + hb) / 2; // Mid-point
      if (tbl[mp] == key) return mp; // Found!
      if (tbl[mp] > key) hb = mp-1; // Bottom half
      if (tbl[mp] < key) lb = mp+1; // Top half
    }
    return -(mp+1); // Not found!
}
```

*Is `assert` appropriate? If not:*
  - What should we do instead?
  - Can we just take it out?
  - Then what happens if `high < low`?