Defensive Programming

Dawn Song

dawnsong@cs.berkeley.edu

Reasoning About Code

- **Functions make certain assumptions about their arguments**
  - **Caller must make sure assumptions are valid**
  - **These are often called *preconditions***
- **Precondition for `f()` is an assertion (a logical proposition) that must hold at input to `f()`**
  - **Function `f()` must behave correctly if its preconditions are met**
  - **If any precondition is not met, all bets are off**
- **Caller must call `f()` such that preconditions true – an obligation on the caller, and callee may freely assume obligation has been met**

Simple Precondition Example

```
int deref(int *p) {
    return *p;
}
```

- **Unsafe to dereference a null pointer**
  - **Impose precondition that caller of `deref()` must meet: *p* ≠ *NULL* holds at entrance to `deref()`**
- **If all callers ensure this precondition, it will be safe to call `deref()`**
- **Can combine assertions using logical connectives (and, or, implication)**
  - **Also existentially and universally quantified logical formulas**

Another Example

```
int sum(int *a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

- **Precondition:**
  - Forall j.(0 ≤ j < n) → a[j]≠NULL
  - If you're comfortable with formal logic, write your assertions this way for precision
- **Not necessary to be so formal**
  - Goal is to think explicitly about assumptions and communicate requirements to others

Postconditions

- ***Postcondition*** **for `f()` is an assertion that holds when `f()` returns**
  - `f()` has obligation of ensuring condition is true when it returns
  - Caller may assume postcondition has been established by `f()`
- **Example:**

```
/* Ensures: retval != NULL */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) {
        perror("Out of memory");
        exit(1);
    }
    return p;
}
```


Process for Writing Function Code

- **First write down its preconditions and postconditions**
  - Specifies what obligations caller has and what caller is entitled to rely upon
- **Verify that, no matter how function is called, if precondition is met at function's entrance, then postcondition is guaranteed to hold upon function's return**
  - Must prove that this is true for all inputs
  - Otherwise, you've found a bug in either specification (preconditions/postconditions) or implementation (function code)

Proving Precondition→Postcondition

- **Basic idea:**
  - Write down a precondition and postcondition for every line of code
  - Apply same sort of reasoning as for function
- **Requirement:**
  - Each statement's postcondition must match (imply) precondition of any following statement
  - At every point between two statements, write down *invariant* that must be true at that point
    - Invariant is postcondition for preceding statement, and precondition for next one

Example

- Easy to tell if an isolated statement fits its pre- and post-conditions
- Valid postcondition for "`v=0;`" is `v=0` (no matter what the precondition is)
  - Or, if precondition for "`v=v+1;`" is `v≥5`, then a valid postcondition is `v≥6`
- If precondition for "`v=v+1;`" is `w≤100`, then a valid postcondition is `w≤100`
  - Assuming `v` and `w` do not alias

Loop Invariant

- An assertion that is true at entrance to the loop, on any path through the code
  - Must be true before every loop iteration
    - Both a pre- and post-condition for the loop body
- Example: Factorial function code

```
/* Requires: n >= 1 */
int fact(int n) {
    int i, t;
    i = 1;
    t = 1;
    while (i <= n) {
        t *= i;
        i++;
    }
    return t;
}
```

  - Prerequisite: input must be at least 1 for correctness
  - Prove: value of `fact()` is always positive

Verifying Invariant Correctness

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int i, t;          /* n>=1 */
    i = 1;             /* n>=1 && i==1 */
    t = 1;             /* n>=1 && i==1 && t==1 */
    while (i <= n) {
       /* 1<=i && i<=n && t>=1   <-- loop invariant */
       t *= i;         /* 1<=i && i<=n && t>=1 */
       i++;            /* 2<=i && i<=n+1 && t>=1 */
    }                  /* i>n && t>=1 */
    return t;
}
```

- Easy if we examine each step:
  - Function's precondition implies invariant at function body start
  - Invariant at end of function body implies function's postcondition
  - If each statement matches invariant immediately before and after it, everything's OK
- That leaves the loop invariant…

Verifying the Loop Invariant

- **Loop invariant:** `1<=i && i<=n && t>=1`
- **Prove it is true at start of first loop iteration**
  - Follows from:
    - » $n \geq 1 \wedge i=1 \wedge t=1 \rightarrow 1 \leq i \leq n \wedge t \geq 1$
    - » if $i=1$, then certainly $i \geq 1$
- **Prove that if it holds at start of any loop iteration, then it holds at start of next iteration (if there's one)**
  - True, since invariant at end of loop body $2 \leq i \leq n+1 \wedge t \geq 1$ and loop termination condition $i \leq n$ implies invariant at start of loop body $1 \leq i \leq n \wedge t \geq 1$
- **Follows by induction on number of iterations that loop invariant is always true on entrance to loop body**
  - Thus, `fact()` will always make postcondition true, as precondition is established by its caller

Another Example: Recursion

```
/* Requires: n >= 1 */
int fact(int n) {
    int t;
    if (n == 1)
        return 1;
    t = fact(n-1);
    t *= n;
    return t;
}
```

- **Do you see how to prove that this code always outputs a positive integer?**

Analysis

```
/* Requires: n >= 1
   Ensures: retval >= 0 */
int fact(int n) {
    int t;
    if (n == 1)
        return 1;          /* n>=2 */
    t = fact(n-1);        /* t>=0 */
    t *= n;                /* t>=0 */
    return t;
}
```

- Before recursive call to `fact()`, we know:
  - `n`≥`1` (by precondition), `n`≠`1` (since if stmt didn't follow then branch), and `n` is an integer
  - Follows that `n`≥`2`, or `n-1`≥`1` (means precondition is met when making recursive call)
- Can conclude that `fact(n-1)` return value is positive from postcondition for `fact()`

Function Post-/Pre-Conditions

- Any time we see a function call, we have to verify that its precondition will be met
  - Then we can conclude its postcondition holds and use this fact in our reasoning
- Annotating every function with pre- and post-conditions enables *modular reasoning*
  - Can verify function `f()` by looking only its code and the annotations on every function `f()` calls
    - Can ignore code of all other functions and functions called transitively
  - Makes reasoning about `f()` an almost purely local activity

Documentation

- Pre-/post-conditions serve as useful documentation
  - To invoke Bob's code, Alice only has to look at pre- and post-conditions – she doesn't need to look at or understand his code
- Useful way to coordinate activity between multiple programmers:
  - Each module assigned to one programmer, and pre-/post-conditions are a contract between caller and callee
  - Alice and Bob can negotiate the interface (and responsibilities) between their code at design time

Avoiding Security Holes

- To avoid security holes (or program crashes)
  - Some implicit requirements code must meet
    - » Must not divide by zero, make out-of-bounds memory accesses, or deference null ptrs, …
- We can try to prove that code meets these requirements using same style of reasoning
  - Ex: when a pointer is dereferenced, there is an implicit precondition that pointer is non-null and in-bounds

Proving Array Accesses are in-bounds

```
/* Requires: a != NULL and a[] holds n elements */
int sum(int a[], size_t n) {
    int total = 0, i;
    for (i=0; i<n; i++)
        /* Loop invariant: 0 <= i < n */
        total += a[i];
    return total;
}
```

- Loop invariant true at entrance to first iteration
  - First iteration ensures `i=0`
- It is true at entrance to subsequent iterations
  - Loop termination condition ensures `i<n`, and `i` only increases
- So array access `a[i]` is within bounds

Buffer Overruns

- Proving absence of buffer overruns might be much more difficult
  - Depends on how code is structured
- Instead of structuring your code so that it is hard to provide a proof of no buffer overruns, restructure it to make absence of buffer overruns more evident
- Lots of research into automated theorem provers to try to mathematically prove validity of alleged pre-/post-conditions
  - Or to help infer such invariants

Pre-/Post-Condition Summary

- Looks tedious, but gets easier over time
  - With practice you can avoid writing down detailed invariants before every statement
    - » Think about data structures and code in terms of invariants first, then write the code
  - Usually can avoid formal notation, omit obvious parts, and only write down important ones
    - » Usually writing down pre-/post-conditions and loop invariant for every loop is enough
- Reasoning about code takes time and energy
  - Worth it for highly secure code

Defensive Programming

- Like defensive driving, but for code:
  - Avoid depending on others, so that if they do something unexpected, you won't crash – survive unexpected behavior
- Software engineering focuses on functionality:
  - Given correct inputs, code produces useful/correct outputs
- Security cares about what happens when program is given invalid or unexpected inputs:
  - Shouldn't crash, cause undesirable side-effects, or produce dangerous outputs for bad inputs
- Defensive programming
  - Apply idea at every interface or security perimeter
    - » So each module remains robust even if all others misbehave
- General strategy
  - Assume attacker controls module's inputs, make sure nothing terrible happens

Defensive Programming

- Write module *M* to provide functionality to a single client
  - *M* should provide useful responses if client provides valid inputs
  - If client provides an invalid input, then *M* is no longer under any obligation to provide useful output
    - » *M* must still protect itself (and rest of system) from being subverted by malicious inputs

Very Simple Example

```
char charAt(char *str, int index) {
    return str[index];
}
```

- Function is too fragile!
  - `charAt(NULL, any)` will cause a crash
  - `charAt(s, i)` causes a buffer overrun if `i` is out-of-bounds (too small or large) for `s`
- Neither can be easily fixed without changing function's interface

Another Simple Example with Many Flaws

```
char *double(char *str) {
    size_t len = strlen(str);
    char *p = malloc(2*len+1);
    strcpy(p, str);
    strcpy(p+len, str);
    return p;
}
```

- `double(NULL)` will cause a crash
  - Fix: test if `str` is a null ptr, and if so, return `NULL`
- Return value of `malloc()` is not checked
  - If out-of-memory, `malloc()` will return null ptr and call to `strcpy()` will cause program crash
  - Fix: test return value of `malloc()`
- If `str` is very long, then expression `2*len+1` will overflow, potentially causing a buffer overrun
  - 2^{31} byte input str on 32-bit machine will have 1 byte allocated, and `strcpy` will immediately trigger a heap overrun

Trickier Example: Java Sort Routine

- Accepts array of objects that implements `Comparable` interface and sorts them
  - Each object implements `compareTo()` method, and `x.compareTo(y)` must return a negative, zero, or positive integer, depending on whether `x` is less than, equal to, or greater than `y`
- Implementing a defensive sort routine is actually fairly tricky, because a malicious client could supply objects whose `compareTo()` method behaves unexpectedly
  - Calling `x.compareTo(y)` twice might yield two different results (if `x` or `y` are malicious)
  - Or, consider: `x.compareTo(y) == 1`, `y.compareTo(z) == 1`, and `z.compareTo(x) == 1`
- Sort routine might go into an infinite loop or worse

Some General Advice

- *1. Check for error conditions*
  - Always check return values of all calls (assuming this is how they indicate errors)
  - In languages with exceptions, can locally handle it or propagate (expose) to caller
  - Check error paths very carefully
    - » Often poorly tested, so they often contain memory leaks and other bugs
- What if you detect an error condition?
  - For expected errors, try to recover
  - Harder to recover from unexpected errors
  - Always safe to abort processing and terminate if an error condition is signaled (*fail-stop* behavior)

Some General Advice

- 2. *Validate All Inputs*
  - Sanity-check all inputs from rest of program
  - Treat external inputs (could be from adversary) with particular caution
  - Be conservative
    - » Better to limit inputs to expected values (might cause some loss of functionality) than to liberally allow all (might permit unexpected security holes)

What's Wrong with this Code?

```
char *username = getenv("USER");
char *buf = malloc(strlen(username)+6);
sprintf(buf, "mail %s", username);
FILE *f = popen(buf, "r");
fprintf(f, "Hi.\n");
fclose(f);
```

- Answer: If attacker controls `USER` environment variable, then could arrange for its value to be something like "`adj; /bin/rm -rf $HOME`"
  - `popen()` passes its input to shell for execution, and shell will execute command "`mail adj`" followed by "`/bin/rm -rf $HOME`"
- Solution: validate that username looks reasonable
  - If attacker can control other env vars (e.g., `PATH`), then could cause wrong `mail` command to be invoked → have to validate whole environment!

Advice: 3. *Whitelist, Don't Blacklist*

- **Common mistake:**
  - **When validating input from an untrusted source, trying to enumerate bad inputs and block them**
  - **Don't do that! Why?**
  - **Known as *blacklisting* (analogous to default-allow policy)**
  - **Can overlook some patterns of dangerous inputs**
- **Instead, use *whitelist* of known-good types of inputs, and block anything else**
  - **Default-deny policy (much safer)**

Whitelisting Example

- **Check a username using a regular expression:**
  - `[a-z][a-z0-9]*`
  -
```
char *validate_username(char *u) {
        char *p;
        if (!u || *u < 'a' || *u > 'z')
            die();
        for (p=u+1; *p; p++)
            if ((*p < '0' || *p > '9') &&
                (*p < 'a' || *p > 'z'))
                die();
        return u;
    }
```
- **Use with appropriate error-checking before using a user-supplied username**

More Advice

- **4. *Don't crash or enter infinite loops, Don't corrupt memory***
  - **Regardless of received inputs – NO abnormal termination, infinite loops, internal state corruption, control flow hijacks**
  - **Explicitly validate all inputs and avoid memory leaks**
  - **Defend against DoS attacks:**
    - » **Attacker supplies inputs that lead to worst-case performance (hashtable with O(1) expected, but O(n) worst case lookup)**

More Advice

- 5. *Beware of integer overflow*
  - Integer overflow often violates programmer's mental model and leads to unexpected (undesired) behavior
- 6. *Check exception-safety of the code*
  - Explicitly (programmer) thrown and implicitly (platform) thrown exceptions
  - Verify that your code doesn't throw runtime exceptions (null ptr deref, div 0,…)
  - Less restrictively, check that all such exceptions are handled and will propagate across module boundaries

Famous Example: Ariane 5

- Ariane 4 flight control sw written in Ada
  - Same software reused for more powerful Ariane 5
- Ariane 5 blew up shortly after first launch
  - Cause: uncaught integer overflow exception caused software to terminate abruptly…
- 16-bit reg: flight trajectory's horizontal velocity
  - Ariane 4 – verified range of physically possible flight trajectories could not overflow variable, so no need for exception handler…
- Ariane 5's rocket engine was more powerful, causing larger horizontal velocity to be stored into register triggering overflow…
  - Losses of around $500 million