CSE 503 Winter 2013 • David Notkin

Lecture #2 Notes: programs, behaviors, ambiguity

1. International Obfuscated C Code Contest

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define _                 ;double
#define void                    x,x
#define case(break,default)     break[O]:default[O]:
#define switch(bool)            ;for(;x<bool;
#define do(if,else)             inIine(else)>int##if?
#define true                    (--void++)
#define false                   (++void--)
char*O=" <60>!?\\\n"_ doubIe[010]_ int0,int1 _ Iong=0 _ inIine(int eIse){int
O1O=!O _ l=!O;for(;O1O<010;++O1O)l+=(O1O[doubIe]*pow(eIse,O1O));return l;}int
main(int booI,char*eIse[]){int I=1,x=-*O;if(eIse){for(;I<010+1;I++)I[doubIe-1]
=booI>I?atof(I[eIse]):!O switch(*O)x++)abs(inIine(x))>Iong&&(Iong=abs(inIine(x
)));int1=Iong;main(-*O>>1,0);}else{if(booI<*O>>1){int0=int1;int1=int0-2*Iong/0
[O]switch(5[O]))putchar(x-*O?(int0>=inIine(x)&&do(1,x)do(0,true)do(0,false)
case(2,1)do(1,true)do(0,false)6[O]case(-3,6)do(0,false)6[O]-3[O]:do(1,false)
case(5,4)x?booI?0:6[O]:7[O])+*O:8[O]),x++;main(++booI,0);}}}
```

2. Why is this program distasteful?
    a. Indentation?
    b. Documentation?
    c. Behaviors?
    d. Structure?
    e. Reasoning – loops, invariants?
    f. Fixing it?
    g. Changing it?
3. Programs have three immediate audiences
    a. The computer
    b. The developers
    c. The users
4. Are the audiences happy?
    a. Given that this program compiles and executes as intended, the computer is perfectly happy
    b. Under almost no conditions are the developers happy with this program
    c. What about the users?

5. Software engineering…is primarily concerned with the "happiness" of the software engineering team and with the "happiness" of the users – The "happiness" of the computer (performance, etc.) is material, but less so
6. We will focus more overall on the software engineering team than on the users – due largely to my knowledge and interests
7. The developers need to be able to – at reasonable cost, whatever that means – understand, reason about, fix, change, enhance, etc. the program
8. An aside: performance: quotations
    a. Michael Jackson
        i. Rule 1: Don't do it.
        ii. Rule 2 (for experts only): Don't do it yet.
    b. Bill Wulf: More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.
    c. Don Knuth: We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
9. Why is software engineering hard? (This in contrast to perceptions of what software engineering should be able to do.
    a. Validation vs. verification
        i. Building the system right (verification) vs. building the right system (validation) –Barry Boehm
        ii. Distinct objectives intertwined in non-obvious ways – the distinction itself is often poorly understood or ignored
        iii. Changes to the system's requirements cause changes to the implementation
        iv. Difficulties in implementation can cause (the need for) changes to the requirements
    b. Dominant discipline – Stu Feldman

10^3 Lines of Code	Mathematics
10^4 LOC	Science
10^5 LOC	Engineering
10^6 LOC	Social Science
10^7 LOC	Politics
10^+ LOC	???

    c. Design under constraints
        i. Software, like other engineered entities, is designed and built under constraints
        ii. Some of the constraints are explicit and many are implicit
        iii. Constraints are broad, ranging across customer needs, shipping deadlines, resource limitations (memory, power, money, etc.),

compatibility, reward structure, organizational culture, and much more…

d. A consequence of varied constraints
    i. There is *no single right way to engineer software:* no best programming language, design method, software process, testing approach, team structure, etc.
    ii. This does not imply that every approach is good under some constraints
    iii. Nor does it suggest that there are no consistent themes across effective approaches
    iv. But committing to a single "best approach" can be limiting
    v. "Please don't fall into the trap of believing that I am terribly dogmatical about [the goto statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!" –Dijkstra
    vi. "Don't get your method advice from a method enthusiast. The best advice comes from people who care more about your problem than about their solution." –M. Jackson
e. Complexity
    i. "Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one… In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound." —Brooks
    ii. Complexity and people – Dijkstra
        1. "The competent programmer is fully aware of the limited size of his own skull."
        2. "Software is so complex that our poor head cannot cope with it at all. Therefore, we have to use all possible means and methods to try to control this complexity."
f. Size
    50MLOC = 50 million lines of code
    50 lines/page-side ⇒ 1M page-sides
    1K page-sides/ream ⇒ 1K reams
    2 inches/ream ⇒ 2K inches
    2K inches = 167 feet ≈ twice the height of the Allen Center
    5 words/LOC @ 50 wpm ⇒ 50MLOC/5M min
    5M min = 83,333 hr = 3,472 days ≈ 10 years
    Just for typing … no fair thinking!
g. Design space complexity [Jackson]

        i. Designing both automobiles and bridges requires *specialized* knowledge
        ii. Automobile design is *standardized*: the designers know virtually everything about the context in which the automobile will be used: expected passenger weights, what kind of roads will be encountered, etc.
        iii. But bridge design is *not standardized*: the designers must understand the specific location in which the bridge will be built: the length of the span, the kind of soil, the expected traffic, etc.
        iv. Software design space is widely and wildly *non-standardized* (as well as being *specialized*). Figuring out what the user wants and needs is hard and is almost always part of the job; for most software systems, this goes far beyond designing a bridge for a specific location
        v. A classic exception is some classes of compilers
            1. The PQCC project at CMU (Wulf et al., 1980) led to the formation of Tartan Laboratories, which was acquired by TI (1996) primarily to construct C compilers for DSPs – in essence, this became standardized
            2. Jackson suggests that "compiler engineering" (and such) might make sense, in contrast to "software engineering"
        vi. All useful programs undergo continuing change
        Belady and Lehman (1976)
            1. Adding floors to skyscrapers, lanes to bridges
            2. Accommodating new aircraft at airports
            3. Adding Cyrillic-based languages to European Union documents
            4. Scaling software systems by an order of magnitude (pick your dimension)
            5. Supporting the web in a desktop productivity suite
            6. Adding support for Asian languages to a tool
            7. That is, a significant amount of "software maintenance" makes changes for which roughly analogous changes would be considered non-routine in most other fields

10. One more difficulty in more depth
    a. Dijkstra's1968 "**goto** considered harmful" letter to the editor of CACM is a classic
    b. Mark Twain: "A classic is something everyone wants to have read, but nobody wants to read."
    c. My version of his key argument
        i. We write programs but we care about executions – getting the behaviors we want is indirect

   ii. But reasoning about arbitrary behaviors is very hard due to the limits of the human brain
   iii. By reducing the gap between the program and the behaviors, we can do better in terms of reasoning

d. Example [Adapted from Wikipedia, Spaghetti Code]

```
10 i = 0
20 i = i + 1
30 PRINT i * i
40 IF i >= 10 THEN GOTO 60
50 GOTO 20
60 PRINT "Done"
70 END
```

Only one behavior in this example, but not straightforward to reason about (at least in general) – must simulate the control flow

e. Example continued: use **while** loop

```
i := 1;
while i <= 10 do
   PRINT i*i;
   i := i + 1
end;
PRINT "DONE"
```

Still only one behavior, but the loop is clearer – can more easily separate "doing the loop" and "exiting the loop"

Will allow invariants and proofs – easier reasoning

f. Böhm and Jacopini • CACM May 1966 [Wikipedia for additional history]
   i. They showed a construction that takes an arbitrary program and produces a program with equivalent behaviors that has a structured control flow graph that uses only sequencing (;), conditionals (**if-then**), loops (**while-do**)
   ii. Basic idea: encode the program counter in a program variable
   iii. So, what's the problem?

11. Programming languages research

   a. *Very* roughly, (my view is that) most programming languages research focuses on ways to reason about sets of behaviors through programs
   b. One program with (most often) an unbounded numbers of behaviors
   c. Changes are to the program, with the intent of achieving desired changes in the behaviors

12. Proofs-of-correctness
    a. A strong connection between the static program and the dynamic behaviors also enables proofs-of-correctness to be done precisely and formally
    b. Dijkstra, Hoare, Wirth, et al. did this in the late 1960's and early 1970's as step-wise refinement
       i. Pseudo-code is repeatedly expanded until the translation into programming language code is obvious
       ii. Choose a module
       iii. Decompose into smaller modules
       iv. Repeat until all modules are easily understood
       v. Provide explicit specification of the program, annotate it with assertions, use programming language semantics to prove those assertions
13. Basics of proofs-of-correctness
    a. In a logic, write down the *specification*
       i. the effect of the computation that the program is required to perform (the postcondition **Q**)
       ii. any constraints on the input environment to allow this computation (the precondition **P**)
    b. A Hoare triple is a predicate **{ P } S { Q}** that is true whenever **P** holds and the execution of **S** guarantees that **Q** holds
    c. To prove **{ P } S { Q}** requires
       i. a precisely defined logical meaning for each construct in the programming language
       ii. insertion of intermediate assertions to allow proofs to "flow" through the program
           1. **{ P } S { Q }**

              { P } S1; S2; if (..) S3 else S4 fi; S5 { Q }

              { P } S1 {A}; S2 {B}; if (..) S3 else S4 fi; {C} S5 { Q }
           2. Prove **{ P } S1 { A }**
              Prove **{ A } S2 { B }**
              Prove **{ B } if**… **{ C }**
              Prove **{ C } S5 { Q }**

d. Trivial examples

```
{ true }
y := x * x;
{y ≥ 0}

{ x <> 0 }
y := x * x;
{ y > 0 }

{ x > 0 }
x := x + 1;
{ x > 1 }

{ x = k }
if (x < 0)
  x := -x
fi;
{    ?    }

{    ?    }
x := 3;
{ x = 8 }
```

14. The objective is the proof
    [Example from Aldrich/Leino]; simply having **true** post-conditions is not sufficient

```
{ x = 5 } x := x * 2 { true }
{ x = 5 } x := x * 2 { x > 0 }
{ x = 5 } x := x * 2 { x  = 10 || x = 5 }
{ x = 5 } x := x * 2 { x  = 10  }
```

It is generally important to look for the logically strongest post-condition – that is, one that represents the most restrictive assertion consistent with the specification or with intermediate assertions

15. Weakest preconditions [Example from Aldrich/Leino]
    a. Here are a number of valid Hoare Triples

```
{x = 5 && y = 10} z := x / y { z < 1 }
{x < y && y > 0}  z := x / y { z < 1}
{y ≠ 0 && x / y < 1} z := x / y { z < 1 }
```

The last one is the most useful because it allows us to invoke the program in the most general condition – it is called the *weakest precondition,* **wp(S,Q)** of **S** with respect to **Q**

   b. If **{P} S {Q}** and for all **P'** such that **P' => P**, then **P** is **wp(S,Q)**

16. Conditional execution – formal semantics of "if-else"
    {P} if C S1 else S2 fi {Q}

```
{true}
if x >= y
  max := x
else
  max := y
fi
{(max >= x ∧ max >= y)}
```

    Is this a good post-condition? Does it do what we "want" or "expect"? Formalism doesn't eliminate all confusion. We likely want
    $(max = x \vee max = y) \wedge (max \geq x \wedge max \geq y)$

17. In essence, every specification is satisfied by an infinite number of programs and vice versa: formalism is much more useful in showing you've built the system right than in showing you've built the right system
18. Assignment statements
    {Q(E)} x := E {Q(x)}

    If we knew something to be true about **E** before the assignment, then we know it to be true about **x** after the assignment (assuming no side-effects)

```
{y > 0}
 x := y
{x > 0}

{x > 0}          [Q(E) ≡ x + 1 > 1 ≡ x > 0 ]
 x := x + 1;
{x > 1}          [Q(x) ≡ x > 1]
```

19. Loops: **{P} while B do S {Q}**
    a. We can try to unroll this into

       {P ∧ ¬ B} S **{Q}** ∨
       {P ∧ B} S**{Q ∧ ¬B}** ∨
       {P ∧ B} S **{Q ∧ B} S {Q ∧ ¬B}** ∨ …

But we don't know how far to unroll, since we don't know how many times the loop can execute

b. The most common approach to this is to find a *loop invariant*, which is a predicate that
    i. is true each time the loop head is reached (on entry and after each iteration)
    ii. and helps us prove the post-condition of the loop
    iii. The loop invariant approximates the fixed point of the loop
    iv. Three steps: find **I** such that…
        1. $\mathbf{P \Rightarrow I}$ –Invariant is correct on entry
        2. $\mathbf{\{B \wedge I\}\ S\ \{I\}}$ –Invariant is maintained
        3. $\mathbf{\{\neg B \wedge I\} \Rightarrow Q}$ –**Q** is true when loop terminates

```
{n > 0}
  x := a[1];
  i := 2;
  while i <= n do
    if a[i] > x then x := a[i];
    i := i + 1;
  end;
{x = max(a[i],…,a[n])}
```

c. Termination
    i. Proofs with loop invariants do not guarantee that the loop terminates, only that it does produce the proper post-condition if it terminates – this is called *weak correctness*
    ii. A Hoare triple for which termination has also been proven is *strongly correct*
    iii. Proofs of termination are usually performed separately from proofs of correctness, and they are usually performed through well-founded sets

20. Coming up next week: Proving properties of abstract data types
    a. Separate proofs of the specification (e.g., properties like **x = S.top(S.push(x))** and of the concrete implementation of the methods (**top**, **push**, etc.)
    b. Define an *abstraction function* that gives a mapping from instances of the concrete representation to the abstract representation
    c. Define a *representation invariant* that holds across all legal instances of the concrete representation

21. Open issues
    a. Automation – proof engines, proof assistants, etc.

b. Programming language dimensions – side-effects, procedures/methods (and parameter passing), non-local control (e.g., exceptions), classes/objects etc., other language paradigms (e.g., functional), …
c. Whence post-conditions?
d. How much of a proof needs to be redone if the specification and/or the program changes slightly?
e. The original promise of program verification has not been achieved, at least to the degree many anticipated
    i. At the same time, as we'll see, it's clear that the underlying techniques have made a huge difference and have supported a shift from trying to prove big theorems about little programs to trying to prove little theorems about big programs
    ii. Aside: type-checking is in the second category

22. Debunking a myth
    a. A culture – at least in the research world – developed in part due to this proof-centric view of the world
    b. Roughly this culture says it is crucial to prove properties of programs over all possible executions – otherwise the other executions may have unexpected behaviors
    c. That is, sampling of the behaviors ("testing") is inherently problematic
    d. Sources of unsoundness: Dwyer et al.
        i. Matt Dwyer's talk at ICSE 2007 put much of this issue in perspective: in my words, he argues that it's *all* sampling
        ii. Dynamic techniques sample across executions (behaviors)
        iii. The "hope" is that some behaviors are characteristic of other behaviors
        iv. Static techniques sample across properties (requirements)
        v. The "hope" is that some requirements are good proxies for other requirements (e.g., type-safe and deadlock-free build confidence in correctness)
        vi. What we need to know is the degree of unsoundness; that is, we need to know what we know, and what we don't know
        vii. It really shouldn't be static "versus" dynamic – each has strengths, each has weaknesses: this is increasingly recognized in research – but not by everybody!