Foundations, Reasoning About Algorithms, and Design By Contract
CMPSC 122

I. Logic 101

In logic, a **statement** or **proposition** is a sentence that can either be true or false.

A **predicate** is a sentence in terms of a finite number of variables that becomes a statement whenever values are substituted for those variables.

**Examples:**
- A statement: 
  - Not a statement:
- A predicate: 
  - A value that makes it true: 
  - A value that makes it false:

Okay, so what does this have to do with programming? Quite a lot, in fact! Any statement is either true or false and programs establish the state of memory and produce side effects -- things like messages that uses see -- and we can formally write down what’s going on. Sometimes you’ll see statements about what’s going on in a program referred to as **assertions**. We can use them both to communicate about algorithms and to reason about their correctness.

There are three primary kinds of assertions that are fundamental in computer science: preconditions, postconditions, and loop invariants. We’ll look at each in detail and write them as we proceed.

While we’ll get a little taste of the idea of reasoning about algorithms in this course, your formal training about logic will come in CMPSC 360 and you’ll work more with formal correctness proofs there and in CMPSC 465 (both of which have this course as a prerequisite).

II. Design By Contract

Lawyers and businesses use contracts all the time. We steal the idea here.

**Question:** Suppose your friend who’s doing a science experiment knows you’re studying computer science and figures you could help with calculations. Your friend is shooting projectiles in a lab and wants to know if, say, some object will have hit the ground 1 second after the launch. What should be in contract for you to agree to compute results safely? What might your friend want in the contract?

**Second Question:** What if, instead of one trial, your friend had several hundred trials of the same experiment and wanted you to crunch some statistics quickly? What would you want to put in the contract to agree to do this work?
So, let’s make the language a little more generic and precise all at the same time. We’re usually talking about algorithms, but when we do, we tend to break them up into little chunks -- functions or methods. (Important tangent: What’s the difference between a method and a function?)

We then specify each method by a few things:

- Its header, i.e. its name and the names of inputs and outputs (what we call the parameters)
- Its precondition, i.e. a predicate that the method assumes is true in doing whatever it does
- Its body, i.e. the algorithm and logic inside the method
- Its postcondition, i.e. a predicate that is guaranteed to be true after the method has completed

Two discussion points:

1. I said the precondition and postcondition were predicates, so that implies they’re written in terms of some variables. Where do these variables come from?

2. Where is the “contract” in this?

Writing preconditions and postconditions requires using very precise language, really rooted in the language of mathematics. Clarity should always be a major goal. But, mathematicians are lazy, so we have some shorthand you’ll see used commonly:

- PRE is the precondition
- POST is the postcondition
- FCTVAL means “function value,” i.e. whatever a function computes and returns.

Example:
Let’s formally write down the physics example defined earlier.
III. More Issues to Consider for PRE and POST

First, here are some guidelines for when to include preconditions and postconditions:

1. Every method (except main) needs a POST because every method does something. Here are some common questions to consider:
   a. Does the method change the state of any variables?
   b. Does the method return anything?
   c. Does the method produce any output? (Consider the screen, files, and databases.)

2. A method needs a PRE if its behavior depends on something external to work correctly. Here are some common external factors to consider:
   a. Are there any input parameters? Address each one.
   b. Does the method depend on a file? (Consider the path and format.)
   c. Does the method depend on a connection to an external resource, like a database, being open?

Here are some important guidelines for clear PRE/POST (and to save you time too):

1. Use complete sentences. (Mathematical operators like "==" count as verbs, though.
2. Use variable names explicitly:
   a. All parameters must be referenced by name.
   b. Remember this is external documentation: Reference only the names of parameters (from this function's header)
      or class data members (we'll learn about those very soon). Never reference local variables.
3. Write abstractly: The "someone else" using your method doesn't care about what's going on inside the braces, doesn't want to read any of that code, doesn't really care about your algorithms within. He or she only cares about the end result. So, PRE/POST should only focus on things in the parameters and factors external to the method.
4. Write specifically: Address details such as format (e.g. "one value per line" or "comma-separated"), units, and valid ranges.
5. Write concisely: Write the strongest condition possible and don't state the obvious.

You'll get more practice with these issues in the homework and in programming assignments.

IV. Abstraction and Information Hiding

I have some questions for you:

- Can you drive a car?
- Can you send a message with a cell phone?
- Can you fix my car when it’s broken?
- Can you explain what went on to get the message from your cell phone to your friend’s phone?

Okay, this brings us to the principle of abstraction:

More examples:

So where does this fit in with software?
Another fundamental (and certainly related but \textit{not the same}) principle is that of \textbf{information hiding}:

The user of a method (or client) should only know what is necessary to use the method and nothing more. The implementor of a method should only know what is necessary to implement the method and nothing more.

So…

- What does the client \textit{not} need to know to use a method?
- What does the implementor \textit{not} need to know to implement a method? (Why is this so important?)
- What do both need to know?

Ah, so design by contract is not only about reasoning about programs, but also about communication.

\textbf{V. Loops}

Let’s take a moment to review some things about loops. But, let’s initially not concern ourselves with syntax, but instead speak of loops in general.

First, all loops include a \textbf{guard or loop test}, which is a logical statement that is true for the loop to continue iterating. This might be something simple like $x < 10$ or something like $x < 5 \text{ and } \neg (\text{temp} > 100 \text{ or } \text{numValid} > \text{MIN})$.

Next, loops can be broken down into two categories:

- \textbf{determinate loops} - those for which the __________________________________ is known

- \textbf{indeterminate loops} - those for which the number of iterations is not known until __________________________________

\begin{itemize}
  \item \textbf{Examples:}
  \begin{itemize}
    \item \textbf{Determinate loop applications:}
    \item \textbf{Indeterminate loop applications:}
  \end{itemize}
\end{itemize}

The guard can be in the form of a \textbf{pretest}, i.e. a test that is checked before an iteration of a loop to see if that iteration should be executed, or in the form of a \textbf{posttest}, i.e. a test that is checked after an iteration of a loop to see if another iteration should be executed.
Question: How does C++ implement each of these kinds of loops?

Each construct a programming language provides has a reason for existing, and, while you can twist some tools to fit other scenarios (really, though, that’s abuse of the tools), your code is cleanest and most natural when you use the right tools for the right job. Let’s summarize the ideal application of loops here:

In this course, you’ll be expected to make the ideal choices. (People will appreciate your code later on if you keep this up!)

VI. Pseudocode vs. Real Code

You may notice in the loop test example I gave above, I didn’t quite use code from a language you know. Sometimes, in reasoning about algorithms, we will prefer to leave out syntax details that are specific to any one language and write the algorithms down using pseudocode. Pseudocode is nice, because it allows us to get to the real “meat” of the issue at hand, omitting details that have more to do with the language. Thus, it is also language independent.

There isn’t a prescribed syntax for pseudocode; it’s mixture of English, math, and things you see in most programming languages. It leaves out variable types and declarations and leaves out a lot of the “punctuation” you see in many languages.

In many lessons where it’s appropriate, I’ll start with pseudocode to emphasize the essential computer science ideas before getting into language-specific syntax. Since you have some programming experience under your belt, most everything in pseudocode will be obvious to you, but don’t hesitate to ask questions when you have them.

VII. Loop Invariants

Now back to assertions…

Question: What does “vary” mean? What, then, would something that is “invariant” do?

A loop invariant is a predicate that is true at some point in the body of a loop, every time that point is reached, regardless of which iteration is being executed.

Now, while an invariant could technically be written anywhere in a loop, it’s silly (and problematic later) to write them about somewhere in the middle of the loop. Writing an invariant about the end makes some sense, but the ideal and standard practice is to write loop invariants about the state and the start of an iteration of a loop. Thus, pretty much every invariant you ever write should start “Before the start of the i-th iteration, …”
Example: Trace the first few iterations of and write an invariant for the following loop:

\[
\begin{align*}
    \text{sum} &= 0 \\
    \text{for } i = 1 \text{ to } 10 \\
    \{ & \quad \text{sum} = \text{sum} + i \\
    \}
\end{align*}
\]

Example: Write an invariant for the following loop, assuming code that precedes it has initialized \( \text{key} \) and \( \text{vals}[0..\text{LAST}] \):

\[
\begin{align*}
i &= 0 \\
\text{found} &= \text{FALSE} \\
\text{while } i \leq \text{LAST} \text{ and not found} \\
\{ & \quad \text{if } \text{vals}[i] = \text{key} \\
& \quad \quad \text{found} = \text{TRUE} \\
& \quad i = i + 1 \\
\}
\end{align*}
\]

Problem: Write an invariant for the following loop, assuming code that precedes it has initialized \( \text{key} \), \( TOL \), and \( \text{vals}[0..\text{LAST}] \):

\[
\begin{align*}
i &= 0 \\
\text{found} &= \text{FALSE} \\
\text{while } i \leq \text{LAST} \text{ and not found} \\
\{ & \quad \text{if } \text{vals}[i] - TOL \leq \text{key} \leq \text{vals}[i] + TOL \\
& \quad \quad \text{found} = \text{TRUE} \\
& \quad i = i + 1 \\
\}
\end{align*}
\]
**Problem:** Consider this loop, which assumes *input* is initialized and array *f* has been allocated with size 100:

\[
i = 0
\]

\[
\text{for } j = 1 \text{ to } \text{input}/2 \\
\{
\quad \text{if } \text{input} \ mod \ j = 0 \\
\quad \{
\quad \quad f[i] = j \\
\quad \quad i = i + 1
\quad \}
\}
\]

a. First trace the execution of this loop for *input* = 20

b. Write an invariant for this loop.

---

**VIII. A Preview of Correctness Proofs**

**Question:** If you test your program on one input, are you sure your program is correct? 2 inputs? 3? 1000?

We can prove a algorithm is correct using some tools from mathematics. Really, it’s all logic, but you’ll need some formal training in that (and especially in a technique called mathematical induction) to be able to write a full correctness proof for most “interesting” algorithms. You’ll get that in 360, but I want you to be aware of the big picture now. Suppose we have an algorithm (packaged in a method) that contains one loop. Then the correctness proof of that algorithm involves three parts:

- **Initialization:** Assume that the precondition is true and use that and the code before the loop to establish why the loop invariant is true before the loop begins.
- **Maintenance:** Assume that the loop invariant is true at the start of any arbitrary iteration and use the loop body to prove that the invariant is still true at the end of that iteration. (Whoa! Okay, don’t panic…)
- **Termination:** Assume that the loop invariant is true at the end of the last iteration and that the guard is false (since the loop is done) and use whatever code is after the loop to prove that the postcondition is true.

I urge you to think about the initialization and termination steps while you’re programming this semester.

**IX. Closing Opening Thoughts**

There are some exercises I have for you to do on a separate worksheet. They draw upon both what I’ve taught you here and some concepts often taught in a first-semester computer science course (CS1). I have a link on the course site to my own CS1 notes you can use if you need to look anything up. All of the language in the review worksheets is language I will use in this course and expect you to know, both to understand the lectures, but also to read or respond to exam and quiz questions.