Invariants- CS 215

You’ve been familiar with the use of pre and postconditions from CS 110. In short, a precondition is something that must be true before a procedure executes and a postcondition is what the execution of the algorithm guarantees. We use pre and postconditions to assume what must be true and to guarantee what we produce are correct. However, until now, you have not seen a formal linkage between the pre and postcondition – what happens to transfer the precondition into the postcondition?

Pre and postconditions are more powerful than just procedure conditions; statements may also have pre and postconditions. The general definition is:

Given a statement, S, precondition P, and postcondition Q, if S begins with P true, then if S terminates, Q is true.

Notice that this still works for procedures, but also extends your prior usage to all for procedures to be collection of statements, or individual statements.

As a simple example, consider the statement, \( x=x+2; \) If we expect \( x=4 \) after the statement executes the precondition must be \( x=2 \). This is written as follows:

Precondition: \( x=2 \)  
\( x=x+2; \)
Postcondition: \( x=4 \)

A very interesting use of this is in understanding and correctly constructing loops. A loop has three main parts, an initialization, a body, and a conditional.

\[ \begin{align*}
\text{Initialization} & \quad x=2 \\
\text{While conditional} & \quad x=x+2; \\
\text{Body} & \quad x=4
\end{align*} \]

Corresponding to these are the conditions of Initialization, Maintenance, and Termination.
- Initialization can be though of as what is true before the loop executes, i.e. a precondition.
- Maintenance can be thought of as both a pre and postcondition on the loop body.
- Termination is the postcondition of the loop.

Loops are tricky because the Initialization, Maintenance and Termination are all related by something called an Invariant. Since, in general, we don’t know how many times a loop will be executed, we have to always start the body with the invariant true, and make it true again at the end of the body before the start of another loop iteration.
In outline form this becomes:

Precondition: \( \{ I \} \) \hspace{1cm} (1)
while (condition) \hspace{1cm} (2)
Invariant \hspace{1cm} \{ I \land \text{condition} \} \hspace{1cm} (3)
and condition
Loop Body \hspace{1cm} (4)

Invariant: \hspace{1cm} \{ I \} \hspace{1cm} (5)
end while \hspace{1cm} (6)
Postcondition: \( \{ I \land \neg \text{condition} \} \) \hspace{1cm} (7)

Example
Let’s see how this might work in practice for the following simple program that sums up the first \( n \) integers.

\[
i=0; \\
\text{sum}=0; \\
\text{while } i<n \\
\hspace{1cm} \text{sum} = \text{sum} + i; \\
\hspace{1cm} i=i+1; \\
\text{end while}
\]

Note, we’ve already stated our postcondition in words (find it). Now, let’s try to write it in symbols used by the program.

Postcondition: \( \{ Q : \text{sum} = \sum_{j=1}^{n} j \} \)

Does this postcondition result from the program (actually no, there’s an error in the program, let’s see if we can find it).

For the Postcondition to be reached it must be implied by \( I \land \neg(i < n) \) Here
\( I \land \neg(i < n) \Rightarrow \text{sum} = \sum_{j=1}^{n} j \). By looking at the code, we can see that \( \neg(i < n) \) really means \( i=n \). Since the code builds up a sum through increasing values of \( i \), a good candidate for \( I \) is \( I : \text{sum} = \sum_{j=1}^{i-1} j \).
Let see what we have so far

\[
i = 0; \\
\text{sum} = 0; \\
I: \text{sum} = \sum_{j=1}^{i-1} j
\]

while \(i < n\)

\[
\text{sum} = \sum_{j=1}^{i-1} j^\langle i < n \rangle \\
\text{sum} = \text{sum} + i; \\
i = i + 1;
\]

\[
I: \text{sum} = \sum_{j=1}^{i-1} j
\]

end while

\[
Q: \text{sum} = \sum_{j=1}^{n} j
\]

Let’s try this to see if it is correct. Set \(n = 1\), then on the first pass through the loop,

\[
i = 0 \\
\text{sum} = 0 \\
I \text{ holds} \]

while \(0 < n\)

\[
I \wedge (0 < n) \text{ holds} \\
\text{sum} = 0 \\
i = 1;
\]

\[
I: \text{sum} = 0 = \sum_{j=1}^{i-1} j = 0 \text{ so I holds}
\]

return to the while loop

while \(1 < n\)

the while loop fails so we end it and evaluate the postcondition:

\[
I \wedge \neg(i < n) \Rightarrow \text{sum} = \sum_{j=1}^{n} j
\]

\[
\text{sum} = \sum_{j=1}^{i-1} j^\langle i < n \rangle \Rightarrow 0 = \sum_{j=1}^{i-1} j^\langle 1 < 1 \rangle = 0
\]

which does not imply \(\text{sum} = 0 = \sum_{j=1}^{1} j = 1\)
What went wrong? Note that the program actually sums up the first n-1 integers, not the first n, since the while condition is incorrect. It should be while \( i \leq n \).

Fixing the program yields

\[
\begin{align*}
&i=0; \\
&\text{sum}=0; \\
&I : \text{sum} = \sum_{j=1}^{i-1} j \\
&\text{while } i \leq n \\
&\quad \text{sum} = \sum_{j=1}^{i-1} j^{(i < n)} \\
&\quad \text{sum}=\text{sum}+i; \\
&\quad i=i+1; \\
&I : \text{sum} = \sum_{j=1}^{i-1} j \\
&\text{end while} \\
&Q : \text{sum} = \sum_{j=1}^{n} j
\end{align*}
\]

Note that \( I \) does not hold, here

Now the Invariant is maintained and the postcondition, \( Q \), is reached (try a few examples, yourself). Note, however, that the Invariant is maintained only at the top and bottom of the loop, not necessarily inside the loop body. Between the increment of sum and the increment of \( i \), the Invariant does not hold.

In class we will fit the book’s examples into this scheme, showing how the loop counter (usually) and the maintenance of the invariant imply the postcondition.

**Assert Statements**

Many companies, Microsoft included, have been experimenting with putting assertions into their code for testing and debugging. The C `assert` pragma allows logical conditions to be evaluated when the program is run. While most people put in random assertions, we can use our knowledge of invariants to make more intelligent assertions based on preconditions, postconditions, and invariants. We can re-write the above skeleton using asserts as follows.

\[
\begin{align*}
\text{assert} & (I) & (1) \\
\text{while} & (\text{condition}) & (2) \\
\text{assert} & (I \&\& \text{condition}) & (3) \\
& \text{Loop Body} & (4) \\
\text{assert} & (I) & (5) \\
\text{end while} & (6) \\
\text{assert} & (I \&\& \neg\text{condition}) & (7)
\end{align*}
\]