Verification Logic
Michael Main
CU-CS-CT002-97

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE
ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.
Verification Logic

Michael Main
(main@cs.colorado.edu)
Computer Science Department
University of Colorado at Boulder

Research Reports on Curriculum and Teaching
CU-CS-CT002-97
December 7, 1997

Abstract. This is a survey of verification logic. It defines the meaning of a verification formula and a proof outline to express the correctness of a programs written in C, including assignments, conditional statements and while loops. The survey is appropriate for a one- or two-week survey of verification logic for an undergraduate class in principles of programming languages.
Verification Logic

I knew that programs could have a compelling and deep logical beauty...

EDSGER W. Dijkstra
A Discipline of Programming

1. PROGRAM ASSERTIONS
2. VERIFICATION FORMULAS
3. VERIFICATION OF ASSIGNMENT STATEMENTS
4. VERIFICATION OF CONDITIONAL STATEMENTS
5. VERIFICATION OF LOOPS
SUMMARY AND EXERCISES

Verification logic is a method for reasoning about programs and proving program correctness. The method is analytical, which means that you'll analyze programs and prove their properties without actually running them. This is different than program testing, which actually runs programs on carefully selected data to increase our confidence in their correctness. Generally, program testing is insufficient to prove that a program always works, and this shortcoming is one of the motivations for developing analytical methods. The analytical techniques are also relevant to program design, programming language design and other fundamental issues.
1. PROGRAM ASSERTIONS

Program assertions are boolean expressions indicating what is known at certain points in a program. Or to be more accurate, an assertion indicates what should be true at a given point in a correct program—though it’s always possible that the program has mistakes, resulting in incorrect assertions. Some programming languages allow you to place assertions in a program, and the assertions are checked when the program runs. For example, C or C++ provides the assert facility which is obtained by using the include directive:

```c
#include <assert.h>
```

The primary function in the assert facility is a function named `assert`, which has one argument. The argument may be any integer expression, but usually the argument is a true/false expression (that is, an expression where zero indicates “false” and any nonzero value indicates “true”). The `assert` function will evaluate the expression. If the result is true, then no action is taken. But if the result is false, then a useful error message is printed, and the program is halted. For example, suppose a program wants to assert that an integer variable called `hour` lies in the range 0 to 23 as some point in the program. This is accomplished with the assertion:

```c
assert((0 <= hour) && (hour <= 23));
```

If the expression `(0 <= hour) && (hour <= 23)` is true, then the assertion is true and the `assert` function will take no action. On the other hand, if the expression is false, then the assertion is false, so `assert` will print a message and halt the entire program.

Some programmers worry about the fact that many assertion checks will make a program run slower. In C++ this is not a problem because after a program is completely debugged, you can recompile with an extra directive, like this:

```c
#define NDEBUG
#include <assert.h>
```

The `NDEBUG` indicates a “no debugging” mode, in which all assertions are ignored. Anyway, putting assertions in a program or function can help during debugging and can serve as part of the program’s documentation. Often, a few strategically placed assertions suffice to show how a function works.

From the standpoint of showing how a function works, it may also be useful to have assertions that are more powerful than C++ boolean expressions. For example, the binary search procedure of Figure 1 includes assertions that are written in English such as “data is an array of at least n double numbers, sorted from small to large.” Two of these English assertions are useful enough that we give them names at the top of the function’s documentation:
// Assertions used within this function:
// SORTED: data is an array of at least n double numbers, sorted from
// small to large.
// TARGET AREA IS LIMITED: If target appears in the first n elements
// of data, then it must be at or after data[a] and strictly before
// data[z].

The binary search function is notoriously difficult to write correctly, making sure that you handle boundary cases such as the case where n=0, the case where everything is smaller than the target, and the case where everything is bigger than the target. So even simple assertions for documentation will increase our confidence that a program is correct. And if the programming language supports checking some assertions, that's all the better. But even this would just be program testing, which doesn't guarantee correct assertions. Our eventual goal goes beyond just testing—the goal of verification logic is to provide analytical proofs that assertions are correct. In other words, we want to be able to analyze a program, proving that its assertions are correct without actually running or testing the program. These proofs involve verification formulas, which we’ll look at next.

2. VERIFICATION FORMULAS

Verification logic revolves around verification formulas, which have three parts, as in this example:

```c
/* (i == A) && (j == B) */       — Part 1: The precondition
temp = i;
i = j;
j = temp;
/* (i == B) && (j == A) */       — Part 3: The postcondition
/* (i == A) && (j == B) */       — Part 2: A code fragment
```

A verification formula, such as this, represents the following statement about the behavior of the program fragment:

```c
/* (i == A) && (j == B) */       — If the precondition is
 temp = i;
i = j;
j = temp;
/* (i == B) && (j == A) */       — true at this point...
/* (i == B) && (j == A) */       — ...and we execute the code...
/* (i == A) && (j == B) */       — ...and the program reaches this
```

point, then the postcondition will be true.
bool search(double data[], size_t n, double target)
  // Assertions used within this function:
  // SORTED: data is an array of at least n double numbers, sorted from small to large.
  // TARGET AREA IS LIMITED: If target appears in the first n elements of data, then it
  // must be at or after data[a] and strictly before data[z].
  // Precondition: SORTED.
  // Postcondition: If target occurs in the first n elements of data, then the return value is
  // true; otherwise the return value is false.
{
  // The values a, z and mid are indexes to the array
  size_t a = 0;
  size_t z = n;
  size_t mid;

  /* Assertion: SORTED and TARGET AREA IS LIMITED. */
  while ((a < z) && (data[(a+z)/2] != target))
  {
    /* Assertion: SORTED and TARGET AREA IS LIMITED
    ** and (a < z) and (data[(a+z)/2] != target). */
    mid = (a+z)/2;
    if (data[mid] < target)
      a = mid+1;
    else
      z = mid;

    /* Assertion: SORTED and TARGET AREA IS LIMITED. */
  }

  /* Assertion: SORTED and TARGET AREA IS LIMITED
  ** and ( (a >= z) or (data[(a+z)/2] == target)). */

  // Note: If (a < z) then the assertion implies that the target appears at data[(a+z)/2].
  // On the other hand, if (a >= z), then TARGET AREA IS LIMITED implies that the
  // target cannot be in the array.
  return (a < z);
Verification logic is aimed at showing that verification formulas are *correct*—that if the precondition is true, and we execute the program fragment, and the fragment reaches the end, then the postcondition will be true. The example above is correct, because:

- if \(i\) is equal to \(A\), and \(j\) is equal to \(B\) before the three assignment statements...
- ... and we execute the three assignment statements...
- ... then at the point where the three assignments are finished, \(i\) is equal to \(B\), and \(j\) is equal to \(A\).

This is a common kind of verification statement, where the precondition indicates initial values (in this case \(A\) and \(B\)) of variables (in this case \(i\) and \(j\)) and the postcondition tells us how those values have changed. Also note that we are writing the assertions as C++ comments, surrounded by /* and */. This will allow us to use either ordinary boolean expressions or English statements of the assertions.

Sometimes, the assertions will provide relationships between variables, rather than exact values, as in these two examples:

First example:
```cpp
/* (i < j) */
temp = i;
i = j;
j = temp;
/* (j < i) */
```

Second example:
```cpp
/* (true) */
if (i < j)
    min = i;
else
    min = j;
/* (min <= i) && (min <= j) */
```

The example on the right illustrates an interesting precondition: the simple boolean expression `true`. Of course, this expression is always true, so we use it when there's no extra constraints that need to be placed on the variables before executing the program fragment. In effect, the right-hand example claims that whenever the if-then-else statement executed, it will finish with \(\text{min} \leq i\) and \(\text{min} \leq j\), without any special conditions on how things started out.

It's easy to convince yourself of the correctness of these small formulas, which primarily involve a few assignment statements. But for larger formulas, you need an analytical method to demonstrate correctness: a collection of requirements so that if a verification formula meets these requirements, then it is correct, no matter
how complicated the program may be. In the ideal situation, you can write a formula, check that it meets the requirements, and be confident that your program is correct—all from the comfort of your desk, without ever executing a single line of code. But before tackling complicated program fragments, we need a simple rule to deal with a single assignment statement.

3. VERIFICATION OF ASSIGNMENT STATEMENTS

Let’s look at assignment statements involving integers. There’s nothing special about integers, but it’ll make it easier if all the variables in this section are integers. Here’s an example where the precondition has been intentionally omitted:

```c
/* ... */
i = j+42;
/* i > 0 */
```

This assignment statement sets i to 3+42, and afterward we’d like i to be greater than 0. What needs to be true before the assignment, in order for i to be greater than 0 afterward? There’s one obvious answer: in the precondition we require j+42 to be greater than 0. The complete formula looks like this:

```c
/* (j+42) > 0*/
i = j+42;
/* i > 0 */
```

Here are some similar formulas, with the precondition and postcondition written on the same line as the assignment:

```c
/* (j+42) < 0 */  i = j+42;  /* i < 0 */
/* ((j+42) % 7) = 3 */  i = j+42;  /* (i % 7) = 3 */
/* (j+42) = 9 */  i = j+42;  /* i = 9 */
/* (j+42)*(j+42) > 55 */  i = j+42;  /* i*i > 55 */
/* (j+42) is a borogove */  i = j+42;  /* i is a borogove */
```

The last example is kind of silly, especially if you don’t know what a borogove is. We’ll get back to that in a moment, but first can you see the pattern in all the examples? The precondition is identical to the postcondition, except that whenever i occurs, it is replaced by (j+12) in the precondition. In other words:

- if we want ...i... to be true after the assignment i = j+42, then we need ... (j+42) ... to be true before the assignment.

This is an interesting pattern, because we really don’t even need to know what the postcondition means in order to apply the rule: just start with the postcondition, replace each i by (j+42) and viola! the precondition appears. The
“borogove” example shows how mechanical the process is—replace each \( i \) by \((j+42)\), and don’t worry about what else appears in the assertion.

These examples assigned the value \( j+42 \) to the variable \( i \), but, of course, the same pattern applies to any other value that is assigned to any other variable. Here is the pattern for verification formulas where the program fragment consists of a single assignment statement:

<table>
<thead>
<tr>
<th>Assignment — Simple Verification Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppose:</td>
</tr>
<tr>
<td>• ( x ) is a variable</td>
</tr>
<tr>
<td>• (&lt;\text{expr}&gt;) is an expression with the same type as ( x )</td>
</tr>
<tr>
<td>• ( Q ) is an assertion</td>
</tr>
<tr>
<td>• ( P ) is an assertion which is the same as ( Q ) but with every occurrence of ( x ) replaced by (&lt;\text{expr}&gt;)</td>
</tr>
<tr>
<td>Then this is a correct verification formula:</td>
</tr>
<tr>
<td>/* ( P ) <em>/ ( x = &lt;\text{expr}&gt;; ) /</em> ( Q ) */</td>
</tr>
</tbody>
</table>

**Examples:**

\[
\begin{align*}
\text{\/* (j + 42) < 0 */} & \quad i = j + 42; \quad \text{/* i < 0 */} \\
\text{\/* (j + 42) == 9 */} & \quad i = j + 42; \quad \text{/* i == 9 */} \\
\text{\/* (i + 1) == 9 */} & \quad i = i + 1; \quad \text{/* i == 9 */}
\end{align*}
\]

There is one snag with the simple assignment rule: The formulas must be in a rigid format—in fact, it’s too rigid. For example, the simple rule proves that this is correct:

\[
\text{\/* (j-1) > 0 */} \quad i = j-1; \quad \text{/* i > 0 */}
\]

But \text{\/* (j-1) > 0 */} is the only precondition that can fill in the blank:

\[
\text{\/* _______ */} \quad i = j-1; \quad \text{/* i > 0 */}
\]

The rule doesn’t even allow us to prove that this is correct because the precondition is missing a couple of parentheses:

\[
\text{\/* j-1 > 0 */} \quad i = j-1; \quad \text{/* i > 0 */}
\]

That’s because the rule is too mechanical, requiring us to replace \( i \) by \((j-1)\), with parentheses! We can’t use the simpler \( j-1 \), without parentheses. Sometimes these extra parentheses can be important, but in this case, \( j-1 > 0 \) has the same meaning as \( (j-1) > 0 \), so we’d like the rule to allow us to prove the version with the simpler precondition.

Can you come up with some other verification formulas which are correct, but for which our rule is too mechanical? Here are two examples:
/* j > 1 */ i = j-1; /* i > 0 */
/* j == 10 */ i = j-1; /* i > 0 */

Neither of these fits the exact form of the simple assignment rule, because neither of the preconditions is identical to the postcondition /* i > 0 */ with i replaced by (j-1). Apparently, we need a better rule for assignment statements, and the path to that is through assertion implication.

Implication and a Better Assignment Verification Rule

Sometimes when one assertion is true we can conclude that another assertion is also true. Here are a few examples:

- Whenever j == 10, then also (j - 1) > 0
- Whenever j > 1, then also (j - 1) > 0
- Whenever j == i-1, then also j < i
- Whenever (i < j) && (j < k), then also i < k

In each of these cases, if the left side is true, then we can conclude that the right side is also true. For example, in the first line we reason like this: if j == 10, then j-1 is 9 and so we conclude that (j-1) > 0. In each of these cases, we say that the first assertion implies the second assertion, or that there is an implication from the first assertion to the second. Propositional Calculus is a branch of mathematics for determining when one assertion implies another, and it is important for verification logic—although in these note an intuitive description of implication is sufficient:

<table>
<thead>
<tr>
<th>Intuitive Description of Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppose:</td>
</tr>
<tr>
<td>• P and Q are assertions</td>
</tr>
<tr>
<td>&quot;P implies Q&quot; means that whenever P is true, then so is Q.</td>
</tr>
<tr>
<td>&quot;P does not imply Q&quot; means that sometimes P is true, without</td>
</tr>
<tr>
<td>Q also being true.</td>
</tr>
</tbody>
</table>

Examples:
- j > 1 implies (j - 1) > 0
- j == i-1 implies j < i
- j > i does not imply j > i+1

We can use implication to make a better verification rule for the assignment statement. For example, the simple assignment statement rule says this is correct:

/* (j - 1) > 0 */ i = j - 1; /* i > 0 */
In the precondition, we need \((j - 1) > 0\), or at least we need something that implies \((j - 1) > 0\), so that prior to the assignment statement, \((j - 1) > 0\) will be true. In this case, a valid precondition is any assertion that implies \((j - 1) > 0\). An example: \(j = 10\) certainly implies \((j - 1) > 0\), so this is a correct formula:

\[
/* j = 10 */ \quad i = j - 1; \quad /* i > 0 */
\]

Which of these two is also correct?

\[
/* j > 1 */ \quad i = j - 1; \quad /* i > 0 */ -Is this correct? \\
/* j > 0 */ \quad i = j - 1; \quad /* i > 0 */ -How about this?
\]

The first formula is correct, since \(j > 1\) implies \((j - 1) > 0\). But the second formula is wrong, since \(j > 0\) does not imply \((j - 1) > 0\). In particular, if \(j\) happens to be 1 then \(j > 0\) is true, but \((j - 1) > 0\) is not true, so there is no implication. Here is the better rule for correctness of formulas with an assignment statement. The better rule uses implication in the way we have discussed. Check each of the examples at the end of the box to make sure you understand how to apply the new rule.

### Assignment — Verification Rule

**Suppose:**

- \(x\) is a variable
- \(<\text{expr}>\) is an expression with the same type as \(x\)
- \(P\), \(Q\), and \(R\) are assertions
- \(R\) is the same as \(Q\) but with every occurrence of \(x\) replaced by \(<\text{expr}>\)
- \(P \implies R\)

**Then this is a correct verification formula:**

\[
/* P */ \quad x = <\text{expr}>; \quad /* Q */
\]

**Examples:**

\[
/* j < -42 */ \quad i = j + 42; \quad /* i < 0 */
\]
\[
/* j = -33 */ \quad i = j + 42; \quad /* i = 9 */
\]
\[
/* i = 8 */ \quad i = i + 1; \quad /* i = 9 */
\]
Proof Outlines for Sequences of Assignments

With the assignment statement rule in hand, we can tackle program fragments that contain several assignments. For example, here's a correct formula that involves swapping the values of two variables:

```c
/* i < j */
temp = i;
i = j;
/* j < i */
j = temp;
```

How would you go about convincing someone that this is correct? The method we'll use is called a proof outline, which consists of inserting new assertions inside the program fragment—usually one assertion between each pair of statements, but sometimes more. The new assertions must meet certain requirements which depend on the kind of statements in the program, i.e., assignment statements have one requirement, if-then-else statements have another requirement, while-loops have another requirement, and so on. Finally, if the requirements are met, then we may conclude that the verification formula is correct.

Before we look at the general case, let's see how it works for the swapping example:

```c
/* i < j */
temp = i;
/* temp < j */
i = j;
/* temp < i */
j = temp;
/* j < i */
```

We have inserted two new assertions, indicating what we know about the program at intermediate points between the precondition and postcondition. With these intermediate assertions in place, we can reason about each of the three assignment statements individually. For example, we can focus on the first assignment statement and the two assertions that surround it:

```c
/* i < j */
temp = i;
/* temp < j */
```

This one portion is correct, since if we start with \( i < j \) and assign \( \text{temp} = j \), then afterwards \( \text{temp} \) is less than \( j \). The correctness of this portion follows from the Assignment Verification Rule—take the postcondition of this portion, \( \text{temp} < j \), replace \( \text{temp} \) by \((1)\), and we have the precondition \((i) < j\). We can omit the parentheses around \( i \) since the simpler assertion \( i < j \) is equivalent to the more
cluttered version (i) < j. The proof outline has two other assignment statements
which we can also examine with their surrounding assertions:

    /* temp < j */
    i = j;
    /* temp < i */

and

    /* temp < i */
    j = temp
    /* j < i */

You can check that each of these also meets the Assignment Verification Rule.
It's the correctness of the three individual portions that makes the whole formula
correct. The reasoning goes like this:

    /* i < j */  — The initial precondition is true here ...
    temp = i;
    /* temp < j */  — ... therefore this will be true here ...
    i = j;
    /* temp < i */  — ... and this will be true here ...
    j = temp
    /* j < i */  — ... so the postcondition is true here.

Why does this work? Well, for the entire sequence to be correct, we require that
whenever it starts with i < j, then it finishes with j < i. And if we start with
i < j, then after the first assignment we know temp < j ... after one more assign-
ment we know temp < i ... and after the third assignment we know j < i ... which
is exactly what's needed. The individual portions follow the Assignment Verifi-
cation Rule, and these portions link together to guarantee the correctness of the
whole sequence. A proof outline is a method to show that a verification formula
is correct. It consists of inserting new assertions within the program fragment,
following specific requirements for each of the different kinds of C++ state-
ments. If the requirements are followed for each statement in the program frag-
ment, then the verification formula is correct.

The key to a proof outline lies in determining what the requirements are for
each of the different kinds of C++ statements. For assignment statements the new
assertions must occur so that each individual assignment statement, together with
its surrounding assertions, meets the Assignment Verification Rule. This require-
ment for assignment statements is given here, and it is the first of several proof outline requirements:

<table>
<thead>
<tr>
<th>Proof Outlines — Requirement for Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Each assignment statement in a proof outline must be surrounded by two assertions, like this:</td>
</tr>
<tr>
<td>/* P <em>/ x = &lt;expr&gt;; /</em> Q */</td>
</tr>
<tr>
<td>Requirement:</td>
</tr>
<tr>
<td>For each of these occurrences, the Assignment Verification Rule must be met. In other words:</td>
</tr>
<tr>
<td>1. Start with the Q;</td>
</tr>
<tr>
<td>2. Create a new assertion from Q by replacing every occurrence of x by (&lt;expr&gt;);</td>
</tr>
<tr>
<td>3. Then P must imply this new assertion.</td>
</tr>
</tbody>
</table>

There’s one other technique in a proof outline which is convenient at this point: placing two assertions next to each other in the outline. When we reach the first assertion, we know that it is true, and we would like the second assertion to also be true, so that the “links of the proof” may continue. This means that the first assertion must imply the second assertion. Here’s an example:

```c
/* i == 10 */
temp = i;
/* (i == 10) && (temp == 10) */ — Two adjacent assertions
/* temp-i == 0 */
j = temp-i;
/* j == 0 */
```

In the center of this proof outline, there are two adjacent assertions. The first assertion ((i==10) && (temp==10)) implies the second one (temp-i = 0). So that if we reach the first assertion, and it is true, then the second assertion is also true, and the proof outline may continue. The primary reason for placing two adjacent assertions is clarity. The first assertion is a fairly natural choice for the postcondition of temp = i; the second assertion is a natural choice for the precondition of j = temp-i. Although we could write the proof outline with a single assertion between the assignments, writing both assertions makes it clear that there are three requirements to check:

1. That /* i == 10 */ temp = i; /* (i == 10) && (temp == 10) */ is correct;
2. That (i == 10) && (temp == 10) implies temp-i == 0;
3. That /* temp-i == 0 */ j = temp-i; /* j == 0 */ is correct.
The general requirement for two adjacent assertions is given in here:

```plaintext
Proof Outlines — Requirement for Adjacent Assertions
A proof outline may contain adjacent assertions, like this:
/* P */
/* Q */

Requirement:
Whenever there are two adjacent assertions, the first assertion must imply the second assertion. In other words: whenever P is true, the assertion Q must also be true.
```

Once you have a proof outline, it’s fairly easy to check that it meets the requirements are met for assignments and adjacent assertions. But how do you create a proof outline in the first place? Typically, you will write a code fragment with pre- and postconditions in mind, and you are left with the task of filling in the intermediate assertions to complete the proof outline. For example, suppose we have written a code fragment to compute the number of quarters, dimes and pennies needed to provide a certain amount of change. The code fragment, with it’s pre- and postconditions looks like this:

```c
/* change == n */
quarters = change / 25;
change = change % 25;
dimes = change / 10;
change = change % 10;
pennies = change;
/* quarters*25 + dimes*10 + pennies == n */
```

In this code fragment, n is the original value of change, and at the end of the code fragment quarters, dimes and pennies tell us how many quarters, dimes and pennies make n cents. With a sequence like this, the proof outline can be created by starting at the bottom and working upward. In this example, we start by filling in this blank:

```c
/* _____________________________________ */
pennies = change;
/* quarters*25 + dimes*10 + pennies == n */
```
To fill in the blank, start with the postcondition, and replace each occurrence of pennies with change. This is called “pulling the assertion back through the assignment statement,” and we end up with this:

```plaintext
/* quarters*25 + dimes*10 + change == n */
pennies = change;
/* quarters*25 + dimes*10 + pennies == n */
```

We continue working upwards, filling in the blank before the next assignment statement:

```plaintext
/* change = change % 10;
/* quarters*25 + dimes*10 + change == n */
pennies = change;
/* quarters*25 + dimes*10 + pennies == n */
```

How do we fill in the blank?

Pull the assertion `quarters*25 + dimes*10 + change = n` back through the assignment statement `change = change % 10`, replacing each occurrence of change with the expression `change % 10`:

```plaintext
/* quarters*25 + dimes*10 + change % 10 == n */
change = change % 10;
/* quarters*25 + dimes*10 + change == n */
pennies = change;
/* quarters*25 + dimes*10 + pennies == n */
```

Keep going! Pull the topmost assertion back through an assignment statement to get this:

```plaintext
/* quarters*25 + (change / 10)*10 + (change % 10) == n */
dimes = change / 10;
/* quarters*25 + dimes*10 + change % 10 == n */
change = change % 10;
/* quarters*25 + dimes*10 + change == n */
pennies = change;
/* quarters*25 + dimes*10 + pennies == n */
```
And now pause to catch your breath. When you keep pulling assertions back, the assertions sometimes get more and more complex. But sometimes you'll notice a way to simplify an assertion. For example, if you think about how / and % work together, you'll realize that:

\[
\text{change} = (\text{change} / 10) \times 10 + (\text{change} \mod 10)
\]

This equality allows us to simplify our latest assertion, and we'll just place the simpler assertion above the more complicated version in the proof outline:

```c
/* quarters*25 + change = n */
/* quarters*25 + (change / 10)*10 + (change \mod 10) = n */
dimes = change / 10;
/* quarters*25 + dimes*10 + change \mod 10 = n */
change = change \mod 10;
/* quarters*25 + dimes*10 + change = n */
pennies = change;
/* quarters*25 + dimes*10 + pennies = n */
```

The outline now has two adjacent assertions, and the first one implies the second one (since the first assertion is a simpler equivalent version of the second one). There's two more assignment statements to pull the assertion through. When you're done, you'll have a complete proof outline. The complete proof outline for this example is in Figure 2, along with a summary of the technique of pulling an assertion back through an assignment statement to create a proof outline.
FIGURE 2  Creating a Proof Outline For a Sequence of Assignment Statements

To create a proof outline for a sequence of assignment statements:

1. Start with the postcondition at the bottom of the sequence.
2. Pull it back through the assignment statement above it.
3. If appropriate, then simplify the resulting assertion.
4. Repeat steps 2 and 3 for each new assertion...
   ... until you reach the top of the code fragment.

Example:

```
/* change = n */
/* (change / 25)*25 + change % 25 = n */
quarters = change / 25;
/* quarters*25 + change % 25 = n */
change = change % 25;
/* quarters*25 + change = n */
/* quarters*25 + (change / 10)*10 + change % 10 = n */
dimes = change / 10;
/* quarters*25 + dimes*10 + change % 10 = n */
change = change % 10;
/* quarters*25 + dimes*10 + change = n */
pennies = change;
/* quarters*25 + dimes*10 + pennies = n */
```
4. VERIFICATION OF CONDITIONAL STATEMENTS

In this section, we'll examine verification formulas and proof outlines for program fragments which contain two kinds of conditional statements:

```c
if (<expr>)
{
    .
    .
    .
}
else
{
    .
    .
    .
}
```

The ... parts of the program fragment may be any sequence of C++ statements.

**Proof Outlines for if-else**

To start with, here's an example of a verification formula with an if-else-statement.

```c
/* i + j == N */
if (i < j)
{
    minimum = i;
    maximum = j;
}
else
{
    minimum = j;
    maximum = i;
}
/* (minimum + maximum == N) && (minimum <= maximum) */
```

Actually, we could say a bit more about minimum and maximum in the postcondition, but this simple postcondition will do for now. A proof outline for this formula must meet the two requirements that we've already seen: (1) the Requirement for Assignments, and (2) the Requirement for Adjacent Assertions. There will also be a requirement for the if-else-statement, and these three requirements taken together guarantee the correctness of the verification formula.
Before we come up with the if-else requirement, it will be useful to see the entire proof outline for this formula:

```c
/* i + j == N */
if (i < j)
{
  /* (i + j == N) && (i < j) */
  /* (i + j == N) && (i <= j) */
  minimum = i;
  /* (i + maximum <= N) && (i <= maximum) */
  maximum = j;
  /* (minimum + maximum == N) && (minimum <= maximum) */
}
else
{
  /* (j + i == N) && (j <= i) */
  minimum = j;
  /* (minimum + i == N) && (minimum <= i) */
  maximum = i;
  /* (minimum + maximum == N) && (minimum <= maximum) */
} /* (minimum + maximum == N) && (minimum <= maximum) */
```

You can check that this proof outline satisfies the Requirement for Assignments the Requirement for Adjacent Assertions. For example, the outline contains these two adjacent assertions:

```c
/* (i + j == N) && (i < j) */
/* (i + j == N) && (i <= j) */
```

Does the first assertion imply the second one? (Yes — because whenever \( i \) is less than \( j \), it's also true that \( i \) is less than or equal to \( j \).)

As for the if-else-statement, there are three points which are critical in the proof outline:

**First point.** The assertion at the top of the if-branch, \((i + j == N) && (i < j)\). How do we know this assertion is true when we reach this point? Well, \((i + j == N)\) is true because that is the precondition of the entire if-else-statement. And why is the other part—\((i < j)\)—true? Because the program is in the if-branch, and in order to enter the if-branch the test “if \((i < j)\)” must be passed. In general, here’s the requirement for the first assertion of the if-branch:

```c
/* p */
if (<expr>)
{
  /* First assertion of the if-branch is implied by ((p) && (<expr>)) */
  ...
```
Second Point: The assertion at the top of the else-branch, \((i + j == N) \&\& (j <= i)\). How do we know this assertion is true when we reach this point? Just like the if-branch, \((i + j == N)\) is true because that is the precondition of the entire if-else statement. And why is the other part—\((j <= i)\)—true? Because the program is in the else-branch, and in order to enter the else-branch the test “if \((i < j)\)” must be failed, and this means \((j <= i)\). In general, here's the requirement for the first assertion of the else-branch:

```plaintext
/* P */
if (<expr>)
{
    ...
}
else
{
    /* First assertion of the else-branch is implied by ((P) && !(<expr>)) */
    ...
}
```

Third point: Here's the assertion which occurs just after the if-else statement,

\[(minimum + maximum == N) \&\& (minimum <= maximum)\]

How do we know this assertion is true when we reach this point? Well, there are two pathways through the if-else statement. One pathway comes through the if-branch, and in this case the last assertion in the if-branch is:

\[(minimum + maximum == N) \&\& (minimum <= maximum)\]

So, if this is true at the end of the if-branch, then it will be true just after the if-else statement. The other pathway comes through the else-branch, and again when the else-branch finishes, we know that the assertion we need is true. In general, here's the requirement for the first assertion after the if-else statement finishes:

```plaintext
if (<expr>)
{
    ...
    /* The last assertion of the if-branch must imply Q */
}
else
{
    ...
    /* The last assertion of the else-branch must also imply Q */
}
/* Q —the assertion after the if-else statement */
```
The three requirements are summarized in the box shown below. The box shows if- and else-branches that contain a compound statement surrounded by brackets. We could also write a rule that worked without compound statements, but having the brackets makes it easy to identify exactly where the first and last assertions of the if- and else-branches occur.

<table>
<thead>
<tr>
<th>Proof Outlines — Requirement for If-Else</th>
</tr>
</thead>
<tbody>
<tr>
<td>A proof outline may contain an if-else statement, like this:</td>
</tr>
<tr>
<td>/* P */</td>
</tr>
<tr>
<td>if (&lt;expr&gt;)</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>/* P1 */</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>/* Q1 */</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>/* P2 */</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>/* Q2 */</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>/* Q */</td>
</tr>
</tbody>
</table>

**Requirement:**
For each such if-else statement, the following must be valid:

1. \((P \&\& (<expr>))\) implies the first assertion \(P_1\) of the if-branch.
2. \((P \&\& !(<expr>))\) implies the first assertion \(P_2\) of the else-branch.
3. The last assertion, \(Q_1\), of the if-branch implies \(Q\).
4. The last assertion, \(Q_2\), of the else-branch implies \(Q\).

**How to Create a Proof Outline for if-else Statements**

Let's look at a small example to see how a proof outline is actually created for an if-else-statement. In the next example the if-branch is just a single statement, and the else-branch is a compound statement. This example also uses a double variable \(x\), a double constant \(R\), and a bool variable \(error\), rather than integers.
Actually, the example starts with a situation where we’re not sure of the best pre-
condition, because that’s often the case in actual verifications:

```c
// Code fragment to change x to square root of its initial value.
// If this is successful, then error is set to false.
// If this fails because x < 0, then error is set to true.
/* _______ ??? ________ */
if (x < 0)
{
  error = true;
}
else
{
  x = sqrt(x);
  error = false;
}
/* (error && (R < 0)) || ((!error) && (x == sqrt(R))) */
```

The postcondition that we want to establish includes a constant R, which may be
any double number.

We’ll go through a sequence of four steps to create the proof outline. This isn’t
the only method to create the outline, but it’s convenient to have a pattern to flow.

**Step 1:** Place the postcondition at the end of each of the branches, as shown here:

```c
// Code fragment to change x to square root of its initial value.
// If this is successful, then error is set to false.
// If this fails because x < 0, then error is set to true.
/* _______ ??? ________ */
if (x < 0)
{
  error = true;
  /* (error && (R < 0)) || ((!error) && (x == sqrt(R))) */
}
else
{
  x = sqrt(x);
  error = false;
  /* (error && (R < 0)) || ((!error) && (x == sqrt(R))) */
}
/* (error && (R < 0)) || ((!error) && (x == sqrt(R))) */
```

**Step 2:** If possible, simplify the assertions at the end of each branch. The requirement is that each of the simpler assertions must imply the assertion which follows the if-else statement, and each of the simpler assertions must be true at the point where it occurs. In this case, we can make simplifications based on the
known value of the boolean variable error. In particular, at the end of the if-
branch we know that error is true, and at the end of the else-branch, we know
that error is false, so we have these simpler assertions:

// Code fragment to change x to square root of its initial value.
// If this is successful, then error is set to false.
// If this fails because x < 0, then error is set to true.
/* _______ ??? _______ */
if (x < 0)
{
    error = true;
    /* error && (R < 0) */
}
else
{
    x = sqrt(x);
    error = false;
    /* (!error) && (x == sqrt(R)) */
}
/* (error && (R < 0)) || (!error) && (x == sqrt(R)) */

Step 3: Create a proof outline for the if-branch and a second proof outline for the
else-branch. The result of these outlines will be an assertion at the top of each
branch, as shown here:

// Code fragment to change x to square root of its initial value.
// If this is successful, then error is set to false.
// If this fails because x < 0, then error is set to true.
/* _______ ??? _______ */
if (x < 0)
{
    /* R < 0 */
    /* true && (R < 0) */
    error = true;
    /* error && (R < 0) */
}
else
{
    /* sqrt(x) == sqrt(R) */
    x = sqrt(x);
    /* x = sqrt(R) */
    /* (!false) && (x == sqrt(R)) */
    error = false;
    /* (!error) && (x == sqrt(R)) */
}
/* (error && (R < 0)) || (!error) && (x == sqrt(R)) */
Step 4: Propose a precondition for the if-else-statement. This condition needs to be strong enough to guarantee that the assertions at the tops of each branch will be true. If the precondition is $P$ and the if-test is $<\text{expr}>$, then here are the two required implications:

$$(P \land (x < 0)) \implies \text{the assertion at top of if-branch};$$
$$(P \land (x >= 0)) \implies \text{the assertion at top of else-branch}.$$ 

In our example case, we propose the precondition $x==R$, which will work since

$$(x == R) && (x < 0) \implies R < 0$$
$$(x == R) && (x >= 0) \implies \sqrt{x} == \sqrt{R}$$

Do you see why both of these implications are true? For example, the second implication is true since whenever $x==R$ and $x$ is not negative, then $\sqrt{x}$ exists and must be equal to $\sqrt{R}$. Since these implications are valid, we now have a complete proof outline:

```c
// Code fragment to change x to square root of its initial value.
// If this is successful, then error is set to false.
// If this fails because x < 0, then error is set to true.
/* x == R*/
if (x < 0)
{
    /* R < 0 */
    /* true && (R < 0) */
    error = true;
    /* error && (R < 0) */
}
else
{
    /* \sqrt{x} == \sqrt{R} */
    x = \sqrt{x};
    /* x == \sqrt{R} */
    /* (!false) && (x == \sqrt{R}) */
    error = false;
    /* (!error) && (x == \sqrt{R}) */
}
/* (error && (R < 0)) || ((!error) && (x == \sqrt{R})) */
```

In creating the proof outline, Step 4 required some analysis to obtain a reasonable precondition. In general, it helps to know a bit about the program: What did the programmer intend to be true before the if-else-statement, in order to make it work correctly? The answer to this question is generally the correct precondition to place before the if-else-statement.
And what if the programmer's intended precondition doesn't result in a correct proof outline? Then there is a mistake in the programmer's precondition, or in the program—or both! Start by trying to strengthen the precondition: is there something more that is known about the program at that point? If not, is the program incorrect?

**Example: Nested if-else**

The next example illustrates two new things: the treatment of nested if-else-statements, and the use of an assertion written in English rather than C++. The program fragment finds the minimum value of three numbers:

```c++
/* true */
if (i <= j)
{
    if (i <= k)
    {
        small = i;
    }
    else
    {
        small = k;
    }
}
else
{
    if (j <= k)
    {
        small = j;
    }
    else
    {
        small = k;
    }
}
/* small is minimum of i, j, k */
```

This program has three if-else-statements: the outermost statement with the test i<=j and the two nested statements with tests i<=k and j<=k. In the proof outline, each of the three statements must meet the requirements for an if-else-statement, and we'll see how this is done in a moment.

The other peculiarity is that the postcondition is written in English rather than C++. Of course, in this case, we could write the equivalent postcondition in C++. It could start something like this:

```
((small == i) or (small == j) or ...
```
But the expression would be rather lengthy, and besides, what we really intend to say is that $small$ is the minimum of $i$, $j$, and $k$—so we may as well state it directly. In general, use an English statement whenever the C++ statement becomes unwieldy, and the English statement can be given clearly.

To create a proof outline, we’ll use the same four steps that we saw before:

**Step 1:** Place the postcondition at the end of each of the branches of the large if-else statement:

```c++
/* true */
if (i <= j)
{
    if (i <= k)
    {
        small = i;
    }
    else
    {
        small = k;
    }
    /* small is minimum of i, j, k */
}
else
{
    if (j <= k)
    {
        small = j;
    }
    else
    {
        small = k;
    }
    /* small is minimum of i, j, k */
}
/* small is minimum of i, j, k */
```

**Step 2:** If possible, simplify the assertions at the end of each branch. The assertions are already pretty simple, so we’ll skip this step.
**Step 3:** Create a proof outline for the if- and else-branches. The result of these outlines will be an assertion at the top of each branch. In this case, each of the branches is a nested if-else-statement, so the proof outline for each branch needs to follow the requirements for an if-else-statement. Here are some possible proof outlines for these two branches:

**For the if-branch:**

```plaintext
if (i <= j)
{
    /* (i <= j) */
    if (i <= k)
    {
        /* (i <= j) && (i <= k) */
        /* i is minimum of i, j, k */
        small = i;
        /* small is minimum of i, j, k */
    }
    else
    {
        /* (i <= j) && (k < i) */
        /* k is minimum of i, j, k */
        small = k;
        /* small is minimum of i, j, k */
    }
    /* small is minimum of i, j, k */
}
```

**For the else-branch:**

```plaintext
else
{
    /* (j < i) */
    if (j <= k)
    {
        /* (j < i) && (j <= k) */
        /* j is minimum of i, j, k */
        small = j;
        /* small is minimum of i, j, k */
    }
    else
    {
        /* (j < i) && (k < j) */
        /* k is minimum of i, j, k */
        small = k;
        /* small is minimum of i, j, k */
    }
    /* small is minimum of i, j, k */
}
```
Can you check that both of these proof outlines meet all the requirements for an if-else-statement? There are four requirements to check for each of them—the requirements at the box on page 20.

If you created these outlines yourself, you may have got stuck at one point: what assertion to place at the top of each of these fragments. When you’re at that point, go back to the original larger program, and ask yourself what is known about the program at that point. For example, at the top of the if-branch we know \( i \leq j \) and at the top of the else-branch we know \( j < i \); those were appropriate choices for assertions at the top of these two fragments.
**Step 4:** We already have a precondition for the entire if-else-statement, so we can put together all the pieces for a complete proof outline:

```c
/* true */
if (i <= j)
{
    /* (i <= j) */
    if (i <= k)
    {
        /* (i <= j) && (i <= k) */
        /* i is minimum of i, j, k */
        small = i;
        /* small is minimum of i, j, k */
    }
    else
    {
        /* (i <= j) && (k < i) */
        /* k is minimum of i, j, k */
        small = k;
        /* small is minimum of i, j, k */
    }
    /* small is minimum of i, j, k */
}
else
{
    /* (j < i) */
    if (j <= k)
    {
        /* (j < i) && (j <= k) */
        /* j is minimum of i, j, k */
        small = j;
        /* small is minimum of i, j, k */
    }
    else
    {
        /* (j < i) && (k < j) */
        /* k is minimum of i, j, k */
        small = k;
        /* small is minimum of i, j, k */
    }
    /* small is minimum of i, j, k */
}
/* small is minimum of i, j, k */
To complete Step 4, we need to check that the outline meets the requirements for
the precondition of an if-else-statement:

\[
true \land (i \leq j) \implies i \leq j \\
true \land \neg (i \leq j) \implies j < i
\]

Since these are both valid, the proof outline is finished.

**Proof Outlines for if-then**

When an if-statement does not have an else-branch, then there’s an extra require-
ment on the precondition. Here’s a small illustration that uses the expression
\((x \% 2 == 1)\) to determine whether an integer \(x\) is odd. (which returns true when
its argument is an odd integer):

```c
/* x > 0 */
if (x%2 == 1)
{
    /* (x > 0) && x is odd */
    /* (x+1 > 0) && (x+1) is even */
    x = x + 1;
    /* (x > 0) && x is even */
}
/* (x > 0) && x is even */
```

The conditions on the if-branch are the same as if it were an if-else statement:

\((x>0) \land (x\%2 == 1)\) *imply the first assertion in the if-branch*

The last assertion in the if-branch implies the postcondition.

If there was an else-branch, then there would also be two requirements to guar-
antee that whenever the else-branch is executed, then the postcondition is valid.
When there is no else-branch, we still need to consider the case where the test
\((x\%2 == 1)\) fails, and make sure that the postcondition will be valid. This is
accomplished with an extra requirement for a proof outline that contains an if-
statement without an else-branch: If there is no else-branch, then the precondition
together with the negation of the if-test must imply the postcondition.

In this example, the requirement means that:

\((x > 0) \land \neg (x\%2 == 1)\) *implies (x > 0) \land x is even*

The box on the top of the next page shows the complete requirement for an if-
statement with no else branch.
Proof Outlines — Requirement for if (without else)

A proof outline may contain an if statement with no else-branch, like this:

```c
/* p */
if (<expr>)
{
    /* p1 */
    ...
    /* q1 */
}
/* q */
```

Requirement:
For each such if-statement, the following must be valid:

1. \((P \&\& (<expr>))\) implies the first assertion \(P1\) of the if-branch.
2. The last assertion, \(Q1\), of the if-branch implies \(Q\).
3. \((P \&\& !(<expr>))\) implies \(Q\).
5. VERIFICATION OF LOOPS

Loops provide the steepest challenge for programming logic. But they also provide the greatest reward, because a good understanding of what makes a loop correct also provides a pathway to designing programs with loops. This section will cover both the format for proof outlines of loops, and show a loop design technique based on the outlines.

Reasoning with Loop Invariants

It will be useful if you first think about how you reason about loops in a small example which deals with factorials. The factorial of a natural number \( n \) is written \( n! \) and is equal to the product of all the integers between 1 and \( @n\%: \)

\[
\begin{align*}
1! &= 1 \\
2! &= 1 \times 2 = 2 \\
3! &= 1 \times 2 \times 3 = 6 \\
4! &= 1 \times 2 \times 3 \times 4 = 24 \\
n! &= 1 \times 2 \times \ldots \times n
\end{align*}
\]

The value of 0! is also defined as 1, so that \( n! \) is defined for any non-negative integer. Factorials are important in combinatorial mathematics, and occasionally in practical problems too. For example, suppose you have five desserts and five hungry friends. Then there are 5! different ways to distribute the desserts with to each friend (and whichever of these 120 combinations you decide upon, four friends will be unhappy). Anyway, here's a C++ fragment to compute \( n! \) with a while loop. The assertions in this program are not all C++ expressions—but that's okay because assertions are just comments for our use.

```cpp
/* (0 <= n) */
i = 0;
answer = 1;
while (i < n)
{
    i = i + 1;
    answer = answer * i;
}
/* (answer == n!) */
```

With a small example, you can often just read the program and believe that the postcondition is valid. But the more important problem we want to flush out with this example deals with your reasoning, namely *How do you know that the postcondition is valid?* and *Is there a method to your reasoning that will provide a proof outline technique for while loops?* In this example, your reasoning might look like this:
Before the loop starts, \( i \) is equal to 0 and answer is equal to 0!. Each time through the loop, \( i \) is increased by 1, and answer is kept at the value of \( i! \). Also, at all times \( i \leq n \), and the loop must end with \( (i < n) \) being false—which means the loop ends with \( (i == n) \). So at the end of the loop, \( (\text{answer} == i!) \) and \( (i! == n!) \), hence \( (\text{answer} == n!) \).

The foundation to the argument is the relationship which is maintained between answer, \( i \) and \( n \). Each time the body of the loop begins (or ends) the assertion

\[
(\text{answer} == i!) \&\& (i <= n)
\]

remains valid. Within the fragment, this assertion is valid in four places: (1) Before the loop is ever entered; (2) At the top of the body of the loop; (3) At the bottom of the body of the loop; (4) After the loop is exited.

This assertion an example of a loop invariant, which is an assertion that is true at the beginning and end of each iteration of a loop. In other words, the first time the loop is entered, its loop invariant is true. When the body of the loop is executing, the loop invariant may momentarily become false, but by the end of the loop’s body the invariant is true once more. If there is another iteration, then the same process occurs: invariant true when the body begins, invariant true when the body ends. Using loop invariants, you can reason about the correctness of a loop in these four steps:

**Step 1:** Identify the loop invariant. This will be the hardest step, because the invariant must must be strong enough to satisfy the next three steps. We’ll talk more about this issue later, but for our example the invariant is

\[
(\text{answer} == i!) \&\& (i <= n).
\]

**Step 2:** Check that the invariant is true when the loop is first reached. This can be done with a proof outline:

```c
/* (0 <= n) */
i = 0;
/* (0 == i) \&\& (i <= n) */
/* (1 == i!) \&\& (i <= n) */
answer = 1;
/* (answer == i!) \&\& (i <= n) */
...the loop starts here...
```

**Step 3:** Check that the body of the loop maintains the loop invariant. In other words, if the invariant is true when the body begins, then it will also be true when the body ends. This can also be done with a proof outline, where the program fragment is the loop’s body. The precondition for this proof outline can actually be a bit stronger than just the invariant. Do you see why? What else do you know
about the state of the program when the body of the loop begins? Answer: You know that the while-loop’s test had to be passed. In this example, you know that 
(i < n) must be true—otherwise the while-loop would end. So, at the top of this 
proof outline we put the invariant and (i < n). At the bottom of the proof outline 
we put the invariant alone, since that’s what we need to guarantee is true at that 
point, as shown here:

...just before the loop we have the assertion:  
/* (answer == i!) && (i <= n) */
while (i < n)
{
  ...at the top of the loop we start with the invariant and (i < n)...
  /* (answer == i!) && (i <= n) && (i < n)*/
  /* (answer == i!) && (i < n)*/
  /* (answer*i == (i+1)! && (i+1 <= n) */
  i = i + 1;
  /* (answer*i == i!) && (i <= n) */
  answer = answer * i;
  /* (answer == i!) && (i <= n) */
}

At the beginning of this proof outline, there are three consecutive assertions. The 
reason for this is to try to make the implications clear. Can you see that the first 
assertion implies the second one, and the second assertion implies the third one?

**Step 4:** Check the validity of the postcondition of the loop. After the loop ends, 
we know that the invariant is still true. We also know one more fact: that the 
loop’s test is now false. In this case, we know that:

/* (answer == i!) && (i <= n) && not (i < n)*/  

\text{The invariant} \quad \text{the negation of} \quad \text{the loop’s test}

This long assertion, which must be true when the loop ends, implies (i == n), 
and therefore (answer == n!), which is the postcondition that we were after.
These four steps fit together into a proof outline. For clarity, we also write "Loop Invariant" in front of the invariant the first time it appears, as shown here:

```c
/* 0 <= n */
i = 0;
/* (0 == i) && (i <= n) */
/* (1 == i!) && (i <= n) */
answer = 1;
/* Loop Invariant: (answer == i!) && (i <= n) */
while (i < n) do
{
    /* (answer == i!) && (i <= n) && (i < n)*/
    /* (answer == i!) && (i < n)*/
    /* (answer*i == (i+1)! && (i+1 <= n) */
    i = i + 1;
    /* (answer*i == i!) && (i <= n) */
    answer = answer * i;
    /* (answer == i!) && (i <= n) */
};
/* (answer == i!) && (i <= n) && not (i < n) */
/* (answer == i!) && (i = n) */
/* answer = n! */
```

Proof Outlines for while-loops

Proof outlines for a while-loop will always include a loop invariant which is treated like the invariant in the factorial example. The exact requirements appear here:

<table>
<thead>
<tr>
<th>Proof Outlines — Requirement for While-Loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>A proof outline may contain a while-loop. The while-loop must have an associated loop invariant. The loop invariant, which is called P in this example, must appear in the four places indicated below:</td>
</tr>
</tbody>
</table>

```c
/* Loop Invariant: P */
while (<expr>)
{
    /* P && (<expr>) */
    ...body of the loop...
    /* P */
}
/* P && not(<expr>) */
```

*Requirement:*  
Note that the body of the loop must be a correct proof outline that begins with the assertion /* P && (<expr>) */ and ends with the loop invariant (P) on its own.
The requirement for the while-loop could be written in other forms—for example, all we really need is that \((P \&\& (\text{expr}))\) implies whatever occurs at the top of the loop’s body. But the more rigid format that we have written makes the role of the loop invariant clearer.

### Indentifying and Using Loop Invariants

There are no simple recipes for discovering appropriate loop invariants in all situations. But there are some common patterns which you can recognize. Once you learn to identify certain patterns in while-loops and their invariants, you may also find yourself “turning the tables,” and actually using the patterns to help design loops. We’ll look at several examples, starting with the factorial function.

The most common while-loop pattern consists of setting up and maintaining a relationship between a group of variables and a control variable. Each iteration of the while-loop changes the value of the control variable, and then restores the relationship between the control variable and the other variables. This pattern occurs in the factorial example, where the control variable is \(i\):

```c
/* n >= 0 */
i = 0;
answer = 1;
while (i < n) do
{
    i = i + 1;
    answer = answer * i;
}
/* answer == n! */
```

When you recognize this pattern, you should be able to identify the relationship which is established and continually maintained. In the factorial example, the relationship is the equality \((answer == i!)\). This relationship between \(answer\) and \(i\) can be your first guess at the loop invariant. Of course, it is not always the right loop invariant: to be correct, it must be strong enough to guarantee the post-condition that you are seeking. In this case, we need to check whether

\[(answer == i!) \&\& \neg(i < n) \implies (answer == n!)]\)

Unfortunately, this implication is invalid—can you see why? It’s because the left side only guarantees that \(i\) is \textit{greater than or equal to} \(n\), whereas we really need
to know that $i$ is actually equal to $n$. This suggests that we strengthen the loop invariant by including the fact that $i$ is never greater than $n$:

**Loop invariant:** $(\text{answer} == i!) \&\& (i <= n)$

The loop invariant is now strong enough to guarantee the postcondition, because:

$$(\text{answer} == i!) \&\& (i <= n) \&\& \text{not } (i < n) \implies (\text{answer} == n!)$$

**Invariant and negation of loop's test**

**Assertion that we want after the loop**

**Another Example**

We'll follow the same steps with the next example, which deals with loan payments. Here's the situation: You've loaned your teacher $2000 so he can buy a turbo-charged new computer. He will make monthly payments of $100 on the loan, and there is an interest rate charge of $\frac{1}{2}$ of one percent, which is added to the loan balance before each of the monthly payments. We'll write a program fragment to compute how many months before the loan is paid off, and the total of all the monthly payments. The program will use these constants to describe the loan:

```c
const double AMOUNT = 2000.00; /* Initial amount of the loan */
const double RATE = 0.005;      /* Monthly interest, 1/2% */
const double PAY = 100.00;      /* Usual monthly payment */
```

As we design the program, we'll keep in mind the pattern of a control variable together with a group of other variables. The control variable, $m$, will be the number of months that have passed since the outset of the the loan, and we will maintain a relationship between $m$ and two other double variables:

- **total_payments will be the total of all payments made in first $m$ months**
- **balance will always be the balance of the loan after $m$ months**

These two equalities will be our initial guess at the loop invariant—though we might need to strengthen it later. The idea for the loop itself is to iterate until the balance reaches zero, at which point $m$ gives the length of the loan and total_payments gives the total payments throughout the life of the loan. These variables and their connection to $m$ are good choices for two reasons: (1) We can start $m$ at zero, and easily fill in the initial values of total_payments and balance; (2) Each time that we increment $m$ by one, its easy to update the current values of total_payments and balance to maintain their relationship with $m$. 
Here is an outline of the loan program using these ideas:

```c
m = 0;
total_payments = 0;
balance = amount;
/**************************
 * Loop Invariant: (initial guess)
 * total_payments == total payments after m months.
 * balance == balance of the loan after m months.
**************************/
while (balance != 0)
{
    /* <Loop Invariant> && (balance != 0) */
    m = m + 1;
    /* put statements here to restore loop invariant */
    /* <Loop Invariant> */
}
/**************************
 * postcondition:
 * m == months needed to pay off the loan.
 * total_payments == total payments during m months.
**************************/
```

There are a couple things to notice in the outline: First, since the loop invariant is rather long, we just write it once before the loop, and then use `<Loop Invariant>` whenever we want to refer to it later. Second, most of the statements in the while-loop are omitted—that’s because we want to get the loop invariant correct before we go to the trouble of designing the loop’s body. For the invariant to be correct, we need this implication:

```c
<Loop Invariant> && not (Balance > 0) implies
<Postcondition>
```

To figure out whether this implication is valid, look at the two parts of the postcondition: (1) $m$ is the number of months needed to pay off the loan, and (2) `total_payments` is the total payments during those $m$ months. Part (2) is part of the loop invariant itself, so the loop invariant certainly implies part (2). But to imply part (1), we need to strengthen the loop invariant. The problem is that $m$ could actually be greater than the number of months to pay off the loan, and the two parts of the current loop invariant would still be true. Of course, in the actual program, $m$ is never greater than the number of months to pay off the loan, so we’ll revise the loop invariant to reflect this fact, as shown at the top of the next page.
/* Loop Invariant: (revised) */
* total_payments == Total payments after m months.
* balance == Balance of the loan after m months.
* m <= Number of months to pay off the loan.
***************************************************************************/

Now that we have this stronger loop invariant, when the loop ends, we are guaranteed of four things:
1. total_payments == Total payments after m months.
2. balance == Balance of the loan after m months.
3. m <= Number of months to pay off the loan.
4. balance == 0

These four items together are now strong enough to also imply part (1) of the precondition.

Now that we have a loop invariant which is strong enough to imply the entire postcondition, we can complete the design of the loop’s body—in other words, fill in the rest of the statements in the body of the loop:

```
while (balance != 0)
{
    /* <Loop Invariant> && (balance != 0) */
    m = m + 1;
    < put statements here to restore loop invariant >
    /* <Loop Invariant> */
}
```

Try writing the code yourself, before you look at \*F. One snag that you need to avoid is accidently letting balance go less than zero in the last last month’s payment—that would invalidate the loop invariant since the real loan balance will never become negative. As programs become larger, writing a complete proof outline (particularly by hand) becomes more cumbersome, and to some extent even detrimental to correctness because of the clutter that’s introduced. Because of this, \*F is not a complete proof outline, but instead includes. Only enough assertions to make it clear how the program is intended to work. Of these assertions, the most important one is probably the loop invariant, which also played some role in the design process.
/* true */
m = 0;
/* m = 0 */
total_payments = 0;
/* (m = 0) && (total_payments = 0) */
balance = amount;
/***************************************************************************/
* Loop Invariant:  
  * total_payments == Total payments after m months.  
  * balance == Balance of the loan after m months.  
  * m <= Number of months to pay off the loan.  
/***************************************************************************/
while (balance != 0)
{
  /* <Loop Invariant> && (balance != 0) */
  M = M + 1;
  balance = balance * (1+RATE);
  if (PAY > Balance)
    one_payment = balance;
  else
    one_payment = PAY;
  /***************************************************************************/
  * one_payment == the payment due for month m. 
  * balance == Balance of loan after m-1 months, 
  * plus interest charge for month m. 
  * total_payments == Total payments after m-1 months. 
  * m <= Number of months to pay off the loan.  
  /***************************************************************************/
  balance = balance - one_payment;
  total_payments = total_payments + one_payment;

  /* <Loop Invariant> */
};
/* <Loop Invariant> && not (balance != 0) */
/***************************************************************************/
* Postcondition:  
  * m = Months needed to pay off the loan.  
  * total_payments = Total payments during m months.  
/***************************************************************************/
Searching Algorithms

This section examines the correctness of searching algorithms which scan an array for a particular item. The algorithms often fit the pattern that we have already seen: establishing and maintaining a relationship between a control variable and the rest of the data. The methods can also be viewed in an especially visual way, thinking of the array as a landscape which the loop traverses, with each iteration accumulating more knowledge about the lay of the land.

For example, consider a sequential search of a one-dimensional array of real numbers. The number that we’re looking for will be called target, and to simplify things, we’ll assume that we know that target appears somewhere in the array, and it’s just a matter of finding exactly where it is. Thinking of this problem visually, the array itself is a landscape which we’ll walk over looking for the target, with an organized plan. Initially we’ll start at one end of the landscape, and even at that point, we have a bit of knowledge:

Next we’ll step through the landscape, and during each step we’ll check to see whether the target is in the area we just stepped over. Perhaps partway through the array, we still haven’t found the target, but we have accumulated some more information, as shown here:
You know the C++ code for this sequential search, and the visual representation suggests the loop invariant. In the code, data is a double array with an index that begins at 0, and i is a size_t variable.

```c++
    i = 0;
    //*******************************************************************
    // Loop Invariant:
    // Target is in data, but not before data[i].
    //************************************************************************
    while (data[i] != target)
    {
        // Loop invariant; and data[i] != target */
        // target is in data, but not before data[i+1] */
        i = i + 1;
        // Loop invariant */
    }
    // Loop invariant; and not (data[i] != target) */
    // data[i] = target */
```

Can you spot the control variable in this example?—it’s the integer i, which corresponds to how far we’ve travelled through the landscape.

A binary search can also be seen as searching a landscape, but in this case, each iteration of the loop provides much more information, by eliminating fully half of the remaining landscape. Again, we are looking for a number called target, but this time the array is sorted from lowest to highest number. This is the first example where there are actually two control variables: size_t variables low and high, which mark the boundaries of the area which has not yet been searched. At the start of the search, low is at the first index of the array, and high is one step beyond the last index of the array. Here’s the picture:

![binary search landscape diagram]

You’ve seen the body of the binary search loop before: it works by examining the number halfway between the low index and the high index. If that number is smaller than target, then the low index moves up, and otherwise the high index...
moves down. In either case, half of the landscape is eliminated, with just one iteration. For example, if the high index moves down, then the picture looks like this:

Think about the code for implementing the binary search. Can you figure out the loop invariant from the landscapes? Figure 1 on page 4 shows one way to express it, where the number \( n \) tells the number of elements in the array. The low index is maintained in the variable \( a \), and the high index is maintained in the variable \( z \). Actually, the implementation of Figure 1 isn’t too efficient, since it keeps recomputing the expression \((a + z) / 2\)—twice for each iteration. But it is correct, and given the choice of a slow, correct program versus a lightning-fast, incorrect program, which would you choose? Of course, in this case, small changes will make the program both fast and efficient, but we’ll leave that for you.

**SUMMARY AND EXERCISES**

I haven’t written a summary yet, but the work that you have to do for this portfolio is available from the “Portfolio 1” link in http://www.cs.colorado.edu/~main/proglang/quotemedia.html

**REFERENCES**
