SOME SMALL EXAMPLES OF FORMAL VERIFICATION

Assertion: A logical expression involving program variables which should be true when control "passes through" the assertion.

Assertions in this sense are not confined to the programming language; they may use logical or mathematical concepts that we find convenient. They may also introduce variables that are not part of the program. You might think of these as formal comments.

Notation: We write {{ <some logical formula> }} to indicate an assertion. If included in a source file we make it a comment.

{{ name: <some logical formula> }} defines an assertion with a name.

Assignment: <variable> := <expression>

means evaluate <expression> and write the result in <variable>. It's unfortunate that most languages use "=" for this purpose.

The equality relation: = .

Our verifications commonly take the form:

{{ Precondition }}

Program or program fragment

{{ Postcondition }}

The precondition expresses what we know (or assume) at the beginning of the calculation; the postcondition express what we expect to be true at the end of the calculation. Often the program or fragment will incorporate additional assertions which should be true when they are encountered.

We exclude program expressions with side effect in this discussion.
Calculations which involve such expressions can usually be expressed with statements so as to remove the offending expressions.

A very easy example - swapping the values of two variables

```c
int x, y, t;
...
{{ Pswap: x = x0 and y = y0 }}
```

t := x; x := y; y := t;

```c
{{ Qswap: x = y0 and y = x0 }}
```

Note: "x0" and "y0" are sometimes described as "ghost variables". They permit us to talk about the numbers that used to be in x and y.

The rule for verifying an assignment statement:

```c
{{ P }} var := expr {{ Q }}
```

if P[(expr)/var] implies Q

P[(expr)/var] is the result of substituting expr, suitably parenthesized, for var in P.

Think of this as "what P says about the state of things after var is assigned".

We usually use this working backwards. Here's a verification for the swap fragment:

The annotated fragment:

```c
{{ Pswap: x = x0 and y = y0 }}
```

t := x;

```c
x := y;
```
\begin{verbatim}
y := t;

{} {Qswap: x = y0 and y = x0 }

Qswap will be true after y := t if Qswap[t/y] is true before y := t so

{} {Pswap: x = x0 and y = y0 }

t := x;

x := y;

{} {Qswap[t/y]: x = y0 and t = x0 }

y := t;

{} {Qswap: x = y0 and y = x0 }

Now we need Qswap[t/y][y/x] before x := y so

{} {Pswap: x = x0 and y = y0 }

t := x;

{} {Qswap[t/y][y/x]: y = y0 and t = x0 }

x := y;

{} {Qswap[t/y]: x = y0 and t = x0 }

y := t;

{} {Qswap: x = y0 and y = x0 }

Now we need Qswap[t/y][y/x][x/t] before t := x so
\end{verbatim}
{{ Pswap: $x = x_0$ and $y = y_0$ }}

{{ Qswap[t/y][y/x][x/t]: $y = y_0$ and $x = x_0$ }}

t := x;

{{ Qswap[t/y][y/x]: $y = y_0$ and $t = x_0$ }}

x := y;

{{ Qswap[t/y]: $x = y_0$ and $t = x_0$ }}

y := t;

{{ Qswap: $x = y_0$ and $y = x_0$ }}

and it's obvious that $P_{swap}$ implies $Q_{swap}[t/y][y/x][x/t]$!

Now that we've worked through the example we see that the derivation might be expressed more succinctly as

{{ Pswap: $x = x_0$ and $y = y_0$ }}

{{ y = y_0 and $x = x_0$ }}

t := x;                                          // Assignment, [x/t]

{{ y = y_0 and $t = x_0$ }}

x := y;                                          // Assignment, [y/x]

{{ x = y_0 and $t = x_0$ }}

y := t;                                          // Assignment, [t/y]

{{ Qswap: $x = y_0$ and $y = x_0$ }}

Although we worked backward to construct the proof, the inferences proceed forward.
Let's see what happens if we attempt to verify an incorrect version of the program:

```
{{ Pswap: x = x0 and y = y0 }}
```

```
x := y;
t := x;
y := t;
```

```
{{ Qswap: x = y0 and y = x0 }}
```

Working backwards we obtain:

```
{{ Pswap: x = x0 and y = y0 }}
```

```
{{ y = y0 and y = x0 }}  ????  Not implied by Pswap
```

```
x := y;  // Assignment, [y/x]
```

```
{{ x = y0 and x = x0 }}
```

```
t := x;  // Assignment, [x/t]
```

```
{{ x = y0 and t = x0 }}
```

```
y := t;  // Assignment, [t/y]
```

```
{{ Qswap: x = y0 and y = x0 }}
```

In fact, this tells us that the incorrect program will work in the case that \( x = y \) initially, but that's not very helpful.