Assertions and Preconditions

- **Assertions** are used by programmers to verify run-time execution
  - An assertion is a logical formula to verify the state of variables and data to ensure safe continuation
  - A failed assertion should stop the program
  - Assertions are placed by programmers explicitly in code:
    ```
    assert (len>0);
    mean = sum/len;
    ```

- **Preconditions** state the necessary conditions for “safe” execution of part of the program (such as a function)
  - Testing too few conditions (too weak) makes the program unsafe
  - Testing more conditions than necessary (too strong) makes the program less useful
Preconditions and Postconditions

- What if we want to guarantee that the program always produces the correct output assuming that the initial assertion passes?

- Axiomatic semantics: axiomatic proof that a program produces a machine state described by a postcondition if the precondition on the initial state holds (initial assertion passes)

- Example with a postcondition that may fail to pass:
  ```c
  assert (len > 0); // given this passes
  mean = sum/len;
  assert (mean > 0); // is this always true?
  ```

  answer: no, cannot be proven
  can be proven when we change precondition to (len>0 && sum>0)
Preconditions, Postconditions and Partial Correctness

- **Triple notation**: place *conditions* before and after a *command* $C$:

  \[
  \begin{align*}
  \{ \text{Precondition} \} & \quad C \quad \{ \text{Postcondition} \}
  \end{align*}
  \]

- We say that $C$ is *partially correct* with respect to the <precondition,postcondition> specification, provided that:
  - The command $C$ is executed in a machine state that makes the precondition true
  - If the command terminates, then we guarantee that the resulting machine state makes the postcondition true

- *Total correctness* requires termination
Assignment Axiom

- If we view conditions in assertions as logical predicates, the assignment axiom can be stated

\[
\{ P(E) \} \ V := E \ { P(V) \}
\]

that is, if we state a property of \( V \) after the assignment, then the property must hold for expression \( E \) before the assignment.

- We can use substitution to derive the precondition given a postcondition formula \( P \): this is the assignment axiom:

\[
\{ P[V \rightarrow E] \} \ V := E \ { P \}
\]

where \( P[V \rightarrow E] \) denotes the substitution of \( V \) by \( E \) in \( P \).
Examples for Variable Assignments

- \( \{ k = 5 \} \ k := k + 1 \ \{ k = 6 \} \)
  \( (k = 6)[k \rightarrow k+1] \equiv (k+1 = 6) \equiv (k = 5) \)

- \( \{ j = 3 \text{ and } k = 4 \} \ j := j + k \ \{ j = 7 \text{ and } k = 4 \} \)
  \( (j = 7 \text{ and } k = 4)[j \rightarrow j+k] \equiv (j+k = 7 \text{ and } k = 4) \equiv (j = 3 \text{ and } k = 4) \)

- \( \{ \text{true} \} \ x := 2 \ \{ x = 2 \} \)
  \( (x = 2)[x \rightarrow 2] \equiv (2 = 2) \equiv (\text{true}) \)

- \( \{ a > 0 \} \ a := a - 1 \ \{ a > 0 \} \)
  \( (a > 0)[a \rightarrow a - 1] \equiv (a - 1 > 0) \equiv (a > 0) \)  
  Assuming a is int!

- \( \{ \text{false} \} \ y := 1 \ \{ y = 2 \} \)
  No state can satisfy precondition! 
  This is still proven partially correct!
Validity of the Variable Assignment Axiom

- At first glance it seems that working backwards from a postcondition is more complicated than necessary and we are tempted to incorrectly assume we can use

\[
\{ \text{true} \} \; V := E \; \{ V = E \}
\]

- However, consider

\[
\{ \text{true} \} \; m := m + 1 \; \{ m = m + 1 \}
\]

and we find that \( \{ m = m + 1 \} \equiv \{ \text{false} \} \)  

This is wrong.
Statement Composition: Sequence Axiom

- The sequence axiom:

\[ \{ P \} \ C_1 \{ Q \} \ C_2 \{ R \} \]

Q is a postcondition of \( C_1 \) and a precondition for \( C_2 \)

- Written as a rule of inference

\[ \begin{align*}
\{ P \} & \quad C_1 \{ Q \} \\
\{ Q \} & \quad C_2 \{ R \}
\end{align*} \]

\[ \{ P \} \quad C_1 ; C_2 \{ R \} \]
Example Sequencing

- We usually write the sequencing vertical and insert the inferred conditions between the statements:

\[
\begin{align*}
\{ i \geq 0 \} \\
& k := i + 1; \\
\{ k > 0 \text{ and } j = j \} \equiv \{ k > 0 \} \\
& i := j; \\
\{ k > 0 \text{ and } i = j \}
\end{align*}
\]

\[
\begin{align*}
\{ i > 0 \} & k := i + 1 \{ k > 0 \} \quad \{ k > 0 \} & i := j \{ k > 0 \text{ and } i = j \} \\
\{ i > 0 \} & k := i + 1; i := j \{ k > 0 \text{ and } i = j \}
\end{align*}
\]

- The rule of inference:
Skip Axiom

- The ‘skip’ statement is a no-op

\{ P \} \textbf{skip} \{ P \}

pre- and postconditions are identical
If-then-else Axiom

- The *if-then-else axiom* written vertically:

\[
\begin{align*}
\{ P \} \\
\text{if } B \text{ then} \\
\{ P \text{ and } B \} \\
C_1 \\
\{ Q \} \\
\text{else} \\
\{ P \text{ and not } B \} \\\nC_2 \\
\{ Q \} \\
\text{end if} \\
\{ Q \}
\end{align*}
\]
If-then-else Axiom

And as an inference rule:

\[
\{ P \text{ and } B \} \ C_1 \ \{ \ Q \ \}\ \\
\{ P \text{ and not } B \} \ C_2 \ \{ \ Q \ \}
\]

\[\{ P \} \text{ if } B \text{ then } C_1 \text{ else } C_2 \text{ end if } \{ \ Q \ \}\]
The if-then-else Weakest Precondition Rule

- We can derive the weakest precondition $P$ of an if-then-else using:

$$P \equiv (\text{not } B \text{ or } P_1) \text{ and } (B \text{ or } P_2)$$

where $P_1$ is the precondition of $C_1$ given postcondition $Q$ and $P_2$ is the precondition of $C_2$ given postcondition $Q$

- Example:

\[
\{ (x \leq 0 \text{ or } x > 0) \text{ and } (x > 0 \text{ or true}) \} \equiv \{ \text{true} \} \\
\text{if } x > 0 \text{ then} \\
\{ x > 0 \} \\
y := x \\
\text{else} \\
\{ 0 \geq 0 \} \equiv \{ \text{true} \} \\
y := 0 \\
\text{end if} \\
\{ y \geq 0 \} \]

1. Compute preconditions $P_1$ and $P_2$ of $C_1$ and $C_2$

2. Compute precondition $P$ given $P_1$ and $P_2$
Precondition Strengthening

- Logical implication (⇒ or ⊃) means

  stronger condition ⇒ weaker condition
  (more restrictive)     (less restrictive)

- For example:
  - x = y and y = 0 ⇒ y = 0
  - x ≠ 0 ⇒ x = 0 or x < 0 or x > 0
  - x = 0 ⇒ x ≥ 0
  - x = y ⇒ true
  - false ⇒ x = y^2
Using Precondition Strengthening

- We can always make a precondition stronger than necessary to complete a proof

- For example, suppose we know that $x > 0$ and $y = 2$ at the start of the program:

\[
\begin{align*}
\{ x > 0 \text{ and } y = 2 \} &\implies \\
\{ x > 0 \} &
\end{align*}
\]

\[
\begin{align*}
y := x &
\end{align*}
\]

\[
\begin{align*}
\{ y = x \text{ and } y > 0 \} &\equiv (x = x \text{ and } x > 0)
\end{align*}
\]
Loops and Loop Invariants

- A loop-invariant condition is a logical formula that is true before the loop, in the loop, and after the loop
- A familiar example: grocery shopping
- The invariant is:
  \[
  \text{groceries needed} = \text{groceries on list} + \text{groceries in cart}
  \]

\[
\text{cart := empty;}
\]
\[
\{ \text{groceries needed} = \text{groceries on list} + \text{groceries in cart} \}
\Rightarrow \{ \text{groceries needed} = \text{groceries on list} \}
\]

**while** list not empty **do**

\[
\{ \text{groceries needed} = \text{groceries on list} + \text{groceries in cart} \text{ and not empty list} \}
\]
add grocery to cart;

\[
\{ \text{groceries needed} = \text{groceries on list} + \text{groceries in cart} \}
\]
take grocery off list;

\[
\{ \text{groceries needed} = \text{groceries on list} + \text{groceries in cart} \}
\]
**end do**;

\[
\{ \text{groceries needed} = \text{groceries on list} + \text{groceries in cart} \text{ and empty list} \}
\Rightarrow \{ \text{groceries needed} = \text{groceries in cart} \}
\]
While-loop Axiom

- The while-loop axiom uses a loop invariant $I$, which must be determined.
- Invariant cannot generally be automatically computed and must be “guessed” by an experienced programmer.

```plaintext
{ I }
while B do
  { I and B }
  C
  { I }
end do
{ I and not B }
```
While-loop Example (1)

- Loop invariant $I \equiv (f^k! = n! \text{ and } k \geq 0)\{$ $n \geq 0 \}$

  $k := n$;

  $f := 1$;

  while $k > 0$ do
    $f := f^k$;
    $k := k-1$;
  end do

  $\{$ $f = n! \}$

Prove that this algorithm is correct given precondition $n \geq 0$ and postcondition $f = n!$
While-loop Example (2)

- Loop invariant $I \equiv (f^k! = n! \text{ and } k \geq 0)$
  
  $\{ n \geq 0 \}$

  $k := n;$

  $f := 1;$
  
  $\{ f^k! = n! \text{ and } k \geq 0 \}$

  while $k > 0$ do
    
    $\{ f^k! = n! \text{ and } k \geq 0 \text{ and } k > 0 \}$

    $f := f^k;$

  end do

  $\{ f^k! = n! \text{ and } k \geq 0 \}$

  $k := k-1;$
  
  $\{ f^k! = n! \text{ and } k \geq 0 \}$

end do

$\{ f = n! \}$

Add while-loop preconditions and postconditions based on the invariant
While-loop Example (3)

- Loop invariant \( I \equiv (f^k! = n! \text{ and } k \geq 0) \)
  \( \{ n \geq 0 \} \)

  \( k := n; \)
  \( \{ 1^k! = n! \text{ and } k \geq 0 \} \)

  \( f := 1; \)
  \( \{ f^k! = n! \text{ and } k \geq 0 \} \)

  \text{while } k > 0 \text{ do}
  \( \{ f^k! = n! \text{ and } k \geq 0 \text{ and } k > 0 \} \)

  \( f := f^k; \)
  \( \{ f^{(k-1)}! = n! \text{ and } k-1 \geq 0 \} \)

  \( k := k-1; \)
  \( \{ f^k! = n! \text{ and } k \geq 0 \} \)

  \text{end do}

  \( \{ f^k! = n! \text{ and } k \geq 0 \text{ and } k \leq 0 \} \)

  \( \{ f = n! \} \)
While-loop Example (4)

- Loop invariant $I \equiv (f^*k! = n! \text{ and } k \geq 0)$
  
  ```
  { n \geq 0 } 
  { n! = n! \text{ and } n \geq 0 } 
  k := n; 
  { 1^*k! = n! \text{ and } k \geq 0 } 
  f := 1; 
  { f^*k! = n! \text{ and } k \geq 0 } 
  while k > 0 do 
    { f^*k! = n! \text{ and } k \geq 0 \text{ and } k > 0 } 
    { f^*k^*(k-1)! = n! \text{ and } k-1 \geq 0 } 
    f := f^*k; 
    { f^*(k-1)! = n! \text{ and } k-1 \geq 0 } 
    k := k-1; 
    { f^*k! = n! \text{ and } k \geq 0 } 
  end do 
  { f^*k! = n! \text{ and } k \geq 0 \text{ and } k \leq 0 } 
  ```

- Use assignment axioms
While-loop Example (5)

- Loop invariant \( I \equiv (f^*k! = n! \text{ and } k \geq 0) \)
  
  \[
  \begin{align*}
  \{ & n \geq 0 \} \quad \Rightarrow \\
  \{ & n! = n! \text{ and } n \geq 0 \} \equiv \{ n \geq 0 \} \\
  k & := n; \\
  \{ & 1^*k! = n! \text{ and } k \geq 0 \} \\
  f & := 1; \\
  \{ & f^*k! = n! \text{ and } k \geq 0 \} \\
 \text{while } & k > 0 \text{ do} \\
  \{ & f^*k! = n! \text{ and } k \geq 0 \text{ and } k > 0 \} \quad \Rightarrow \\
  \{ & f^*k^*(k-1)! = n! \text{ and } k-1 \geq 0 \} \\
  f & := f^*k; \\
  \{ & f^*(k-1)! = n! \text{ and } k-1 \geq 0 \} \\
  k & := k-1; \\
  \{ & f^*k! = n! \text{ and } k \geq 0 \} \\
 \text{end do} \\
 \{ & f^*k! = n! \text{ and } k \geq 0 \text{ and } k \leq 0 \} \\
 \end{align*}
\]

Use precondition strengthening to prove the correctness of implications
While-loop Example (6)

Loop invariant $I \equiv (f^*k! = n! \text{ and } k \geq 0)$

\[
\begin{align*}
\{ n \geq 0 \} & \Rightarrow \\
\{ n! = n! \text{ and } n \geq 0 \} & \equiv \{ n \geq 0 \} \\
k := n; & \\
\{ 1^*k! = n! \text{ and } k \geq 0 \} & \\
f := 1; & \\
\{ f^*k! = n! \text{ and } k \geq 0 \} & \\
\textbf{while} k > 0 \textbf{ do} \\
\{ f^*k! = n! \text{ and } k \geq 0 \text{ and } k > 0 \} & \Rightarrow \\
\{ f^*k^*k^*(k-1)! = n! \text{ and } k-1 \geq 0 \} & \\
f := f^*k; & \\
\{ f^*(k-1)! = n! \text{ and } k-1 \geq 0 \} & \\
k := k-1; & \\
\{ f^*k! = n! \text{ and } k \geq 0 \} & \\
\textbf{end do} & \\
\{ f^*k! = n! \text{ and } k \geq 0 \text{ and } k \leq 0 \} & \equiv \\
\{ f^*k! = n! \text{ and } k = 0 \} & \Rightarrow \\
\{ f = n! \} & \\
\end{align*}
\]

Use simplification and logical implications to complete the proof.
Using Axiomatic Semantic Rules to find Bugs

- A postcondition specification can be any logical formula.
- A specification that states the input-output requirements of an algorithm is needed to prove correctness.
- A specification that tests a violation can aid in debugging.

For example, if \((n < 0)\) then
\[
\begin{align*}
\text{if } (n < 0) \text{ then} & \quad p = 2; \\
\text{else} & \quad p = n+1; \\
\text{k = m / (p-1);} & \quad \text{ // Div error when } p = 1
\end{align*}
\]

\[
\{ (n > 0 \text{ or true}) \text{ and } (n \leq 0 \text{ or } n \neq 0) \} \equiv \{ \text{true} \}
\]

Means: \(p = 1\) never possible.
Proof of Correctness of the Euclidian Algorithm

- Loop invariant $I \equiv (\gcd(x,y) = \gcd(a,b))$

  \[
  \begin{align*}
  &\{ \text{true} \} \equiv \\
  &\{ \gcd(a,b) = \gcd(a,b) \} \\
  &x := a; \\
  &\{ \gcd(x,b) = \gcd(a,b) \} \\
  &y := b; \\
  &\{ \gcd(x,y) = \gcd(a,b) \} \\
  \end{align*}
  \]

  while $x \neq y$ do

  \[
  \begin{align*}
  &\{ \gcd(x,y) = \gcd(a,b) \text{ and } x \neq y \} \Rightarrow \\
  &\{ (x > y \text{ or } \gcd(x,y-x) = \gcd(a,b)) \text{ and } (x \leq y \text{ or } \gcd(x,y) = \gcd(a,b)) \} \\
  &\text{if } x \leq y \text{ then} \\
  &\{ \gcd(x,y-x) = \gcd(a,b) \} \\
  &\quad y := y - x \\
  &\text{else} \\
  &\{ \gcd(x,y,y) = \gcd(a,b) \} \\
  &\quad x := x - y \\
  &\text{end if} \\
  &\{ \gcd(x,y) = \gcd(a,b) \} \\
  \end{align*}
  \]

  end do

  \[
  \begin{align*}
  &\{ \gcd(x,y) = \gcd(a,b) \text{ and } x = y \} \Rightarrow \\
  &\{ x = \gcd(a,b) \} \\
  \end{align*}
  \]

Use the fact that:
- $\gcd(x-y,y) = \gcd(x,y)$ if $x > y$
- $\gcd(x,y-x) = \gcd(x,y)$ if $x < y$
- $x = \gcd(x,x)$
Concluding Remarks

- Convincingly demonstrated with small algorithms to prove their correctness
- Defining and proving everything formally has not succeeded (at least not yet)
- Proving has not replaced testing (and praying)
- Formal treatment of program correctness sharpens our reasoning skills “why does this program work or fail?”
- Formal reasoning with axiomatic semantics can be found in logic design (hardware), security protocol verification, and distributed system correctness analysis
Some Historical Notes

- Program verification is almost as old as programming (e.g., “Checking a Large Routine”, Turing 1949)
- Floyd had rules for flowcharts in the late ’60s
- Hoare developed axiomatic rules in the ’60s for a simple programming language
- Since then, there have been axiomatic semantics for substantial languages, and many applications
Some Historical Notes

- Software engineering has a reliability challenge
- Testing does not suffice
  - Dijkstra in the ’60s said: “Program testing can be used to show the presence of bugs, but never to show their absence!”
- Program correctness proving is hard
  - Hoare in “An Axiomatic Basis for Computer Programming”, 1969: “Thus the practice of proving programs would seem to lead to solution of three of the most pressing problems in software and programming, namely, reliability, documentation, and compatibility. However, program proving, certainly at present, will be difficult even for programmers of high caliber; and may be applicable only to quite simple program designs.”
Some Historical Notes

Formal language definition assists in the design of better programming languages

- Hoare: “It has been found a serious problem to define these languages [ALGOL, FORTRAN, COBOL] with sufficient rigour to ensure compatibility among all implementors. Since the purpose of compatibility is to facilitate interchange of programs expressed in the language, one way to achieve this would be to insist that all implementations of the language shall “satisfy” the axioms and rules of inference which underlie proofs of the properties of programs expressed in the language, so that all predictions based on these proofs will be fulfilled, except in the event of hardware failure. In effect, this is equivalent to accepting the axioms and rules of inference as the ultimately definitive specification of the meaning of the language.”