Partial correctness refers to a program’s terminating behavior.

We specify partial correctness using **Hoare triples**

\[
\{ P \} \; c \; \{ Q \}
\]

- $c$ is a program
- $P$ and $Q$ are **assertions** in a first-order theory
- Free variables in $P, Q$ can range over program variables
- $P$ is the **precondition** and $Q$ is the **postcondition**
The meaning of \( \{P\} \ c \ \{Q\} \) is as follows:

- If we begin executing \( c \) in an environment satisfying \( P \),
- and if \( c \) terminates,
- then its final environment will satisfy \( Q \)

The specification says nothing about:

- Executions that do not terminate (i.e., diverge)
- Executions that do not begin in \( P \)

**Notice:** \( \{P\} \ c \ \{Q\} \) is a predicate

Goal of verification: prove that it holds, i.e., is a valid Hoare triple
Partial correctness doesn’t require termination

**Total correctness** is a stronger statement, written:

\[[P] \ c \ [Q]\]

The meaning of \([P] \ c \ [Q]\) is:

- If we begin executing \(c\) in an **environment satisfying** \(P\),
- **then** \(c\) **terminates**,
- **and** its final environment will satisfy \(Q\)

Total correctness introduces another obligation for verification
Hoare Logic Inference Rules

**Skip**

\[
\begin{align*}
\{P\} \text{ skip } \{P\}
\end{align*}
\]

**Asgn**

\[
\begin{align*}
\{Q[a/x]\} \; x := a \{Q\}
\end{align*}
\]

**Conseq**

\[
\begin{align*}
P \Rightarrow P' & \quad \vdash \{P'\} \; c \{Q'\} \\
\{P\} \; c \{Q\}
\end{align*}
\]

**Seq**

\[
\begin{align*}
\{P\} \; c_1 \{P'\} \quad \{P'\} \; c_2 \{Q\} \\
\{P\} \; c_1; c_2\{Q\}
\end{align*}
\]

**If**

\[
\begin{align*}
\{P \land b\} \; c_1 \{Q\} \quad \{P \land \neg b\} \; c_2 \{Q\} \\
\{P\} \; \text{if } b \; \text{then } c_1 \; \text{else } c_2 \{Q\}
\end{align*}
\]

**While**

\[
\begin{align*}
\{P \land b\} \; c \{P\} \\
\{P\} \; \text{while } b \; \text{do } c \{P \land \neg b\}
\end{align*}
\]
The proof rules we’ve just covered are sound for partial correctness:

If \( \vdash \{ P \} \ c \ \{ Q \} \), then \( \models \{ P \} \ c \ \{ Q \} \)

If we can derive a triple using the rules, then it is valid

To prove this, we use the operational semantics

Show equivalence between proof rules and reductions

Need to use induction on derivations
Completeness of Hoare Logic

Completeness of Hoare logic is stated as:

\[
\text{If } \models \{P\} c \{Q\}, \text{ then } \vdash \{P\} c \{Q\}
\]

If \( \{P\} c \{Q\} \) is valid, then we can derive it using the rules.

Is this true?

For strengthening, we need to prove statements of the form:

\[ P \Rightarrow Q \]

This requires proving a universal implication in Peano arithmetic.

Recall that \( T_{PA} \) is undecidable!
So, we know there can’t be a proof system that derives all valid triples

The Hoare logic has relative completeness

If we assume an oracle for deciding $P \Rightarrow Q$

Then we can derive any valid Hoare triple for Imp

However, for more complex languages, this isn’t always the case
Working in Hoare logic is nicer than working directly with semantics

But still isn’t “fun”, and not quite trivial

- How to decompose the program?
- When to apply the rule of consequence, and how?
- Lots of tedious details, e.g., discharging
  \[(x = r + (q \times y) \land y \leq r) \Rightarrow x = (r - y) + ((q + 1) \times y)\]
- Even for short programs, the proofs can be long (and boring)

Now we’ll take steps towards mechanizing this process
Given a program $c$ annotated with a specification:

$$\{P\} \ c \ \{Q\}$$

To prove the triple, we’ll generate a set of verification conditions

- VC’s are a function of the code, specification, and loop invariants
- Each VC is a first-order formula in some theory
- If all VC’s are valid, then so is $\{P\} \ c \ \{Q\}$

Program verifiers consist of two main components:

1. Verification condition generator producing $T$-formulas
2. Decision procedure for first-order theory $T$

Intuitively, VC gen. “compiles” a verification problem into a math problem
This is the approach taken by tools like Dafny

As you know, it doesn’t prove things automatically

You often need to provide:

1. Loop invariants
2. Termination metrics
3. Extra assertions
4. Different (but possibly equivalent) contracts
5. Moral support and encouragement

These are all the subject of active research (except 5)

- (1), (2) addressed by static analysis, but unsolvable in general
- (3), (4) are improved by more powerful decision procedures
Given an assertion $Q$ and program $c$, we’ll describe a function:

- That is a **predicate transformer**: produces another assertion
- Assertion for the corresponding precondition $P$ for $c$
- $P$ guaranteed to be the **weakest** such assertion

This is the **weakest precondition** predicate transformer $\text{wp}(c, Q)$

The weakest precondition satisfies the following conditions:

1. The triple $[\text{wp}(c, Q)] \quad c \quad [Q]$ is valid
2. For any $P$ where $[P] \quad c \quad [Q]$ is valid, $P \Rightarrow \text{wp}(c, Q)$

For partial correctness, use **weakest liberal precondition** $\text{wlp}(c, Q)$
“Backwards” VC Generation

Intuitively, \( wlp \) allows us to:

1. Start with a desired postcondition
2. Work backwards to a precondition that must hold
3. Verify that \( P \Rightarrow wlp(c, Q) \)
4. Or just use \( wlp(c, Q) \) to write a contract

There is also a “forward” transformer: \textbf{strongest postcondition}

1. Written \( sp(c, P) \)
2. The triple \( [P] c [sp(c, P)] \) is valid
3. For any \( Q \) where \( [P] c [Q] \) is valid, \( sp(c, P) \Rightarrow Q \)

What does \( sp(c, true) \) characterize?

For the rest of today (and most of the semester), we’ll focus on \( wlp \)
Language Extension: Arrays

But first, let’s add arrays to Imp

\[
\begin{align*}
a & \in \text{AExp} & ::= & \quad n \in \mathbb{Z} \mid x \in \text{Var} \mid a_1 + a_2 \mid a_1 \times a_2 \mid x[a] \\
b & \in \text{BExp} & ::= & \quad \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \land b_2 \\
c & \in \text{Com} & ::= & \quad \text{skip} \mid x := a \mid c_1; c_2 \mid x[a_1] := a_2 \\
& & & \quad \text{if } b \text{ then } c_1 \text{ else } c_2 \\
& & & \quad \text{while } b \text{ do } c
\end{align*}
\]

Allow array lookup, assignment to array indices
The environment can map to array values (i.e., from $T_A$)

**ALookup**

\[
\frac{\langle a, \sigma \rangle \Downarrow n \quad \sigma(x) \text{ is an array}}{\langle x[a], \sigma \rangle \Downarrow \sigma(x)[n]}
\]

**AWrite**

\[
\frac{\langle a_1, \sigma \rangle \Downarrow n_1 \quad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle x[a_1] := a_2, \sigma \rangle \Downarrow \sigma[x \mapsto \sigma(x)[n_1 \triangleleft n_2]]}
\]

Assignment produces a new array using the array update term.
We’ll use Hoare triples to define \(\text{wlp}\)

Recall the rule for assignment:

\[
\text{Asgn} \quad \frac{\{Q[a/x]\} \; x := a\{Q\} }{Q[a/x]} \]

The corresponding transformer is:

\[
\text{wlp}(x := a, Q) = Q[a/x]
\]

This will produce valid preconditions. Is it weak enough?

If \(P \not\models Q[a/x]\), then \(\{P\} \; c \; \{Q\}\) won’t hold
What is \( \text{wlp}(y := x + 1, (\forall x. x < z \rightarrow x < y) \rightarrow x + 1 \leq y) \)?

Applying the definition of \( \text{wlp} \) for assignment,

\[(\forall x. x < z \rightarrow x < x + 1) \rightarrow x + 1 \leq x + 1\]

Is this right?

**No.** When we substituted \( y \) with \( x + 1 \) in

\[(\forall x. x < z \rightarrow x < y)\]

the variable \( x \) was **captured**

Recall: \( x \) is bound in \( \forall x. x < a \rightarrow x < y \)

Outside the scope of this \( \forall \), we’re referring to a different \( x \)
Capture-avoiding substitution

When performing substitution, we need to be careful about scoping.

When we expand out \( P[a/x] \), we need to:

- Only substitute **free occurrences** of \( x \)
- Rename bound variables appearing in \( a \) to avoid capture

Renaming bound variables is called **\( \alpha \)-substitution**

For a substitution \( P[F/G] \), let

\[
V_{\text{free}} = \bigcup_i \text{free}(F) \cup \text{free}(G)
\]

To expand \( P[F/G] \):

- For each quantified variable \( x \) in \( P \) such that \( x \in V_{\text{free}} \), rename \( x \) to a fresh variable to produce \( P' \)
- Apply the substitution directly to \( P' \)
Consider the formula:

\[ F : (\forall x. p(x, y)) \rightarrow q(f(y), x) \]

Suppose we want \( F[y/f(x), q(f(y), x)/(\exists x. h(x, y))] \)

First, we find all the free variables:

\[
\text{free}(y) \cup \text{free}(f(x)) \cup \text{free}(q(f(y), x)) \cup \text{free}(\exists x. h(x, y))
\]

\[
= \{x\} \cup \{x\} \cup \{y\} \cup \{x\} \cup \{x, y\} \cup \{y\}
\]

\[
= \{x, y\}
\]

\( F \) has one quantified variable \( x \), which is in this set. So:

\[ F : (\forall x'. p(x', y)) \rightarrow q(f(y), x) \]

Applying the substitution:

\[ (\forall x'. p(x', f(x))) \rightarrow \exists x. h(x, y) \]
Example

\[ F' : (\forall x'. p(x', y)) \rightarrow q(f(y), x) \]

Applying the first substitution \( q(f(y), x)/(\exists x. h(x, y)) \):

\[ F' : (\forall x'. p(x', y)) \rightarrow \exists x. h(x, y) \]

Now we apply the second substitution: \( y/f(x) \)

We need to rename bound variables that occur in \( V_{\text{free}} \):

\[ F'' : (\forall x'. p(x', y)) \rightarrow \exists x'. h(x', y) \]

Which brings us to:

\[ F'' : (\forall x'. p(x', f(x))) \rightarrow \exists x'. h(x', f(x)) \]
Defining $wlp(c, Q)$: Array Assignment

Array assignment gives the rule

$$\text{AsgnArr} \quad \frac{\{Q[x\langle a_1 \triangleleft a_2 \rangle/x]\} \ q[a_1] := a_2 \ \{Q\}}$$

The corresponding transformer is:

$$wlp(x[a_1] := a_2, Q) = Q[a\langle a_1 \triangleleft a_2 \rangle/x]$$

This is no different than normal assignment

In this case, we’re working with array values rather than integers
Let’s compute:

\[ \text{wlp}(b[i] := 5, b[i] = 5) \]

Proceeding with the substitution,

\[ \text{wlp}(b[i] := 5, b[i] = 5) = (b\langle i < 5 \rangle[i] = 5) = (5 = 5) = \text{true} \]
Let’s compute:

\[ \text{wlp}(b[n] := x, \forall i. 1 \leq i < n \rightarrow b[i] \leq b[i + 1]) \]

Proceeding with the substitution,

\[ \text{wlp}(b[n] := x, \forall i. 1 \leq i < n \rightarrow b[i] \leq b[i + 1]) \]

\[ = \forall i. 1 \leq i < n \rightarrow (b\langle n < x \rangle)[i] \leq (b\langle n < x \rangle)[i + 1] \]

\[ = (b\langle n < x \rangle)[n - 1] \leq (b\langle n < x \rangle)[n] \]

\[ \wedge \forall i. 1 \leq i < n \rightarrow (b\langle n < x \rangle)[i] \leq (b\langle n < x \rangle)[i + 1] \]

\[ = b[n - 1] \leq x \wedge \forall i. 1 \leq i < n - 1 \rightarrow b[i] \leq b[i + 1] \]
Defining $\text{wlp}(c, Q)$: Sequencing

The corresponding transformer is:

$$\text{Seq} \begin{array}{ccc} \{P\} & c_1 & \{P'\} \\ \{P\} & c_1; c_2 & \{Q\} \end{array}$$

The corresponding transformer is:

$$\text{wlp}(c_1; c_2, Q) = \text{wlp}(c_1, \text{wlp}(c_2, Q))$$

Compose the transformer, working backwards from $c_2$

The precondition for $c_2$ becomes the postcondition for $c_1$

Note: we don’t need to use consequence to “glue” these together
Defining \( \text{wlp}(c, Q) \): Conditional

\[
\text{If } \begin{array}{c}
\{ P \land b \} c_1 \{ Q \} \\
\{ P \land \neg b \} c_2 \{ Q \}
\end{array} \\
\{ P \} \text{ if } b \text{ then } c_1 \text{ else } c_2 \{ Q \}
\]

\[
\text{wlp}(\text{if } b \text{ then } c_1 \text{ else } c_2, Q) = (b \rightarrow \text{wlp}(c_1, Q)) \land (\neg b \rightarrow \text{wlp}(c_2, Q))
\]

If \( b \) holds, then wlp of \( c_1 \) branch must hold

Otherwise if \( \neg b \), then wlp of \( c_2 \) branch must hold

Why isn’t the formula a disjunction?
Defining $\text{wlp}(c, Q)$: While Loop

While \[ \frac{\{P \land b\} \quad c \quad \{P\}}{\{P\} \quad \text{while } b \text{ do } c \quad \{P \land \neg b\}} \]

Recall the equivalence:

$$\text{while } b \text{ do } c \equiv \text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip}$$

Let’s apply $\text{wlp}$ for $\text{if}$:

$$\text{wlp}(\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip}, Q)$$

$$= (b \rightarrow \text{wlp}(c; \text{while } b \text{ do } c, Q)) \land (\neg b \rightarrow Q)$$

$$= (b \rightarrow \text{wlp}(c, \text{wlp}(\text{while } b \text{ do } c, Q))) \land (\neg b \rightarrow Q)$$

$$= (b \rightarrow \text{wlp}(c, \text{wlp}(\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip}, Q))) \land (\neg b \rightarrow Q)$$

We haven’t accomplished anything
Approximate Weakest Precondition

In general, we can’t always compute \( wlp \) for loops. Instead, we’ll approximate it with help from annotations.

Now we’ll assume loops have the syntax:

\[
\textbf{while} \; b \; \textbf{do} \; \{I\} \; c
\]

\( I \) is a loop invariant provided by the programmer.

The approximate \( wlp \) for \textbf{while} will still be a valid precondition.

But it may not be the weakest precondition: even if

\[
\{P\} \; \textbf{while} \; b \; \textbf{do} \; c \; \{Q\}
\]

is valid, it might not be that:

\[
P \Rightarrow wlp(\textbf{while} \; \{I\} \; b \; \textbf{do} \; c, \; Q)
\]
Approximate *wlp*: While Loop

Now, suppose we define:

\[
\text{wlp} (\text{while} \ \{I\} \ b \ \text{do} \ c, Q) = I
\]

A couple of things are missing:

- We haven’t checked that \( I \land \neg b \) gives us \( Q \)
- We don’t know that \( I \) is actually a loop invariant

We need to track additional verification conditions: \( vc(c, Q) \)
Verification Conditions: While Loop

If we define

\[ \text{wlp(while } \{I\} \ b \ \text{do c, Q} = I \]

Then we still need to show that

- \( I \land \neg b \) establishes \( Q \)
- \( I \) is a loop invariant

Defining the set of verification conditions,

\[ \text{vc(while } \{I\} \ b \ \text{do c, Q} = \left\{ \begin{array}{l}
I \land \neg b \Rightarrow Q \\
I \land b \Rightarrow \text{wlp(c, Q)}
\end{array} \right\} \]

To summarize, for \( Q \) to hold after executing a loop:

1. Each formula in \( \text{vc(while } \{I\} \ b \ \text{do c, Q} \) must be valid
2. \( \text{wlp(while } \{I\} \ b \ \text{do c, Q} = I \) must be valid
**while** is the only command that introduces new conditions

But other statements might contain loops

Need to define $\text{vc}$ for them as well:

- $\text{vc}(x := a, Q) = \emptyset$
- $\text{vc}(c_1; c_2, Q) = \text{vc}(c_1, \text{wlp}(c_2, Q)) \cup \text{vc}(c_2, Q)$
- $\text{vc}($if $b$ then $c_1$ else $c_2, Q) = \text{vc}(c_1, Q) \cup \text{vc}(c_2, Q)$

In short, compound statements collect conditions from constituents
Bringing all of this together, we can verify

\[ \{P\} \ c \ \{Q\} \]

for an annotated program \(c\)

1. Compute \(P' = \text{wlp}(c, Q)\)
2. Compute \(\text{vc}(c, Q)\)
3. Check validity of \(P \rightarrow P'\)
4. Check validity of each \(F \in \text{vc}(c, Q)\)

If (3) and (4) pass, then \(\{P\} \ c \ \{Q\}\) is valid

If \(\{P\} \ c \ \{Q\}\) is valid, then will (3) and (4) pass?  
**No.** Loop invariants might be too weak!
Example

Let’s verify the example from last lecture:

\[
\{true\}
\]

\[
r := x; q := 0;
\]

**while** \( y \leq r \) **do**

\[
r := r - y; q := q + 1
\]

\[
\{r < y \land x = r + (q \times y)\}
\]

Recall our loop invariant:

\[
\{true\}
\]

\[
r := x; q := 0;
\]

**while** \( y \leq r \) **do**

\[
x = r + (q \times y)
\]

\[
r := r - y; q := q + 1
\]

\[
\{r < y \land x = r + (q \times y)\}
\]
Define the following shorthand:

- $c_1 : r := x$
- $c_2 : q := 0$
- $c_3 : r := r - y$
- $c_4 : q := q + 1$
- $c_5 : \text{while} \ y \leq r \ \text{do} \ c_3; c_4$

We need to show these are valid:

- $true \Rightarrow \text{wlp}(c_1; c_2; c_5, r < y \land x = r + (q \times y))$
- $\text{vc}(c_1; c_2; c_5, r < y \land x = r + (q \times y))$

We’ll start with $true \Rightarrow \text{wlp}(c_1; c_2; c_5, r < y \land x = r + (q \times y))$
Example

\[
\text{true} \Rightarrow \text{wlp}(c_1; c_2; c_5, r < y \land x = r + (q \times y))
\]

Let’s use \( Q : r < y \land x = r + (q \times y) \), \( I : x = r + (q \times y) \)

We begin by applying the rule for composition twice:

\[
\text{wlp}(c_1; c_2; c_5, Q) = \text{wlp}(c_1, \text{wlp}(c_2, \text{wlp}(c_5, Q)))
\]

This brings us to \( \text{wlp}(c_5, Q) : \)

\[
\text{wlp}(\text{while } y \leq r \text{ do } \{I\} \ c_3; c_4, Q) = I
\]

We also have verification conditions:

\[
\text{vc}(c_5, Q) = \{I \land \neg b \Rightarrow Q, I \land b \Rightarrow \text{wlp}(c_3; c_4, Q)\}\]
Let’s work out the VC $I \land b \Rightarrow \text{wlp}(c_3; c_4, Q)$

We have that:

\[
\text{wlp}(r := r - y; q := q + 1, Q) = \text{wlp}(r := r - y, \text{wlp}(q := q + 1, Q)) \\
= \text{wlp}(r := r - y, Q[q/q + 1]) \\
= \text{wlp}(r := r - y, r < y \land x = r + ((q + 1) \times y)) \\
= (x = (r - y) + ((q + 1) \times y))
\]

So, we have:

\[
\text{vc}(c_5, Q) = \{I \land \neg b \Rightarrow Q, I \land b \Rightarrow (x = (r - y) + ((q + 1) \times y))\}
\]
Recalling that $wlp(c_5, Q) = I$, we now need $wlp(c_2, I)$:

$$wlp(q := 0, x = r + (q \times y)) = (x = r + (0 \times y)) = x = r$$

Moving on, our final step is $wlp(c_1, x = r)$:

$$wlp(r := x, x = r) = (x = x)$$

Popping back to our top-level procedure:

1. Compute $P' = wlp(c, Q)$
   
   $$P' = (x = x)$$

2. Compute $vc(c, Q)$
   
   $$vc(c, Q) = \{ I \land \neg b \Rightarrow Q, I \land b \Rightarrow (x = (r - y) + ((q + 1) \times y)) \}$$

3. Check validity of $P \rightarrow P'$
   
   Clearly, $true \Rightarrow (x = x)$

4. Check validity of each $F \in vc(c, Q)$
Example

Check validity of each $F \in \text{vc}(c, Q)$:

$$\text{vc}(c, Q) = \left\{ \begin{array}{l} x = r + (q \times y) \land \neg (y \leq r) \Rightarrow r < y \land x = r + (q \times y) \\ x = r + (q \times y) \land y \leq r \Rightarrow (x = (r - y) + ((q + 1) \times y)) \end{array} \right\}$$

The first is true because $\neg (y \leq r) \iff r < y$

The second we get by algebraic calculation

Therefore, the triple is valid

$$\{\text{true}\}$$

$r := x; q := 0;$

while $y \leq r$ do

$r := r - y; q := q + 1$

$r < y \land x = r + (q \times y)$
Next Lecture

- We’ll add procedures to the language
- Start on total correctness
- **Mid-term 1 week from now**
- We’ll post a study guide by the weekend
- **Assignment 3 due date pushed to October 25**